Instructions:
Language of the Computer

Instruction Set

- The repertoire of **instructions of a computer**
  (vs. human-oriented “high-level” programming language)
- Different computers have different instruction sets
  - but with many aspects in **common**
- Early computers had very simple instruction sets
  - similar to this simplified implementation
- Many modern computers also have **simple** instruction sets
  - Easier hardware and compiler optimization
  - CISC (**C**omplex **I**nstruction **S**et **C**omputer) vs. **RISC** (**R**educed **I**nstruction **S**et **C**omputer)

CISC (IBM 370 MoVe Characters Long – MVCL)

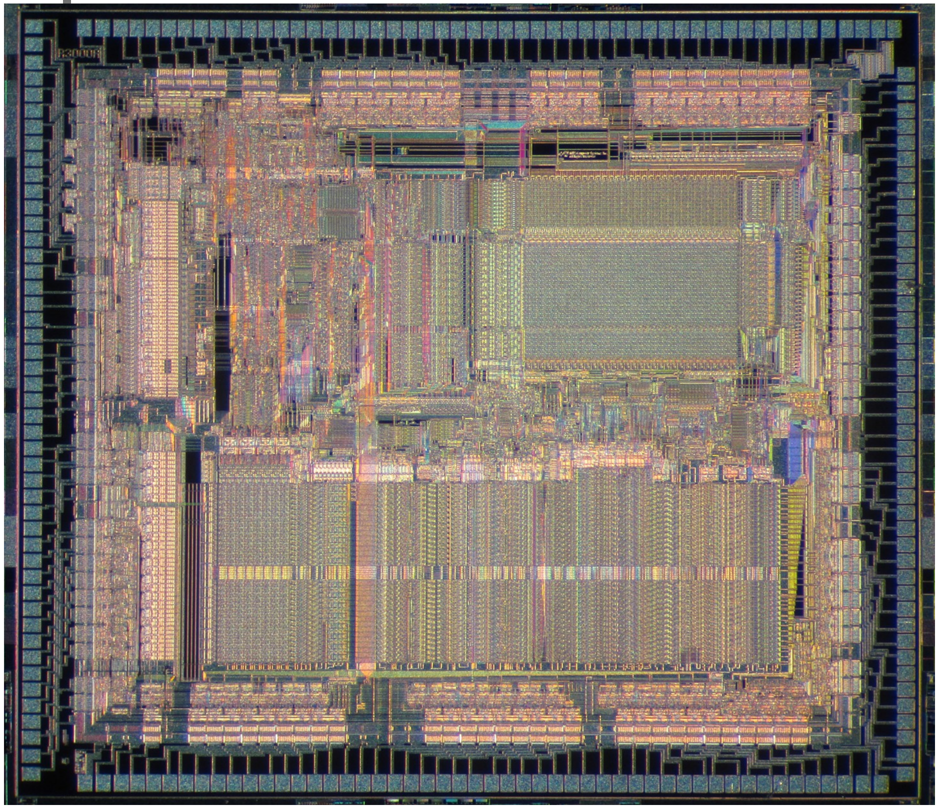
```
         LA     R4,FIELDA          POINT AT TARGET FIELD WITH EVEN REG
         L      R5,LENGTHA         PUT LENGTH OF TARGET IN ODD REG
         LA     R6,FIELDB          POINT AT SOURCE FIELD WITH EVEN REG
         L      R7,LENGTHB         PUT LENGTH OF SOURCE IN ODD REG
         ICM    R7,B'1000',BLANK   INSERT A SINGLE BLANK PAD CHAR IN ODD REG
         MVCL   R4,R6
         …
FIELDA   DC     CL2000' '
BDATA    DC     1000CL1'X'
         ORG    BDATA
FIELDB   DS     CL1000
LENGTHA  DC     A(L'FIELDA)        CREATE AN ADDRESS CONSTANT THAT IS A LENGTH
LENGTHB  DC     A(L'FIELDB)        CREATE AN ADDRESS CONSTANT THAT IS A LENGTH
BLANK    DC     C' '
```

http://csc.columbusstate.edu/woolbright/Instructions/MVCL.HTM

The MIPS Instruction Set

- Used as example throughout the book
  MIPS-32 (vs. MIPS-64)
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of **embedded** core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
    Past: Silicon Graphics workstations
  - General purpose: **Intel** architecture
- Typical of many modern RISC Instruction Set Architectures (**ISA**s)

32 bit MIPS R3000 processor (115000 transistors)



early 1990s

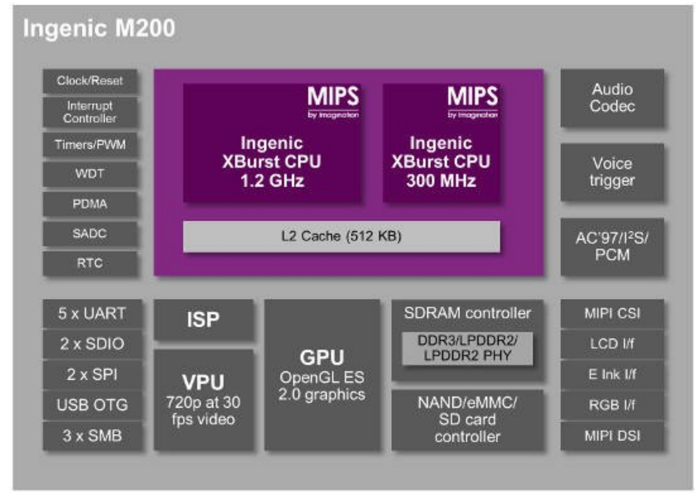
https://www.mips.com/blog/five-most-iconic-devices-to-use-mips-cpus/

MIPS
by Imagination

The future of MIPS?

MIPS Goes Open Source

https://www.cdrinfo.com/d7/content/mips-goes-open-source

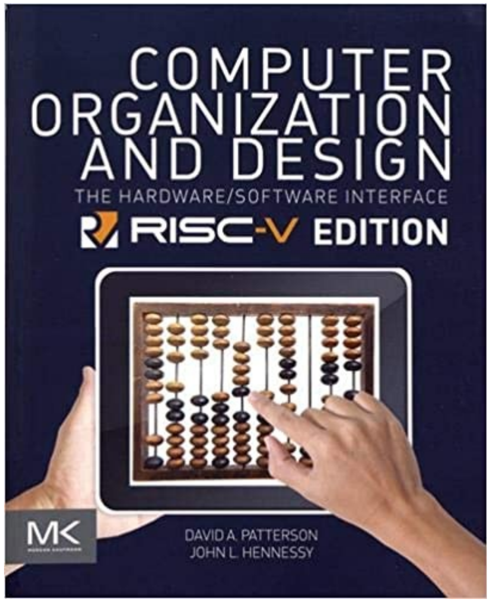


Smart watch **SoC** has dual MIPS cores

open source ISA of the future

https://riscv.org/

MIPS (the company) will build RISC-V processors



The MIPS Instruction Set

- **Human**-readable form:

  assembly language
  (without/with pseudo-instructions)
- **Machine**-readable form:

  machine language (binary)
- **Translation** between both

  by “assembler” (a low-level, very simple compiler)

asm.py assembler (not MIPS)

```
pushi 1      #0  zet 1 op de stack
peek r1      #1  zet 1 in R1, door peek, 1 blijft op de stack
pop r2       #2  zet 1 in R2, door pop, 1 verdwijnt van stack
add r1 r2    #3  r1 = r1 + r2
push r1      #4  zet r1 op de stack
add r2 r1    #5  r2 = r2 + r1
push r2      #6  zet r2 op de stack
j 3          #7  repeat vanaf stap 3
```

```python
def parseLine(line, data):
    tokens = line.split(" ")
    instr = tokens[0].lower()
    if ("or" == instr):
        data += "0"
        data += str(hex((get_register(instr, tokens[1]) << 5) + (get_register(instr, tokens[2]) << 2)))[2:]
    elif ("and" == instr):
        data += "1"
        data += str(hex((get_register(instr, tokens[1]) << 5) + (get_register(instr, tokens[2]) << 2)))[2:]
    elif ("add" == instr):
        data += "2"
        data += str(hex((get_register(instr, tokens[1]) << 5) + (get_register(instr, tokens[2]) << 2)))[2:]
    elif ("sub" == instr):
        data += "3"
        data += str(hex((get_register(instr, tokens[1]) << 5) + (get_register(instr, tokens[2]) << 2)))[2:]
    elif ("ori" == instr):
        data += "4"
        data += str(hex((get_register(instr, tokens[1]) << 5) + get_immediate(instr, tokens[2])))[2:]
    elif ("andi" == instr):
        data += "5"
        data += str(hex((get_register(instr, tokens[1]) << 5) + get_immediate(instr, tokens[2])))[2:]
    elif ("addi" == instr):
        data += "6"
        data += str(hex((get_register(instr, tokens[1]) << 5) + get_immediate(instr, tokens[2])))[2:]
    elif ("subi" == instr):
        data += "7"
        data += str(hex((get_register(instr, tokens[1]) << 5) + get_immediate(instr, tokens[2])))[2:]
    elif ("sw" == instr):
        data += "8"
        offset = str(hex(get_offset(instr, tokens[1])))[2:]
        if len (offset) == 1:
            data += "0"
        data += offset
    elif ("lw" == instr):
        data += "9"
        offset = str(hex(get_offset(instr, tokens[1])))[2:]
        if len (offset) == 1:
            data += "0"
        data += offset
    elif ("ldi" == instr):
        data += "a"
        offset = str(hex(get_offset(instr, tokens[1])))[2:]
        if len (offset) == 1:
            data += "0"
        data += offset
    elif ("cp" == instr):
        data += "b"
        data += str(hex(get_register(instr, tokens[1]) << 1))[2:]
        data += "0"
    elif ("b" == instr):
        data += "c"
        offset = str(hex(get_offset(instr, tokens[1])))[2:]
        if len (offset) == 1:
            data += "0"
        data += offset
```

```
v2.0 raw
51 64 78 26 44 29 48 c3
```

Arithmetic Operations

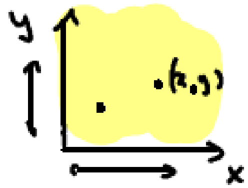
- Add and subtract, **three operands**
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

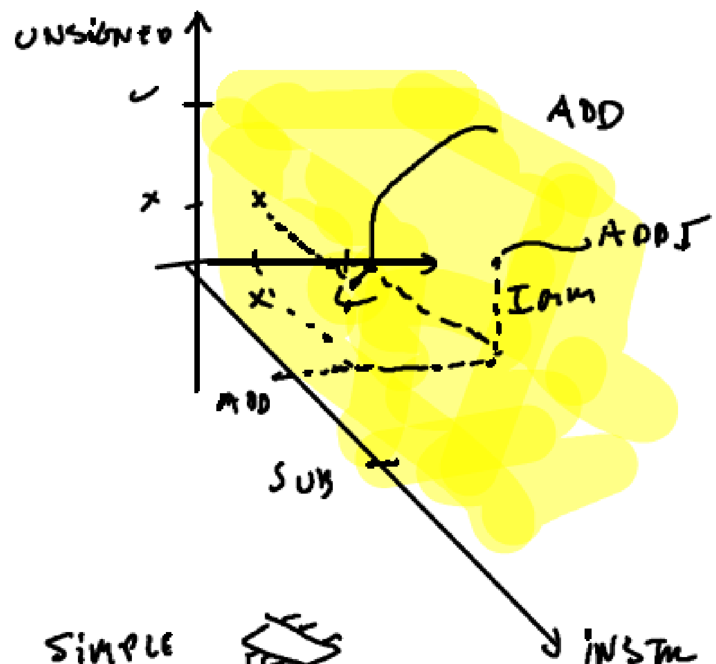
- All arithmetic operations have this “Three-Address Code” (TAC, 3AC) form
- ***Design Principle 1:***

  Simplicity favours **regularity**
  - regularity makes implementation simpler
  - enables higher performance at lower cost
  - ~ **orthogonality** of instruction set

orthogonality of InstructionSet



INSTR		I	U	UI
ADDITION	ADD	ADDI	ADDU	ADDIU
SUBTRACTION	SUB	SUBI	SUBU	SUBIU
LOAD	LD	LDI		



- x IMPLEMENTATION SIMPLE
- x COGNITIVE EASE 

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled to MIPS code (almost):

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Register Operands

- Arithmetic instructions use **register** operands
- MIPS has a 32 × 32-bit **register file**
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data is called a "word"
- Assembler names (convention)
  - `$t0, $t1, …, $t9` for temporary values
  - `$s0, $s1, …, $s7` for saved values

`$t0 – $t7`	denote registers	8 – 15
`$t8 – $t9`	denote registers	24 – 25
`$s0 – $s7`	denote registers	16 – 23

- ***Design Principle 2:* Smaller** is faster
  - Signals travel smaller distance
  - Smaller instructions (uses less memory)

Register Operand Example

- C code:

```
int f, g, h, i, j;
f = (g + h) - (i + j);
```

with `f, …, j` in `$s0, …, $s4`

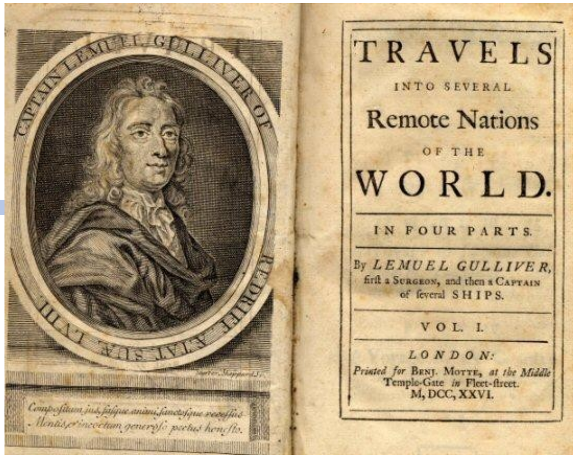
- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Memory Operands

- **Main memory** used for **composite data**
  - Arrays, structures, dynamic data
- To apply arithmetic **operations**
  1. **Load** values from memory into registers
  2. Perform **operation**
  3. **Store** result from register to memory
- (data) memory is **byte addressed**
  - Each address identifies an 8-bit byte
- Words (= 4 bytes) are **aligned** in memory
  - Address must be a multiple of 4 (see `.align`)
- MIPS implements **Big Endian** storage
  - Most-significant **byte** at least address of a word
  - Little Endian: **least**-significant **byte** at **least** address

Endian-ness

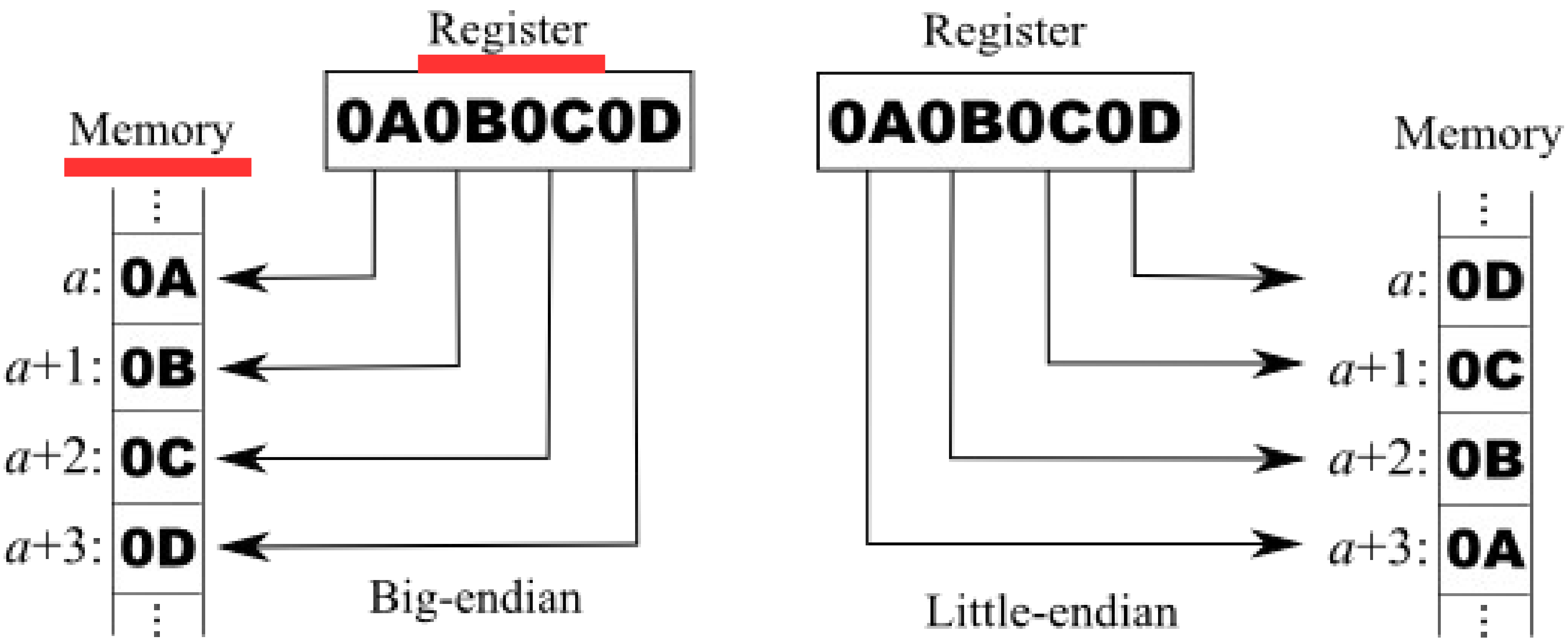


(Jonathan) Swift's point is that the difference between breaking the egg at the little-end and breaking it at the big-end is trivial. Therefore, he suggests, that everyone does it in his own preferred way.

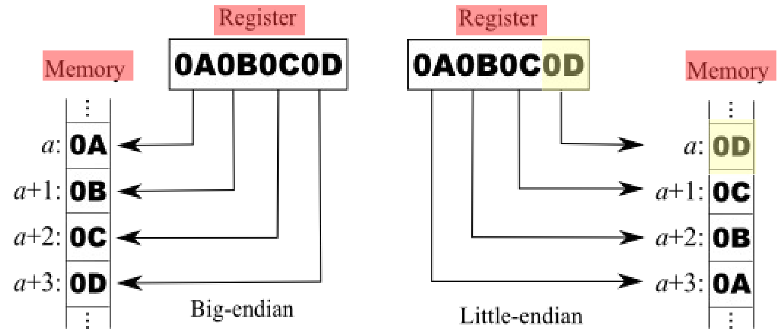
Danny Cohen IEN 137 **1 April** 1980

http://www.ietf.org/rfc/ien/ien137.txt

IETF == Internet Engineering Task Force
RFC == Request For Comments



Endian-ness




```
/* endian.c */

#include <stdio.h>
int main(void)
{
 register int reg_i= 0x0A0B0C0D;
 int i = reg_i;
 /* https://cplusplus.com/reference/cstdio/printf/ */
 printf("0x%08X: 0x%02X\n", (unsigned char *)(&i)  , * ((unsigned char *)(&i)  ));
 printf("0x%08X: 0x%02X\n", (unsigned char *)(&i)+1, * ((unsigned char *)(&i)+1));
 printf("0x%08X: 0x%02X\n", (unsigned char *)(&i)+2, * ((unsigned char *)(&i)+2));
 printf("0x%08X: 0x%02X\n", (unsigned char *)(&i)+3, * ((unsigned char *)(&i)+3));
 return(0);
}
```

```
hv@roke% ./endian          hv@roke% ./endian
0x38D4BF7C: 0x0D           0x6FEC561C: 0x0D
0x38D4BF7D: 0x0C           0x6FEC561D: 0x0C
0x38D4BF7E: 0x0B           0x6FEC561E: 0x0B
0x38D4BF7F: 0x0A           0x6FEC561F: 0x0A
```

```
hv@roke% lscpu
Architecture:       x86_64
CPU op-mode(s):     32-bit, 64-bit
Address sizes:      39 bits physical, 48 bits virtual
Byte Order:         Little Endian
```

Unicode string (en/de)coding

```
>>> ord('a'.encode('UTF-8'))
97                                  # < 127 → fits in 7 bit, compatible with ASCII

>>> 'a'.encode("UTF-8")              # compatible with ASCII
b'a' == b'\x61'                      # b'...' means byte literal, not string

>>> '€'.encode('UTF-8')              # variable length: 8, 16, 24, or 32 bit
b'\xe2\x82\xac'                      # write in binary and see UTF-8 pattern specification
```

```
>>> '€'.encode('UTF-16-LE')
b'\xac '                             # variable length: 16 or 32 bit

>>> 'a'.encode("UTF-16-LE")
b'a\x00'                             # not compatible with ASCII, embedded \x00
```

```
>>> '€'.encode('UTF-32')             # 32 bit, not compatible with ASCII, embedded \x00
b'\xff\xfe\x00\x00\xac \x00\x00'     # 64 bit? … 4 byte Byte Order Mark (BOM)

>>> '€'.encode('UTF-32-LE')          # LE = Little Endian
b'\xac \x00\x00'                     # 32 bit
```

```
>>> b'\xe2\x82\xac'.decode('UTF-8')
'€'

>>> b'\xff\xfe\xac '.decode('UTF-16')
'€'

>>> b'\xff\xfe\x00\x00\xac \x00\x00'.decode('UTF-32')
'€'
```

Memory Operand Example 1

- `C` code:

  `g = h + A[8];`

  `g` in `$s1`, `h` in `$s2`,
  base address of `A` in `$s3`

- Compiled code for MIPS architecture:

  Index 8 (words) requires offset of 32 bytes

  (4 bytes per word)

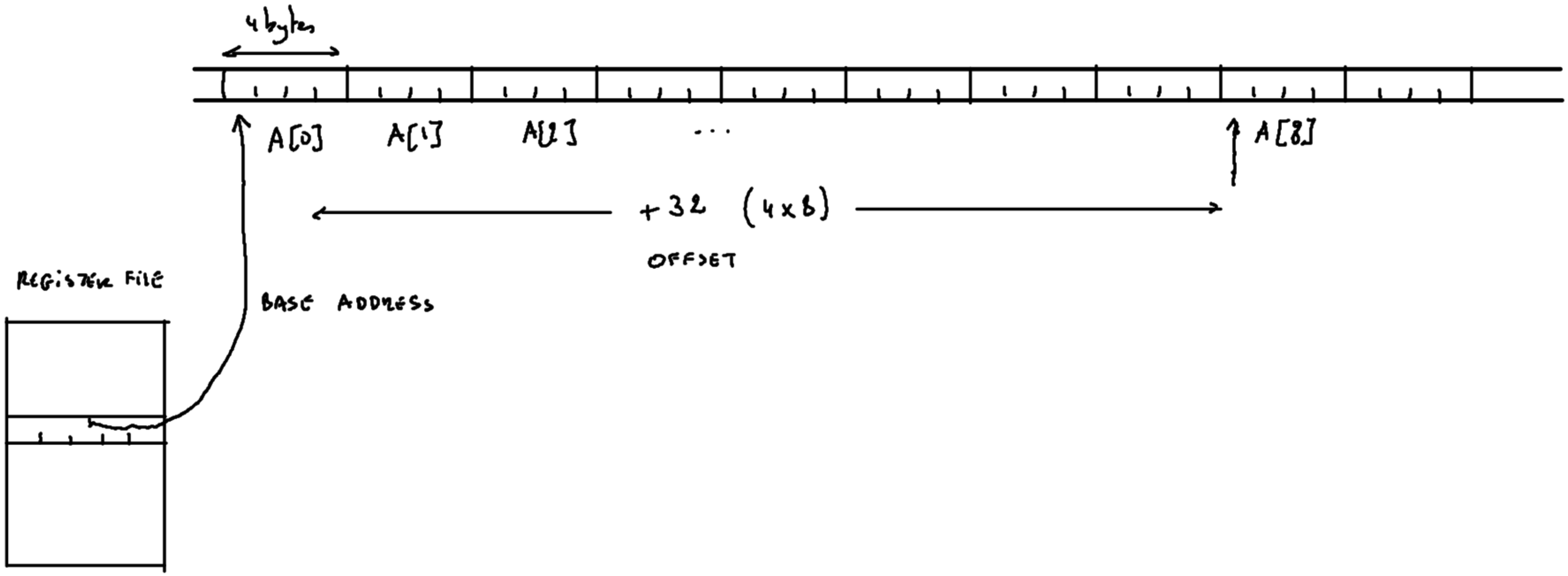
```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```

offset

base register (past: **index** register)

Memory Operand Example 1

data memory



Memory Operand Example 2

- `C` code:

  ```
  A[12] = h + A[8];
  h in $s2,
  base address of A in $s3
  ```

- Compiled MIPS code:

  ```
  lw  $t0, 32($s3)      # load word
  add $t0, $s2, $t0
  sw  $t0, 48($s3)      # store word
  ```

Registers vs. (“main”) Memory

- Registers are **faster** to access than RAM memory
- **Operating on memory data**
  requires loads and stores

  → **more instructions** to be executed

  → Compiler must **use registers** for variables
  as much as possible
  - Only “**spill”** to memory
    for less frequently used variables
  - Register use optimization is important!

  “register allocation”

Immediate Operands

- Constant data specified **in** an instruction

  `addi $s3, $s3, 4`   ~ **orthogonality**

- No subtract immediate instruction (only pseudo-)
  - Just use a negative constant

  subi $s2, $s1, **10** → **add**i $s2, $s1, −**10**

- ***Design Principle 3:***

  Make the **common case fast**
  - Common:
    - 50% of SPEC2006 instructions: immediate
    - small constants (fit in 16bit, 2's complement)
  - Fast:
    - immediate operand avoids one (load) instruction

The Constant Zero

- **MIPS** register `0($zero)` is the constant `0`
  - Cannot be overwritten
- Useful for common operations
  - *e.g.*, **move** between registers

```
move $t2, $s1
```

is a "pseudo-instruction" implemented as

```
addu $t2, $s1, $zero
```

Sign Extension

- Representing a number **using more bits**
  - **preserve** the **numeric value**
- In MIPS instruction set, in datatpath
  - `addi`: extend immediate value
  - `lb, lh`: extend loaded byte/halfword
  - `beq, bne`: extend the displacement/offset from `PC+4`
- Replicate the **sign bit** to the left
  - unsigned values: extend with `0`s, else `1`s
- Examples: 8-bit to 16-bit
  - `+2: 0000 0010  →  0000 0000 0000 0010`
  - `–2: 1111 1110  →  1111 1111 1111 1110`

Representing Instructions

- Instructions are encoded in **binary**
  - Called **machine code** (vs. assembly code – text)
- MIPS instructions
  - **Encoded** as 32-bit instruction words
  - Small number of **formats** encoding operation code (opcode), register numbers, …
  - **Regularity**!
- Register numbers
  - `$t0 – $t7` denote registers `8 – 15`
  - `$t8 – $t9` denote registers `24 – 25`
  - `$s0 – $s7` denote registers `16 – 23`

①

MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R	opcode (31–26)	rs (25–21)	rt (20–16)	rd (15–11)	shamt (10–6)	funct (5–0)
I	opcode (31–26)	rs (25–21)	rt (20–16)	immediate (15–0)		
J	opcode (31–26)	address (25–0)				

ARITHMETIC CORE INSTRUCTION SET

②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR	opcode (31–26)	fmt (25–21)	ft (20–16)	fs (15–11)	fd (10–6)	funct (5–0)
FI	opcode (31–26)	fmt (25–21)	ft (20–16)	immediate (15–0)		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - `op`: operation code (opcode)
  - `r`**`s`**: first **source** register number
  - `rt`: second source register number
  - `r`**`d`**: **destination** register number
  - `shamt`: shift amount (`00000` for now)
  - `funct`: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

`add $t0, $s1, $s2`

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - `rs`: source register number
  - `r`**`t`**: **target** register number
  - `constant` (two's complement 16 bit) in $[-2^{15}, +2^{15} - 1]$
  - `address`: **offset** added to base address in `rs`
- ***Design Principle 4:***

  Good design demands good ***compromises***
  - Different formats complicate decoding hardware, but allow 32-bit instructions uniformly
  - Do keep formats as similar as possible

Logical Operations

Instructions for bitwise manipulation

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Useful for extracting and inserting groups of bits in a word

Shift Operations

Shift Left Logical	`sll`	R	R[rd] = R[rt] << shamt	0 / 00_{hex}
Shift Right Logical	`srl`	R	R[rd] = R[rt] >>> shamt	0 / 02_{hex}

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `shamt`: how many positions to shift (unsigned)
  why 5 bits?
- shift left logical
  - shift left and fill with `0` bits
  - `sll` by *i* bits multiplies by 2^i (`int` only)
- shift right logical (vs. `sra` shift right arithmetic)
  - shift right and fill with `0` bits
  - `srl` by *i* bits divides by 2^i (`unsigned int` only)

AND Operations

Useful to **mask** bits in a word:
select some bits, **clear** others to 0

```
and $t0, $t1, $t2
```

`$t2`	0000 0000 0000 0000 0000 1101 1100 0000
`$t1`	0000 0000 0000 0000 0011 1100 0000 0000
`$t0`	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

Useful to **include** bits in a word
set some bits to **1**, leave others **unchanged**

```
or $t0, $t1, $t2
```

`$t2`	0000 0000 0000 0000 0000 1101 1100 0000
`$t1`	0000 0000 0000 0000 0011 1100 0000 0000
`$t0`	0000 0000 0000 0000 0011 1101 1100 0000

Bit operations are commonly used in 2D games where “sprites” are put on a background using BLT (Bit Block Transfer)
https://en.wikipedia.org/wiki/Bit_blit

NOT Operations

- Useful to **invert** bits in a word (`0/1`)

  `not $t0, $t1`

- MIPS has the NOR 3-operand instruction

  a NOR b == NOT (a OR b)

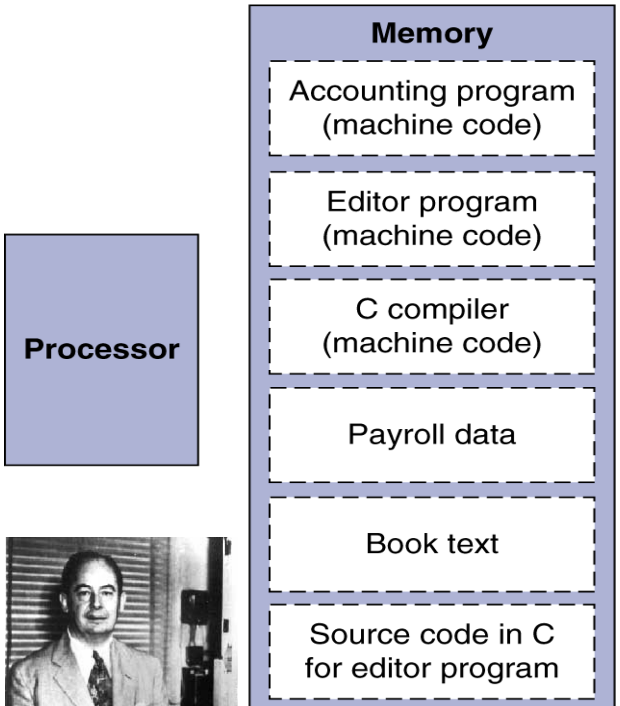
  `nor $t0, $t1, $zero`

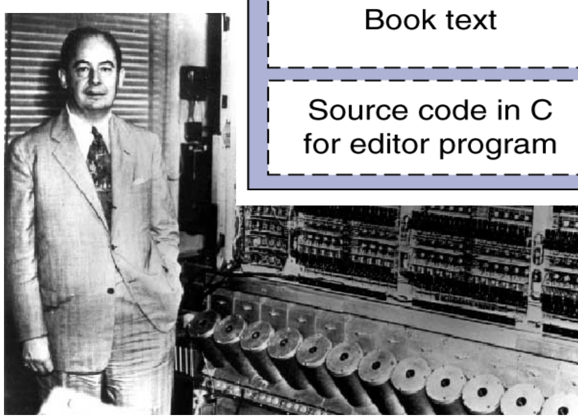
register 0: always zero

`$t1`	0000 0000 0000 0000 0011 1100 0000 0000
`$t0`	1111 1111 1111 1111 1100 0011 1111 1111

Stored Program Computers

The BIG Picture



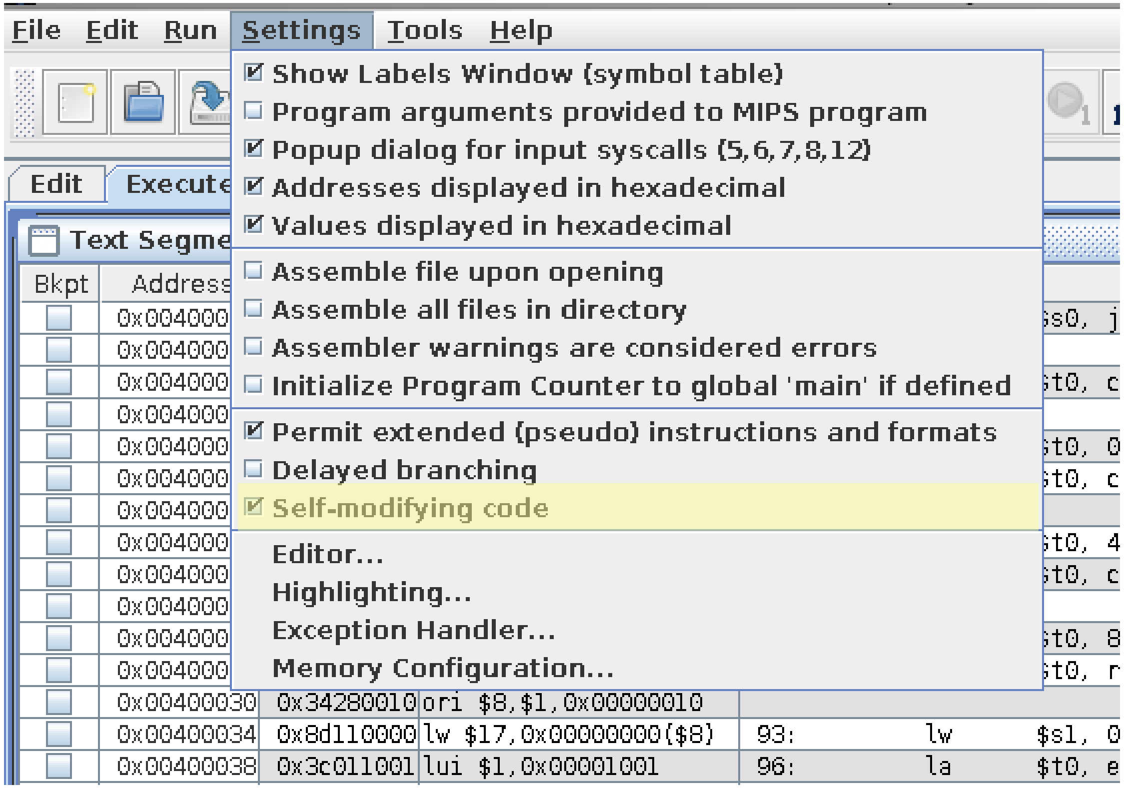


- **Instructions** represented in **binary**, just like **data**
- **Instructions and data** stored in **memory**
- **Programs** can **operate** on **programs**
  - *e.g.*, compilers, linkers, …
- **Binary compatibility** allows **compiled** programs to work on **different computers**
  - **standardized** ISAs

“Von Neumann architecture”

vs. “(modified) Harvard architecture”
(*e.g.,* Atmel AVR, Motorola 68HC12 – different HW technologies for instr/data)

Stored Program Computers



"Von Neumann architecture"

Program Counter (PC)

- Points to current – to be executed – instruction
- Incremented by 4 (all instructions 32 bit)

  or ... changed by branch/jump/...

Conditional Operations

"branch" (in the tree of possible execution paths)
to a labeled instruction (encoded as offset)
if a condition (comparing register values) is true
else, continue to **next sequential instruction**

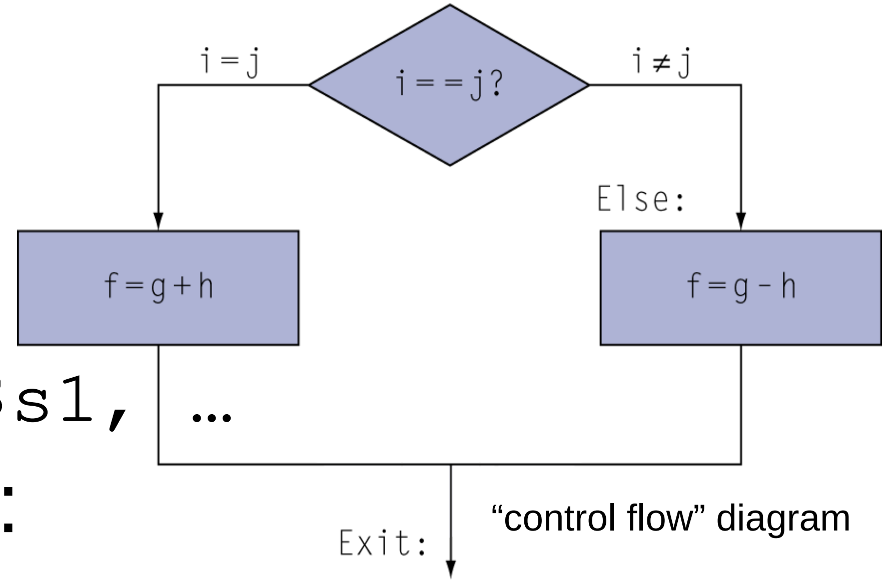
- `beq rs, rt, L1`
  - `if (rs == rt)`
    branch to instruction labeled `L1`
- `bne rs, rt, L1`
  - `if (rs != rt)`
    branch to instruction labeled `L1`
- `j L1`
  - **un**conditional jump to instruction labeled `L1`

Compiling `if` Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - `f, g, …` in `$s0, $s1, …`



"control flow" diagram

- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

code "block"

code "block"

Assembler calculates addresses

Compiling `while` Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - `i` in `$s3`, `k` in `$s5`, address of `save` in `$s6`

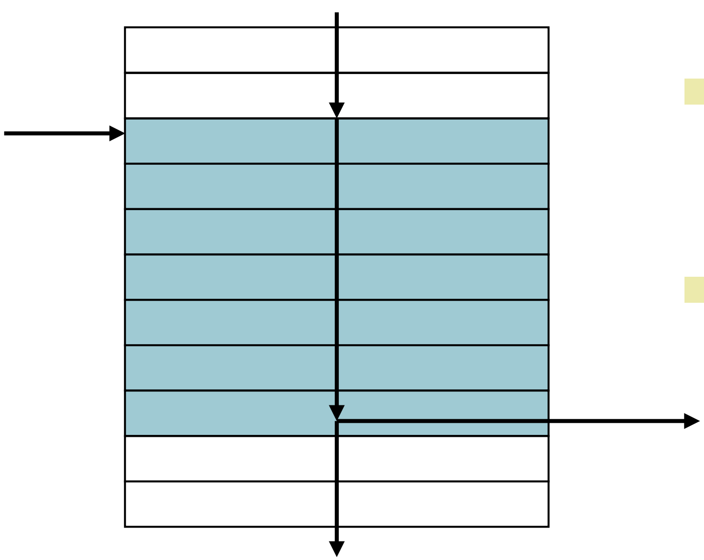
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1        code "block"
      j    Loop
Exit: …
```

Basic Block

is a **sequence of instructions** with

- No embedded **branches** (except at end)
- No **branch targets** (except at beginning)



- A **compiler** identifies basic blocks for **optimization**
- An advanced **processor** can **accelerate execution** of basic blocks (*e.g.*, re-order instructions)

More Conditional Operations

Set result to **1** if a condition is **true**

otherwise, set to **0** "canonical" (vs. C)

- `slt rd, rs, rt`
  - if (rs < rt) rd = **1**; else rd = **0**;
- `slti rt, rs, constant`
  - if (rs < constant) rt = **1**; else rt = **0**;
- Use in combination with beq, bne

2 clock cycles

```
slt $t0, $s1, $s2   # if ($s1 < $s2)
bne $t0, $zero, L   #   branch to L
==
blt $s1, $s2, L      # pseudo-instruction
```

Branch Instruction Design

- Why not `blt, bge,` etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a **slower clock**
  - All instructions penalized!
- `beq` and `bne` are the **common case**
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt, slti`
- Unsigned comparison: `slt`**`u`**`, slt`**`u`**`i`
- Example
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `slt  $t0, $s0, $s1  # signed`
    - `−1 < +1 →  $t0 = 1`
  - `sltu $t0, $s0, $s1  # unsigned`
    - `+4,294,967,295 > +1 →  $t0 = 0`

Switch statement

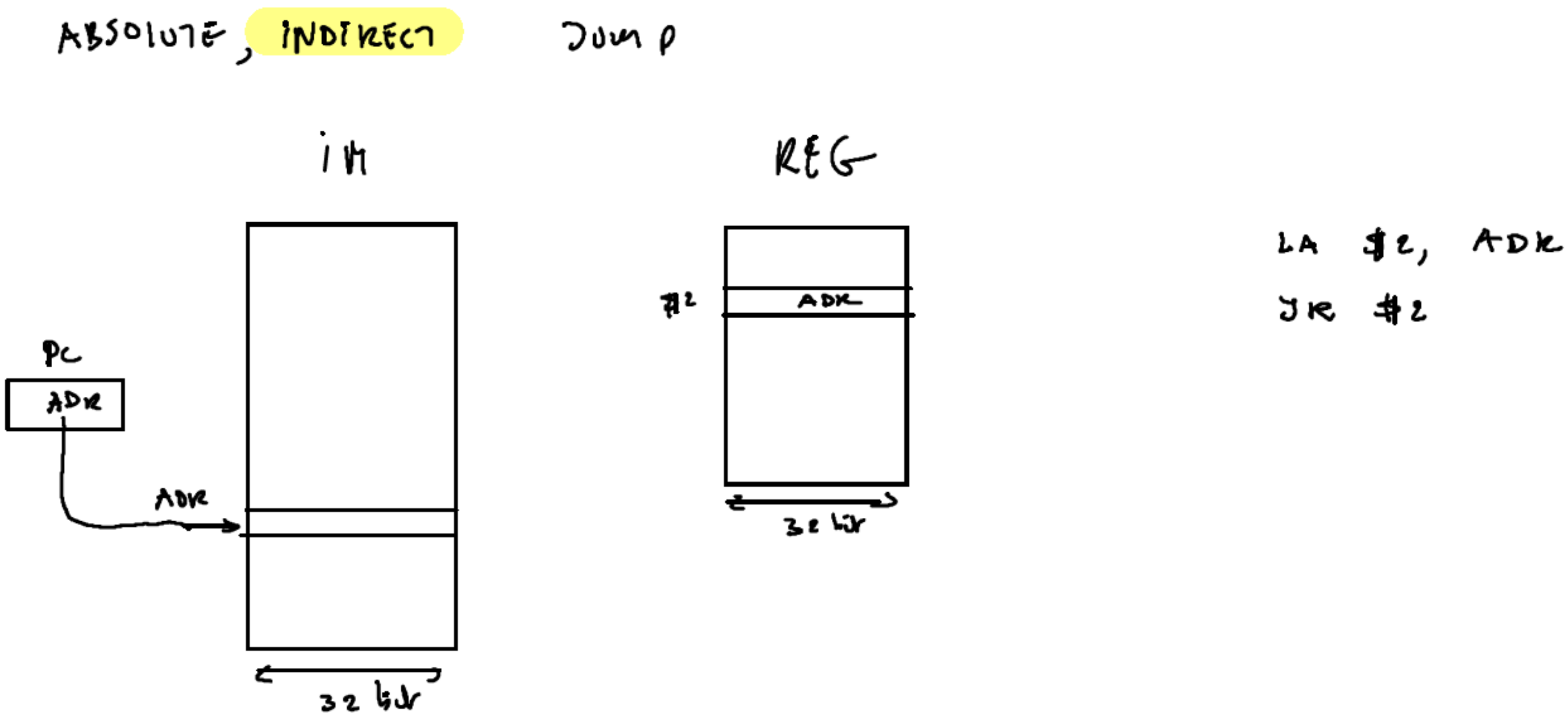
```
# General form

switch(expression_evaluating_to_integer_value) {

   case constant-expression_0 :
      Basic Block 0 statement(s)
      break; /* optional */


   case constant-expression_1 :
      Basic Block 1 statement(s)
      break; /* optional */


   /* … any number of case statements … */


   case constant-expression_LAST_CASE_NUM :
      Basic Block LAST_CASE_NUM statement(s)
      break; /* optional */


   default : /* Optional */
      Basic Block default statement(s);
}
```

Switch statement

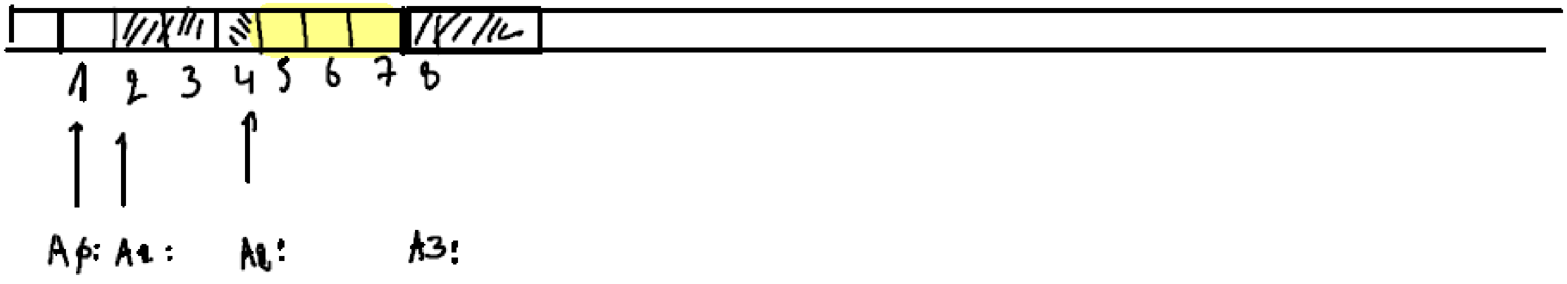
```
# Concrete Example:

 int result = 9;
 int expr_value = 1;


 switch(expr_value) {


   case 0  :
      result += 1;
      break;


   case 1  :
      result += 2;
      break;


   case 2  :
      result = 0;
      break;


   default :
      result = -1;
}
```

Jump Register (absolute, indirect)

Jump to full 32 bit address
(can not be encoded inside instruction)

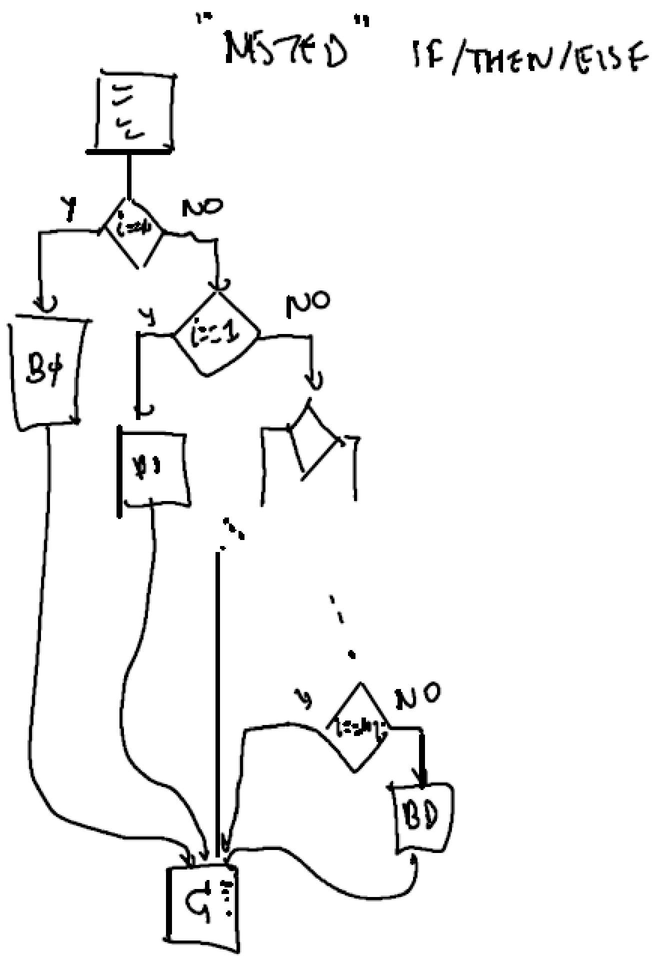


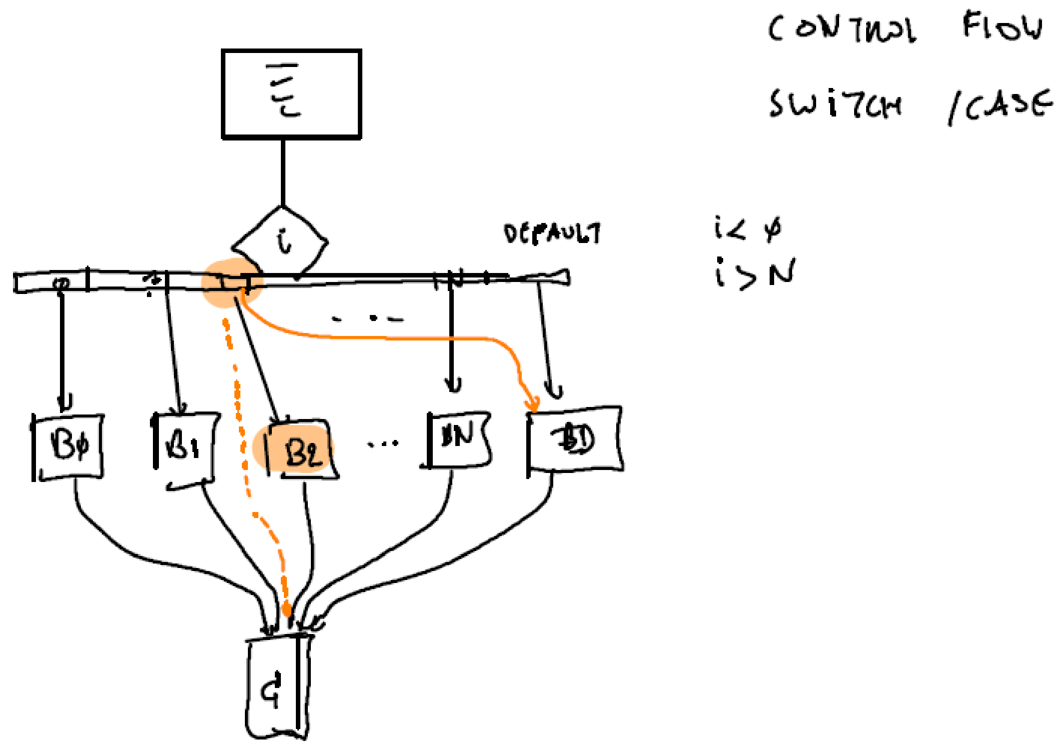
Data Memory "alignment"




```
.align  2           # .align n: align on 2ⁿ boundary
.space 36           # reserve space for 36 bytes
```

Switch statement (aka “computed jump”)





“control (flow) **indirection**”

Switch statement (aka “computed jump”)

jumpTbl:

@BB 0
@BB 1
…
@BB LAST_CASE_NUM

`BB` ==
Basic Block

`@` ==
address of

1. Check if `expr_value` in range
2. Compute where to find `@BB expr_value`
   `= @jumpTbl + 4*expr_value`
3. Load `@BB expr_value` into register `$R`
   by loading the 4 bytes found at address
   `@jumpTbl + 4*expr_value`
4. Indirect Jump `JR $R`

```
# macros for readability

        .eqv LAST_CASE_NUM 2 # cases are numbered 0 .. LAST_CASE_NUM
        .eqv LAST_CASE_NUM_PLUS1_TIMES4 12

        .data

        .align  2               # .align n: align on 2^n boundary
jumpTbl:
        .space  LAST_CASE_NUM_PLUS1_TIMES4 # allocate LAST_CASE_NUM+1 consecutive words,
                # with storage uninitialized,
                # to store LAST_CASE_NUM+1 pointers
                # (addresses of code for different cases)

#   int expr_value = 1
expr_value:   .word   1

#   int result = 9
result: .word   9

        .text

        #       fill jumpTbl with --instruction-- adresses of case0 ... case2

        la      $s0, jumpTbl   # $s0 contains the address of jumpTbl
        la      $t0, case0
        sw      $t0, 0($s0)   # dataMEM[ADDRESS(jumpTbl)+  0]  = ADDRESS(case0)
        la      $t0, case1
        sw      $t0, 4($s0)   # dataMem[ADDRESS(jumpTbl)+  4]  = ADDRESS(case1)
        la      $t0, case2
        sw      $t0, 8($s0)   # dataMEM[ADDRESS(jumpTbl)+  8]  = ADDRESS(case2)

# shorter alternative (let the assembler do the work)
#
# in
#        .data
# instead of
#        .align  2              # .align n: align on 2^n boundary
#        .space 12
#
#jumpTbl:
#        .word case0, case1, case2
```

```
# int result in $s1
        la      $t0, result
        lw      $s1, 0($t0)

# int expr_value in $s2
        la      $t0, expr_value
        lw      $s2, 0($t0)

#   check if expr_value out of range 0 .. LAST_CASE_NUM of the switch cases
        blt     $s2, $zero, default
        li      $t0, LAST_CASE_NUM
        bgt     $s2, $t0, default

#   note: what if the range is not "dense", i.e., some values of expr_value
#   in the range 0 .. LAST_CASE_NUM are also considered out of range?

#   use the jumpTbl to jump to the code for the appropriate case
        sll     $t1, $s2, 2   # multiply expr_value by 4
        add     $t1, $s0, $t1 # address at which address to jump to is found
        lw      $t2, 0($t1)   # address to jump to
        jr      $t2
```

```
#   case 0  :
#       result += 1;
#       break;
case0:
        addi    $s1, $s1, 1
        j       endSwitch

#   case 1  :
#       result += 2;
#       break;
case1:
        addi    $s1, $s1, 2
        j       endSwitch

#   case 2  :
#       result = 0;
#       break;
case2:
        li      $s1, 0
        j       endSwitch

#
#   default :
#       result = -1;
default:
        li      $s1, -1

endSwitch:

#   store result back in memory
        la      $t0, result
        sw      $s1, 0($t0)

#   cleanly exit to OS
        li      $v0, 10
        syscall
```

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	71: la $s0, jumpTbl # $s0 contains the address of jumpTbl
	0x00400004	0x34300000	ori $16,$1,0x00000000	
	0x00400008	0x3c010040	lui $1,0x00000040	72: la $t0, case0
	0x0040000c	0x34280068	ori $8,$1,0x00000068	
	0x00400010	0xae080000	sw $8,0x00000000($16)	73: sw $t0, 0($s0) # dataMEM[ADDRESS(jumpTbl)+ 0] = ADDRESS(case0)
	0x00400014	0x3c010040	lui $1,0x00000040	74: la $t0, case1
	0x00400018	0x34280070	ori $8,$1,0x00000070	
	0x0040001c	0xae080004	sw $8,0x00000004($16)	75: sw $t0, 4($s0) # dataMem[ADDRESS(jumpTbl)+ 4] = ADDRESS(case1)
	0x00400020	0x3c010040	lui $1,0x00000040	76: la $t0, case2
	0x00400024	0x34280078	ori $8,$1,0x00000078	
	0x00400028	0xae080008	sw $8,0x00000008($16)	77: sw $t0, 8($s0) # dataMEM[ADDRESS(jumpTbl)+ 8] = ADDRESS(case2)
	0x0040002c	0x3c011001	lui $1,0x00001001	92: la $t0, result
	0x00400030	0x34280010	ori $8,$1,0x00000010	
	0x00400034	0x8d110000	lw $17,0x00000000($8)	93: lw $s1, 0($t0)
	0x00400038	0x3c011001	lui $1,0x00001001	96: la $t0, expr_value
	0x0040003c	0x3428000c	ori $8,$1,0x0000000c	
	0x00400040	0x8d120000	lw $18,0x00000000($8)	97: lw $s2, 0($t0)
	0x00400044	0x0240082a	slt $1,$18,$0	100: blt $s2, $zero, default
	0x00400048	0x1420000d	bne $1,$0,0x0000000d	
	0x0040004c	0x24080002	addiu $8,$0,0x00000002	101: li $t0, 2
	0x00400050	0x0112082a	slt $1,$8,$18	102: bgt $s2, $t0, default
	0x00400054	0x1420000a	bne $1,$0,0x0000000a	
	0x00400058	0x00124880	sll $9,$18,0x00000002	108: sll $t1, $s2, 2 # multiply expr_value by 4
	0x0040005c	0x02094820	add $9,$16,$9	109: add $t1, $s0, $t1 # address at which address to jump to is found
	0x00400060	0x8d2a0000	lw $10,0x00000000($9)	110: lw $t2, 0($t1) # address to jump to
	0x00400064	0x01400008	jr $10	111: jr $t2
	0x00400068	0x22310001	addi $17,$17,0x0000...	117: addi $s1, $s1, 1
	0x0040006c	0x08100021	j 0x00400084	118: j endSwitch
	0x00400070	0x22310002	addi $17,$17,0x0000...	124: addi $s1, $s1, 2
	0x00400074	0x08100021	j 0x00400084	125: j endSwitch
	0x00400078	0x24110000	addiu $17,$0,0x0000...	131: li $s1, 0
	0x0040007c	0x08100021	j 0x00400084	132: j endSwitch
	0x00400080	0x2411ffff	addiu $17,$0,0xffff...	138: li $s1, -1
	0x00400084	0x3c011001	lui $1,0x00001001	143: la $t0, result
	0x00400088	0x34280010	ori $8,$1,0x00000010	
	0x0040008c	0xad110000	sw $17,0x00000000($8)	144: sw $s1, 0($t0)
	0x00400090	0x2402000a	addiu $2,$0,0x0000000a	147: li $v0, 10
	0x00400094	0x0000000c	syscall	148: syscall

Labels

Label ▲	Address
indirect_control.asm	
case0	0x00400068
case1	0x00400070
case2	0x00400078
default	0x00400080
endSwitch	0x00400084
expr_value	0x1001000c
jumpTbl	0x10010000
result	0x10010010

☑ Data ☑ Text

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000001	0x00000009	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

32-bit Constants

- **Most** constants are **small**
  and 16-bit immediate is sufficient
- For the occasional 32-bit constant:
  pseudo-instruction such as
- `la $s0, 32bitAddrLabel`
  (e.g., $4000000_{10} = 003D0900_{16} = 61 \times 2^{16} + 2304$)

`lui rt, constant # load` **`upper`** `immediate`

- Copies 16-bit constant to left 16 bits of `rt`
- Clears right 16 bits of `rt` to 0

`lui $s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0000 0000	0000 1001 0000 0000
4000000	0000 0000 0011 1101	0000 1001 0000 0000

Procedures/Functions

Group and encapsulate instructions

and refer to by procedure/function **name** (more general: “signature”)

- Hide *inside* (implementation details) → coginitive load reduced (*i.e.*, abstraction)
- Separate *use* from *implementation* can be varied *independently*
- Can be *used* multiple times, in different places
- *Parametrise* with arguments

Procedure/Function Calling

"Caller" vs. "Callee"

steps required:

1. Place **parameters** where callee can find them
2. **Transfer control** to callee
3. Acquire **storage** for callee (save)
4. Perform callee's **operations**
5. Place **result** where caller can find them
6. Release **storage** allocated for callee (restore)
7. **Return** to (just after) place of call (in caller)

Register Usage

- `$a0 – $a3`: argument values (`$4–$7`)
- `$v0, $v1`: result values (`$2, $3`)
- `$t0 – $t9`: temporaries (`$8-$15, $24-$25`)
  - may be overwritten by callee
- `$s0 – $s7`: saved (`$16–$23`)
  - must be **saved/restored by callee**
- `$gp`: global pointer - static data (`$28`)
- `$sp`: stack pointer (`$29`)
- `$fp`: frame pointer (`$30`)
- `$ra`: return address (`$31`)

`pc:` program counter (only indirect control)

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000020
$v1	3	0x00000000
$a0	4	0x00000011
$a1	5	0x00000011
$a2	6	0x00000001
$a3	7	0x00000001
$t0	8	0x00000022
$t1	9	0x00000002
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00000000
hi		0x00000000
lo		0x00000000

Procedure Call Instructions

- Procedure **call/invocation**: **j**ump **a**nd **l**ink

  `jal ProcedureLabel`

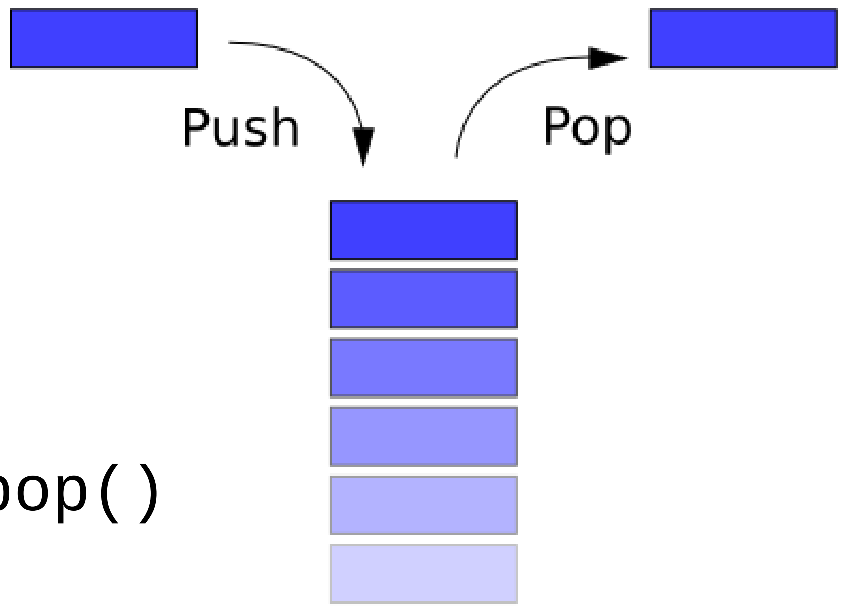
  - **Address of following (aka "next sequential") instruction** (*i.e.*, PC + 4) put in `$ra`
  - **Jumps** to **target** address
- Procedure **return**: jump register

  `jr $ra`

  - Copies `$ra` to **program counter**
  - Can also be used for **computed jumps**
    - *e.g.*, for switch/case statements (see earlier)

Stack (Abstract Data Type)

Data Structure + Operations



- **push**(value)
- value = **pop**()

- **INVARIANT**:
  push(value); res= pop()
  → res == value

Implementation data structures:

Stack memory + Stack Pointer (SP)

Stack: multiple realizations

```
# stack grows “downwards”, SP to top-of-stack

push: addi $sp, $sp, -4     # adjust stack for 1 word
      sw   $s0, 0($sp)      # save $s0
...
pop:  lw   $s0, 0($sp)      # restore saved $s0
      addi $sp, $sp, 4      # pop 1 item from stack



# stack grows “upwards”, SP to top-of-stack

push: addi $sp, $sp, 4      # adjust stack for 1 word
      sw   $s0, 0($sp)      # save $s0
...
pop:  lw   $s0, 0($sp)      # restore saved $s0
      addi $sp, $sp, -4     # pop 1 item from stack
```

Stack: multiple realizations

```
# stack growns “downwards”, SP to free space
#                                  below top-of-stack

push: sw   $s0, 0($sp)      # save $s0
      addi $sp, $sp, -4     # adjust stack for 1 word
...
pop:  addi $sp, $sp, 4      # pop 1 item from stack
      lw   $s0, 0($sp)      # restore saved $s0


# stack grows “upwards”, SP to free space
#                               above top-of-stack

push: sw   $s0, 0($sp)      # save $s0
      addi $sp, $sp, 4      # adjust stack for 1 word
...
pop:  addi $sp, $sp, -4     # pop 1 item from stack
      lw   $s0, 0($sp)      # restore saved $s0
```

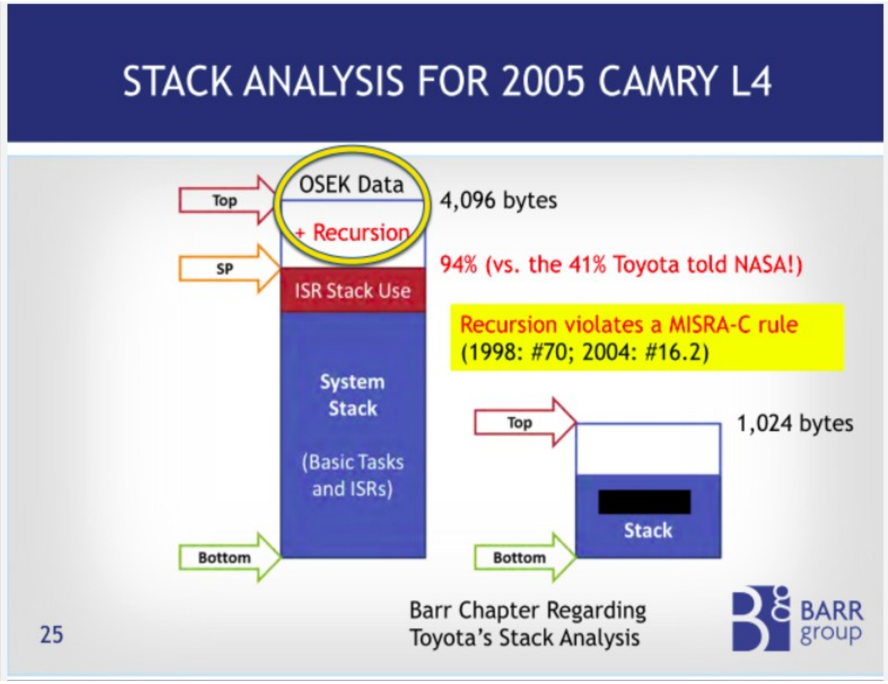
Stack: don’t mix them up!

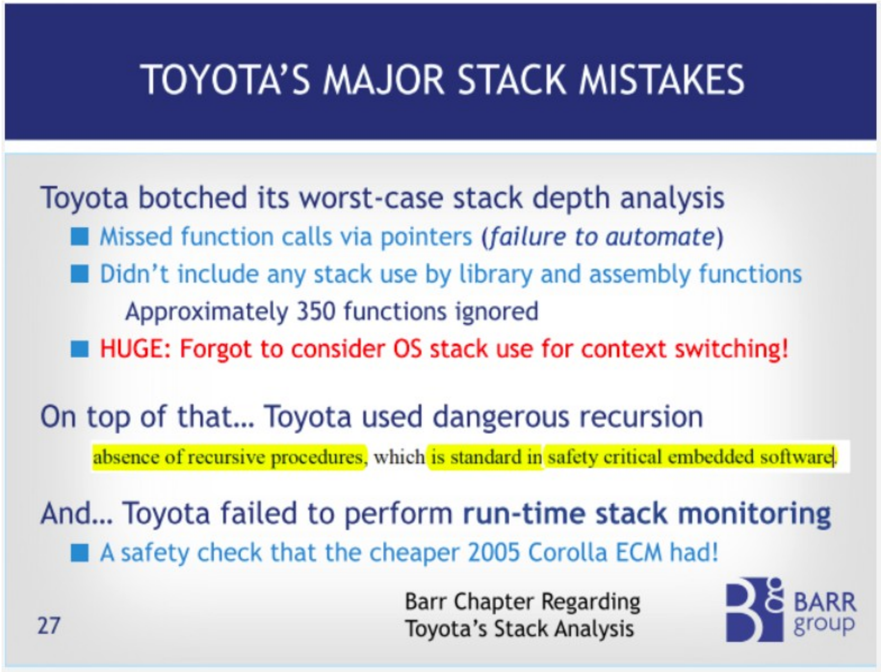
```
push: sw   $s0, 0($sp)      # save $s0
      addi $sp, $sp, -4     # adjust stack for 1 word
...
pop:  addi $sp, $sp, -4     # pop 1 item from stack
      lw   $s0, 0($sp)      # restore saved $s0
```

```
push: sw   $s0, 0($sp)      # save $s0
      addi $sp, $sp, -4     # adjust stack for 1 word
...
pop:  lw   $s0, 0($sp)      # restore saved $s0
      addi $sp, $sp, 4      # pop 1 item from stack
```

→ stack de-synchronization (and possible crashes/exploits)

Stack Overflow





www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf

OSEK = Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f; // local variable
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments `g`, …, `j` in `$a0`, …, `$a3`
  - Local variable `f` in `$s0` (hence, need to save `$s0` on stack)
  - Result in `$v0`

Leaf Procedure Example

- MIPS code:

```
leaf_example:
  addi $sp, $sp, -4
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)
  addi $sp, $sp, 4
  jr   $ra
```

Save `$s0` on stack: "push `$s0` onto stack"

Procedure body (three-address code)

Result (or `move $v0, $s0`)

Restore `$s0`: "pop `$s0` from stack"

Return to caller

How can the above code be optimized?

Assembly Program Template

```
# Comments giving
#  * name of program
#  * description of function
# template.asm (often used extension: .s)
# Bare-bones outline of
# MIPS assembly language program

      .data  # variable declarations follow this line
             # ...

      .text  # instructions (code) follow this line
      .globl main

main:        # entry point
             # indicates start of code to “loader”
             # (first instruction to execute, by convention)
             # ...

# End of program, leave a blank line afterwards
```

Calling a Leaf Procedure

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}


void main(void)
{ int gm, hm, im, jm, res;
  gm = 10; hm = 11; im = 3; jm = 4;
  res = leaf_example(gm, hm, im, jm);
  print(res);
}
```

Calling a Leaf Procedure

```
# main.asm
# Calling leaf_example
      .data
# varName: .word varValue  # need to load into $si
      .text
      .globl main
main: li   $a0,10  # main's variable gm
      li   $a1,11  # main's variable hm
      li   $a2,3   # main's variable im
      li   $a3,4   # main's variable jm
      #note: li == addiu pseudo-instruction
      jal  leaf_example
# result directly into $v0, eliminated res
      add  $a0, $v0, $zero # number to print
      li   $v0, 1          # print Integer system service
      syscall              # print result
      li   $v0, 10 # system call for clean exit
      syscall      # exit (back to operating system)
```

Application Binary Interface

Interacting with the Operating System: ABI

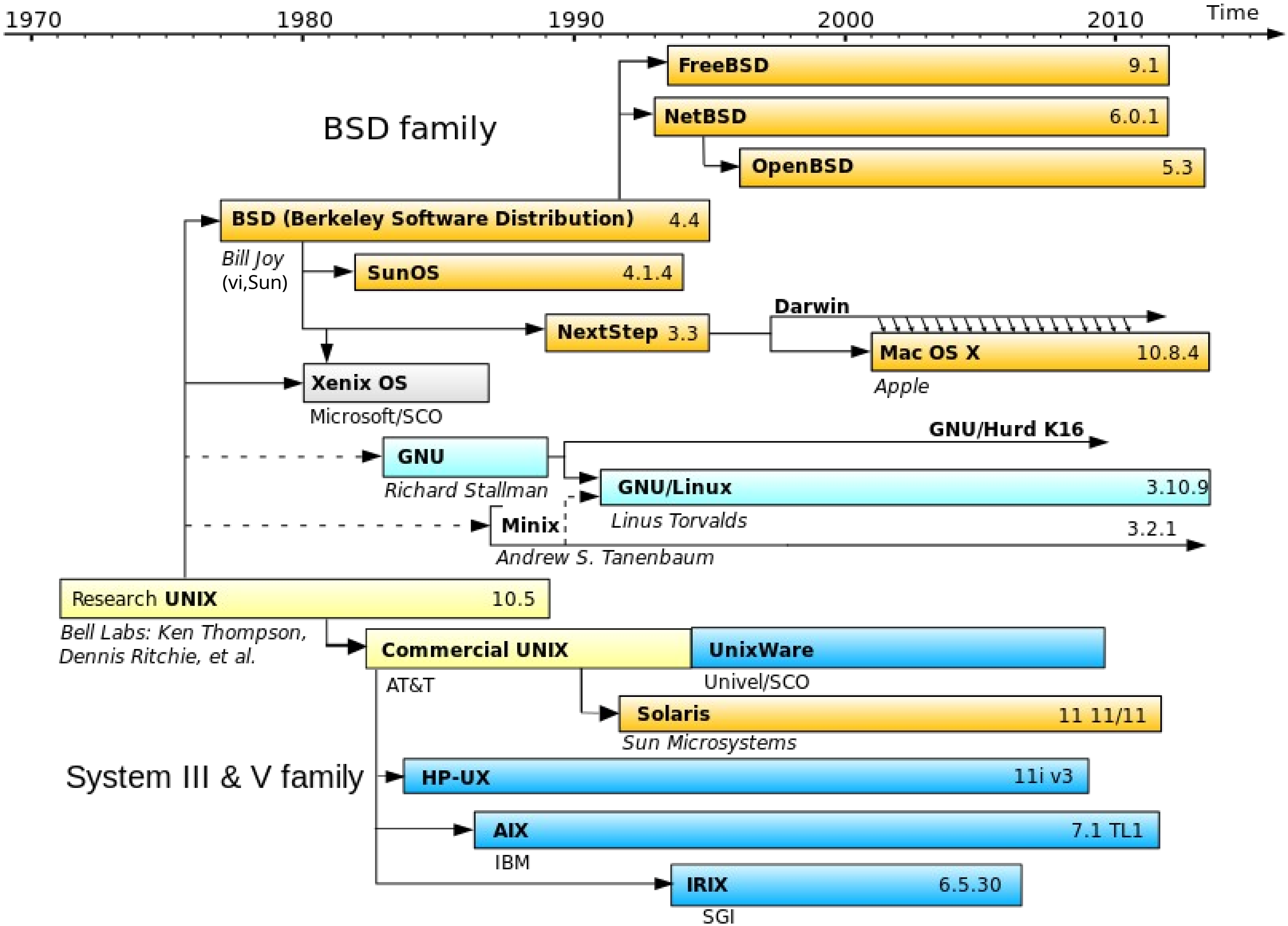
SYSTEM V
APPLICATION BINARY INTERFACE

MIPS® RISC Processor
Supplement
3rd Edition

http://math-atlas.sourceforge.net/devel/assembly/mipsabi32.pdf

Table of Contents

System V ?



1970
1980
1990
2000
2010
Time
FreeBSD 9.1
NetBSD 6.0.1
OpenBSD 5.3
BSD family
BSD (Berkeley Software Distribution) 4.4
Bill Joy
(vi,Sun)
SunOS 4.1.4
Darwin
NextStep 3.3
Mac OS X 10.8.4
Apple
Xenix OS
Microsoft/SCO
GNU/Hurd K16
GNU
Richard Stallman
GNU/Linux 3.10.9
Linus Torvalds
Minix
3.2.1
Andrew S. Tanenbaum
Research UNIX 10.5
Bell Labs: Ken Thompson,
Dennis Ritchie, et al.
Commercial UNIX
AT&T
UnixWare
Univel/SCO
Solaris 11 11/11
Sun Microsystems
System III & V family
HP-UX 11i v3
AIX 7.1 TL1
IBM
IRIX 6.5.30
SGI

Fibonacci

```
# Fibonacci.asm
# Compute first twelve Fibonacci numbers (iteratively, not recursively)
# and put in array, then print
#  F[0] = 1
#  F[1] = 1
#  F[n+2] = F[n] + F[n+1]
#
      .data
      .eqv    NUM_FIBS  12  # must be > 2
fibs: .word   0 : NUM_FIBS  # "array" of NUM_FIBS words to hold Fib values
                            # elements of array initialized to 0
size: .word  NUM_FIBS       # size of "array"

      .text
      .globl main
main:                       # linker/loader starts execution here
      la   $t0, fibs        # load address of array
      la   $t5, size        # load address of size variable
      lw   $t5, 0($t5)      # load array size into $t5
      li   $t2, 1           # 1 is first and second Fibonacci number
      sw   $t2, 0($t0)      # F[0] = 1
      sw   $t2, 4($t0)      # F[1] = F[0] = 1
      addi $t1, $t5, -2     # Counter for loop,
                            # will execute (size-2) times
```

Fibonacci

```
loop: lw   $t3, 0($t0)       # Get value from array F[n]
      lw   $t4, 4($t0)       # Get value from array F[n+1]
      add  $t2, $t3, $t4     # $t2 = F[n] + F[n+1]
      sw   $t2, 8($t0)       # store F[n+2] = F[n] + F[n+1] in array
      addi $t0, $t0, 4       # increment address of Fibonacci number source
      addi $t1, $t1, -1      # decrement loop counter
      bgtz $t1, loop         # repeat if not finished yet
# note: the above fails when only first two Fibonacci numbers needed
# → use "while cond" instead of "repeat … until cond" / "do … while"

# print results
      la   $a0, fibs         # first argument for print (array address)
      add  $a1, $zero, $t5   # second argument for print (array size)
      jal  print             # call print routine

# normal end of program, return control to operating system
      li   $v0, 10           # system call for exit
      syscall                # exit
```

Fibonacci

```
#########  (leaf) routine to print the numbers on one line.
      .data
space:.asciiz  " "              # space to insert between numbers
                                # asciiz = null-terminated, fixed length
head: .asciiz  "The Fibonacci numbers are:\n"

      .text
print:add  $t0, $zero, $a0   # starting address of array
      add  $t1, $zero, $a1   # initialize loop counter to array size
      la   $a0, head         # load address of print heading
      li   $v0, 4            # specify Print ASCII String service
      syscall                # print heading
prFNr:lw   $a0, 0($t0)       # load Fibonacci number for syscall
      li   $v0, 1            # specify Print Integer service
      syscall                # print Fibonacci number
      la   $a0, space        # load address of white space for syscall
      li   $v0, 4            # specify Print String service
      syscall                # output string
      addi $t0, $t0, 4       # increment address
      addi $t1, $t1, -1      # decrement loop counter
      bgtz $t1, prFNr        # repeat if not finished
      jr   $ra               # return
```

byReferenceVSbyValue.c

```
#
# "call by value" vs. "call by reference"
#

#include <stdio.h>


int f_by_value(int arg)
{
 int calc_res = 0;
 calc_res = arg+1;
 return calc_res;
}


void f_by_reference(int *arg_address)
{
 int calc_res = 0;
 calc_res = *arg_address+1;
 *arg_address = calc_res;
}
```

byReferenceVSbyValue.c

```
void main()
{
 int i = 10; int ret_val = 999;

 printf(" before f_by_value():     (i, ret_val) is  (%d, %d)\n", i, ret_val);

 /* call by value */
 ret_val = f_by_value(i);

 printf(" after  f_by_value():     (i, ret_val) is  (%d, %d)\n\n", i, ret_val);


 /* re-inialize values */
 i = 10;  ret_val = 999;


 printf(" before f_by_reference(): (i, ret_val) is  (%d, %d)\n", i, ret_val);

 /* call by reference */
 f_by_reference(&i);

 printf(" after  f_by_reference(): (i, ret_val) is  (%d, %d)\n", i, ret_val);
}
```

byReferenceVSbyValue.c

```
% gcc -o byReferenceVSbyValue byReferenceVSbyValue.c

% ./byReferenceVSbyValue

 before f_by_value():     (i, return_value) is  (10, 999)
 after  f_by_value():     (i, return_value) is  (10, 11)

 before f_by_reference(): (i, return_value) is  (10, 999)
 after  f_by_reference(): (i, return_value) is  (11, 999)
```

Current limitations on function arguments and results?

byReferenceVSbyValue.c

```
% gcc -o byReferenceVSbyValue byReferenceVSbyValue.c

% ./byReferenceVSbyValue

 before f_by_value():     (i, return_value) is  (10, 999)
 after  f_by_value():     (i, return_value) is  (10, 11)

 before f_by_reference(): (i, return_value) is  (10, 999)
 after  f_by_reference(): (i, return_value) is  (11, 999)
```

… call by reference is useful for passing large structures …

Procedure/Function Calling

"Caller" vs. "Callee"

steps required:

1. Place **parameters** where callee can find them
2. **Transfer control** to callee
3. Acquire **storage** for callee (save)
4. Perform callee's **operations**
5. Place **result** where caller can find them
6. Release **storage** allocated for callee (restore)
7. **Return** to place of call (in caller)

Call by Value/Reference

Caller: by value

```
          .data
i:        .word  10
ret_val:  .word 999

          .text
#         f_by_value(i):
#         value of i is passed
          la    $t0, i
          lw    $a0, 0($t0)
          jal   f_by_value

          la    $t1, ret_val
          sw    $v0, 0($t1)
```

by reference

```
          .data
i:        .word  10
ret_val:  .word 999

          .text
#         f_by_reference(&i):
#         address of i is passed
          la    $a0, i

          jal   f_by_reference
```

Callee: by value

```
          .text
f_by_value:
          move $t0, $a0
          addi $t0, 1

          move $v0, $t0 # result

          jr   $ra
```

by reference

```
          .text
f_by_reference:
          lw   $t0, 0($a0)
          addi $t0, 1

          sw   $t0, 0($a0) # result

          jr   $ra
```

compiler generated assembly

```
% gcc -S byReferenceVSbyValue.c
```

produces (intermediate) assembly code

`byReferenceVSbyValue.s`

```
          .file     "byReferenceVSbyValue.c"
          .text
          .globl    f_by_value
          .type     f_by_value, @function
f_by_value:
.LFB0:
          .cfi_startproc
          pushq     %rbp
          .cfi_def_cfa_offset 16
          .cfi_offset 6, -16
          movq      %rsp, %rbp
          .cfi_def_cfa_register 6
          movl      %edi, -20(%rbp)
          movl      $0, -4(%rbp)
          movl      -20(%rbp), %eax
          addl      $1, %eax
          movl      %eax, -4(%rbp)
          movl      -4(%rbp), %eax
          popq      %rbp
          .cfi_def_cfa 7, 8
          ret
```

compiler generated assembly

```
% gcc -S byReferenceVSbyValue.c
```

produces (intermediate) assembly code

`byReferenceVSbyValue.s`

```
          .file     "byReferenceVSbyValue.c"
          .text
          .globl    f_by_value
          .type     f_by_value, @function
f_by_value:
.LFB0:
          .cfi_startproc
          pushq     %rbp
          .cfi_def_cfa_offset 16
          .cfi_offset 6, -16
          movq      %rsp, %rbp
          .cfi_def_cfa_register 6
          movl      %edi, -20(%rbp)
          movl      $0, -4(%rbp)
          movl      -20(%rbp), %eax
          addl      $1, %eax
          movl      %eax, -4(%rbp)
          movl      -4(%rbp), %eax
          popq      %rbp
          .cfi_def_cfa 7, 8
          ret
```

... for **Intel** instruction set, **not for MIPS**

need “cross compilation” (†)

```
% gcc -S byReferenceVSbyValue.c
```

Use `http://crosstool-ng.org`

Produces `byReferenceVSbyValue.s`

```
        .file   1 "byReferenceVSbyValue.c"
        .section .mdebug.abi32
        .previous
        .abicalls
```

(†) or emulate MIPS using `QEMU`

byRefenceVSbyValue.s (f_by_value())

```
            .text
            .align  2
            .globl  f_by_value
            .ent    f_by_value
            .type   f_by_value, @function
f_by_value:
            .frame    $fp,24,$31                    # vars= 8, regs= 1/0, args= 0, gp= 8
            .mask     0x40000000,-8
            .fmask    0x00000000,0
            .set      noreorder
            .cpload   $25
            .set      reorder
            addiu     $sp,$sp,-24
            sw        $fp,16($sp)
            move      $fp,$sp
            sw        $4,24($fp)
            sw        $0,8($fp)
            lw        $2,24($fp)
            addiu     $2,$2,1
            sw        $2,8($fp)
            lw        $2,8($fp)
            move      $sp,$fp
            lw        $fp,16($sp)
            addiu     $sp,$sp,24
            j         $31
            .end      f_by_value
            .align    2
            .globl    f_by_reference
            .ent      f_by_reference
            .type     f_by_reference, @function
```

byRefenceVSbyValue.s (f_by_reference())

```
f_by_reference:
        .frame    $fp,24,$31                # vars= 8, regs= 1/0, args= 0, gp= 8
        .mask     0x40000000,-8
        .fmask    0x00000000,0
        .set      noreorder
        .cpload   $25
        .set      reorder
        addiu     $sp,$sp,-24
        sw        $fp,16($sp)
        move      $fp,$sp
        sw        $4,24($fp)
        lw        $2,24($fp)
        lw        $2,0($2)
        addiu     $2,$2,1
        sw        $2,8($fp)
        lw        $3,24($fp)
        lw        $2,8($fp)
        sw        $2,0($3)
        move      $sp,$fp
        lw        $fp,16($sp)
        addiu     $sp,$sp,24
        j         $31
        .end      f_by_reference
```

byRefenceVSbyValue.s (data)

```
        .rdata
        .align   2
$LC0:
        .ascii   " before f_by_value():     (i, return_value) is  (%d, %d)"
        .ascii   "\n\000"
        .align   2
$LC1:
        .ascii   " after  f_by_value():     (i, return_value) is  (%d, %d)"
        .ascii   "\n\n\000"
        .align   2
$LC2:
        .ascii   " before f_by_reference(): (i, return_value) is  (%d, %d)"
        .ascii   "\n\000"
        .align   2
$LC3:
        .ascii   " after  f_by_reference(): (i, return_value) is  (%d, %d)"
        .ascii   "\n\000"
```

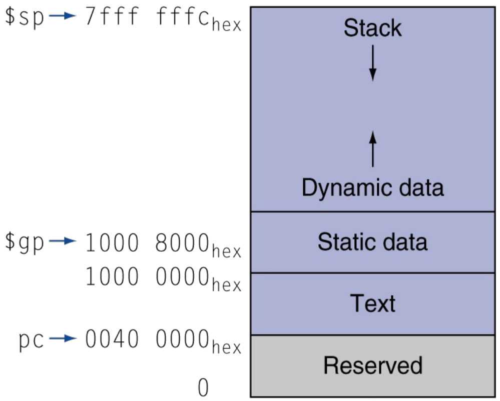
byRefenceVSbyValue.s (main())

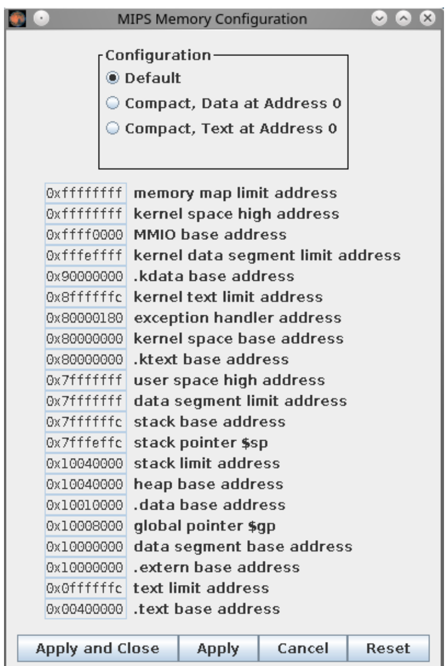
```
        .text
        .align   2
        .globl   main
        .ent     main
        .type    main, @function
main:
        .frame   $fp,40,$31             # vars= 8, regs= 2/0, args= 16, gp= 8
        .mask    0xc0000000,-4
        .fmask   0x00000000,0
        .set     noreorder
        .cpload  $25
        .set     reorder
        addiu    $sp,$sp,-40
        sw       $31,36($sp)
        sw       $fp,32($sp)
        move     $fp,$sp
        .cprestore        16
        li       $2,10                  # 0xa
        sw       $2,24($fp)
        li       $2,999                 # 0x3e7
        sw       $2,28($fp)
        la       $4,$LC0
        lw       $5,24($fp)
        lw       $6,28($fp)
        jal      printf
```

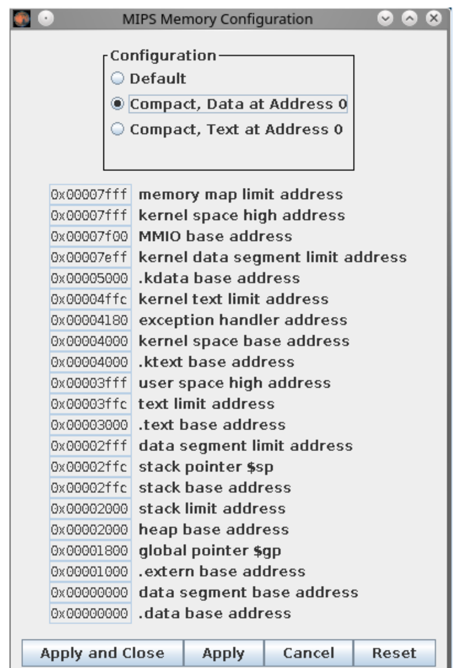
byRefenceVSbyValue.s (main() ctd.)

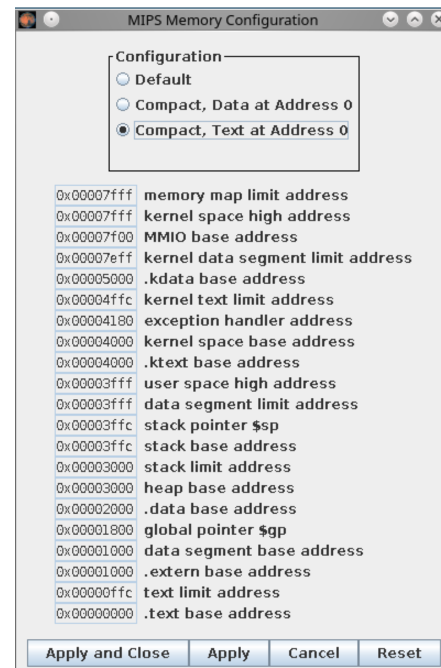
```
lw      $4,24($fp)
jal     f_by_value
sw      $2,28($fp)
la      $4,$LC1
lw      $5,24($fp)
lw      $6,28($fp)
jal     printf
li      $2,10                   # 0xa
sw      $2,24($fp)
li      $2,999                  # 0x3e7
sw      $2,28($fp)
la      $4,$LC2
lw      $5,24($fp)
lw      $6,28($fp)
jal     printf
addiu   $4,$fp,24
jal     f_by_reference
la      $4,$LC3
lw      $5,24($fp)
lw      $6,28($fp)
jal     printf
move    $sp,$fp
lw      $31,36($sp)
lw      $fp,32($sp)
addiu   $sp,$sp,40
j       $31
.end    main
.ident  "GCC: (GNU) 3.4.5"
```

Memory Layout



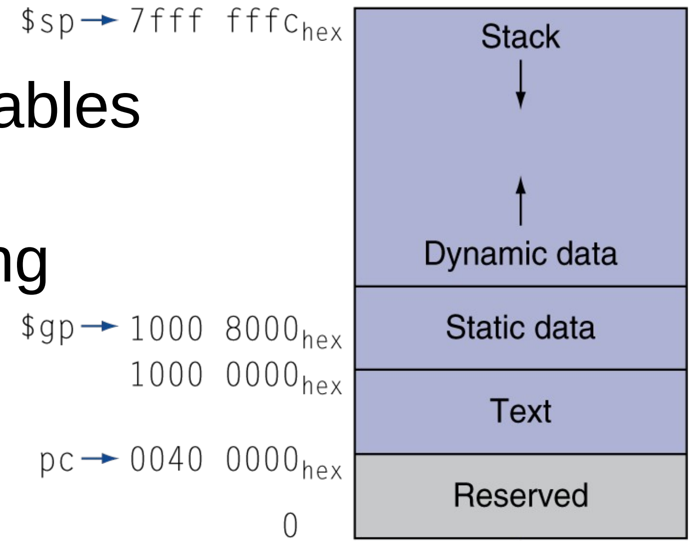




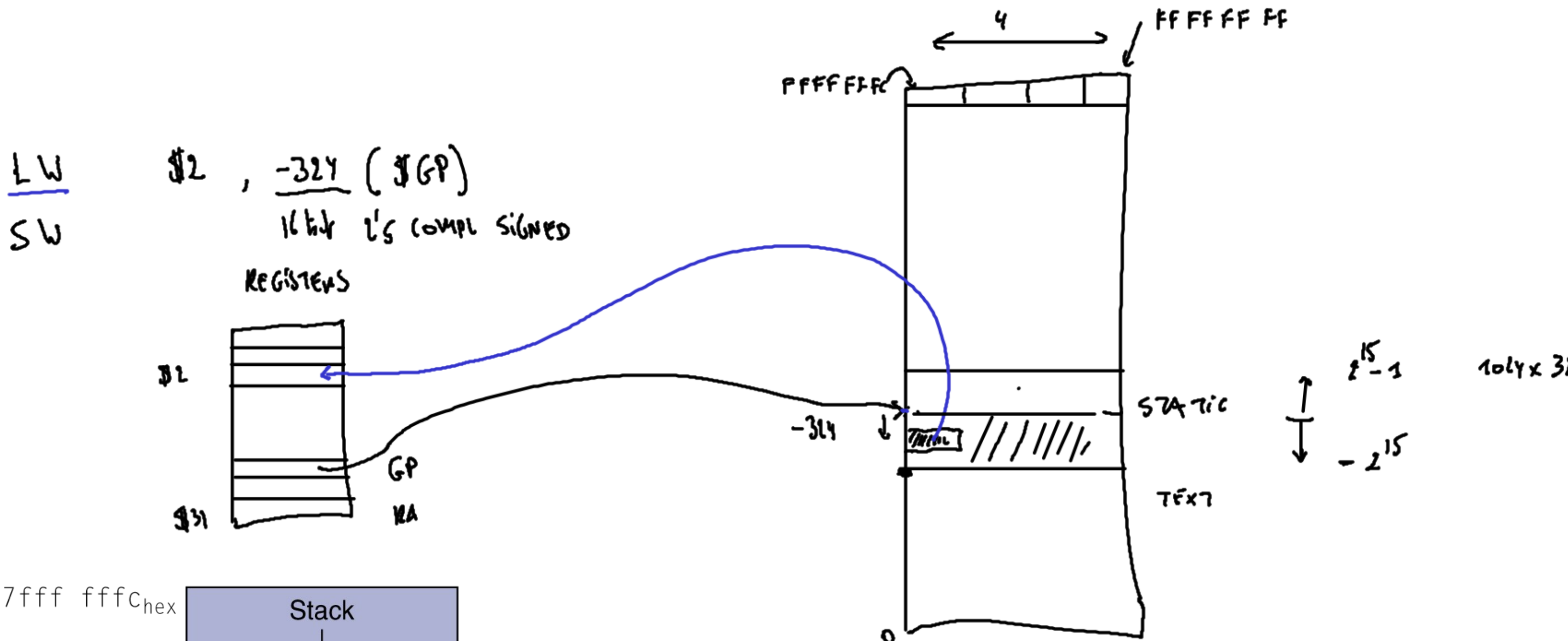


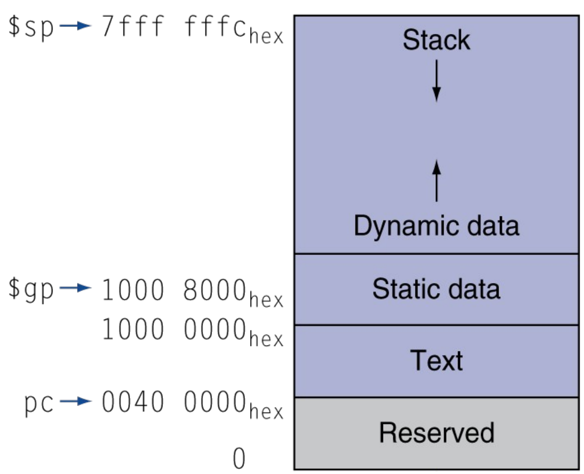
Memory Layout

- **Text**: program **code**
- **Static** data: **global** variables
  - *e.g.*, `static` (in a function) variables in `C`, constant arrays and strings
  - `$gp` initialized to address allowing ±`offsets` into this segment



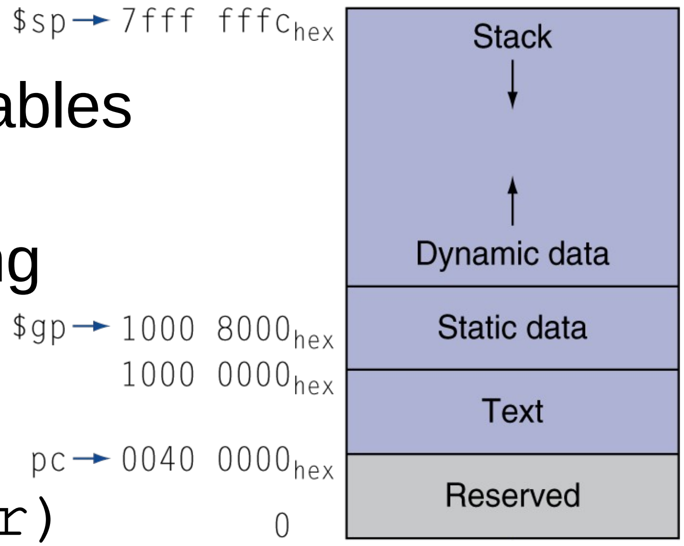
Memory Layout



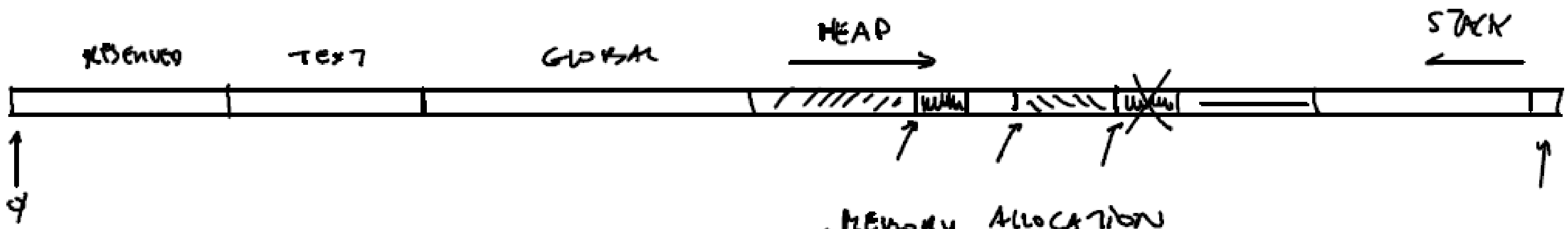


Memory Layout

- **Text**: program **code**
- **Static** data: **global** variables
  - *e.g.*, `static` (in a function) variables in `C`, constant arrays and strings
  - `$gp` initialized to address allowing ±`offsets` into this segment
- **Dynamic** data: **heap**
  - *e.g.*, `malloc(size)/free(ptr)` in `C`, `new(DataType)` in `Java`
- **Stack**: **automatic** storage
  - local variables in functions/procedures and much more, cfr. Stack Frame



Heap vs. Stack (grow towards each other)



MEMORY ALLOCATION
GARBAGE COLLECTION

malloc()
free()

"MEMORY LEAK"
MEMORY OVERFLOW

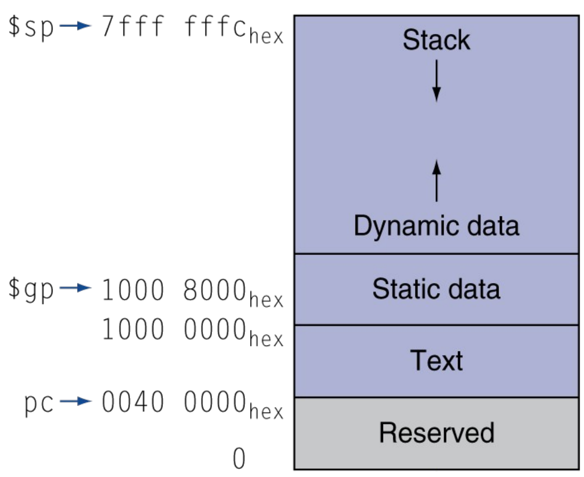
a = 10

a = "ABC"

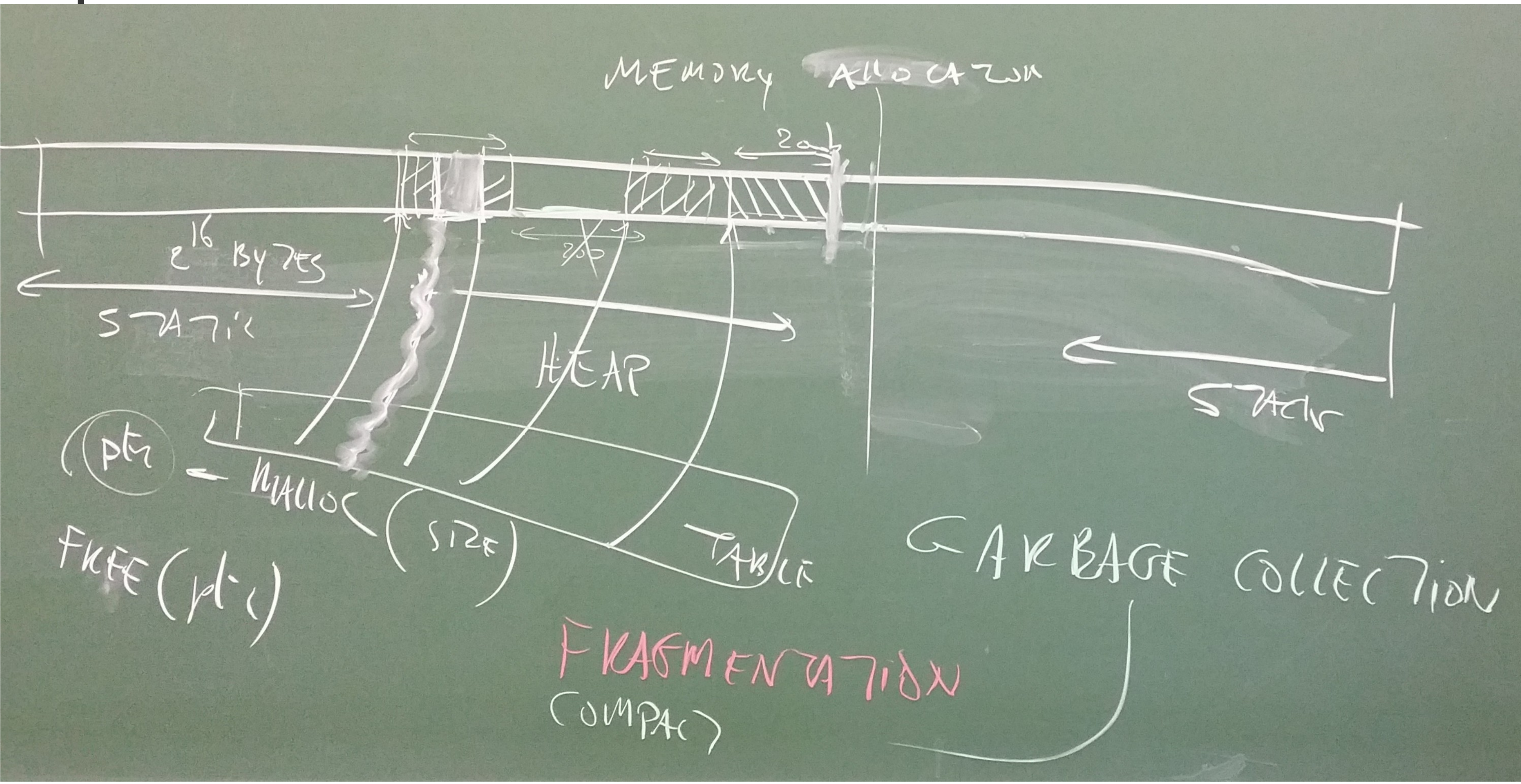
STACK OVERFLOW

"STACK OF SYNCHRONIZATION"

PUSH / POP



Heap needs Garbage Collection

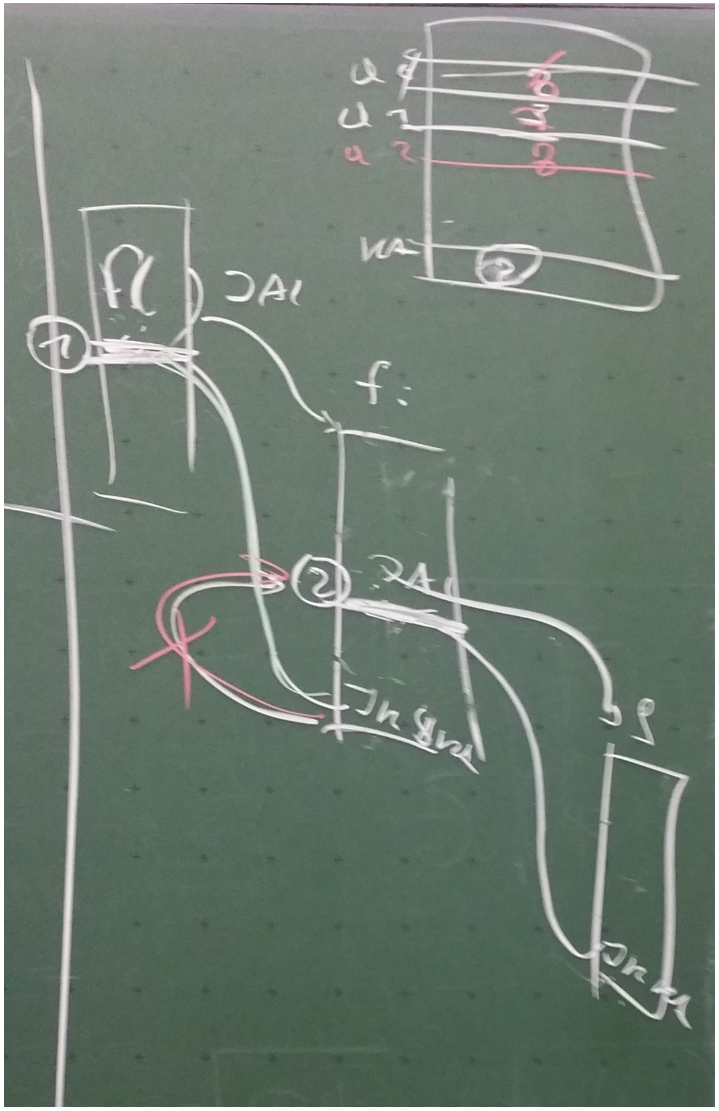


```c
/*
 * overflow.c
 *
 * demonstrates stack and heap overflow (and command-line arguments)
 */

#include <stdio.h>
#include <stdlib.h>

void stackOverflow(void);

void heapOverflow(void);

int main(int argc, char **argv) {
  if (argc > 1) {

    /* stack overflow option */
    if (argv[1][0] == 's') {
      printf("Entering infinite recursion ...\n");
      stackOverflow();
    }

    /* heap out of memory option                                  */
    /* uses virtual memory, so will start swapping and thrash
     * to avoid, set limit on amount of memory process can use:
     *  systemd-run --scope -p MemoryMax=500M --user ./overflow h
     */
    if (argv[1][0] == 'h') {
      printf("Using up the whole heap ...\n");
      heapOverflow();
    }
  }
}

void stackOverflow() { /* oops, forgot termination condition */
  stackOverflow();
}

void heapOverflow(void) {
  char *p;
  while (1) {p = malloc(1000000*sizeof(char));}
}
```

Non-Leaf Procedures

- Procedures that **call other** procedures
- For "nested" calls (may be multiple), **callee** needs to **save on the stack**, whatever could be **overwritten**:
  - its **return address**
  - any **arguments** and **temporaries** needed after the call (note how arguments and local variables are very similar)
- **Restore from the stack** after the call

Non-Leaf Procedures



Register Conventions

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument n in `$a0`
  - Result in `$v0`

Non-Leaf Procedure Example

MIPS code:

```
fact:
     addi $sp, $sp, -8      # adjust stack for 2 items
     sw   $ra, 4($sp)       # push (save) return address
     sw   $a0, 0($sp)       # push (save) argument
     slti $t0, $a0, 1       # test for n < 1
     beq  $t0, $zero, fM1
     addi $v0, $zero, 1     # if so, result is 1
     addi $sp, $sp, 8       #   pop 2 items from stack
     jr   $ra               #   and return
fM1:addi $a0, $a0, -1       # else decrement n
     jal  fact              # recursive call, overwrites
     lw   $a0, 0($sp)       # restore original n
     lw   $ra, 4($sp)       #   and return address
     addi $sp, $sp, 8       # pop 2 items from stack
     mul  $v0, $a0, $v0     # multiply to get result
     jr   $ra               # and return
```

How can the above code be optimized?

Preserved information

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

Assuming the stack grows “downwards”

Stack Frame

SYSTEM V
APPLICATION BINARY INTERFACE

MIPS® RISC Processor Supplement
3rd Edition

Table of Contents

remember: byRefenceVSbyValue.s (f_by_value())

```
        .text
        .align  2
        .globl  f_by_value
        .ent    f_by_value
        .type   f_by_value, @function
f_by_value:
        .frame    $fp,24,$31               # vars= 8, regs= 1/0, args= 0, gp= 8
        .mask     0x40000000,-8
        .fmask    0x00000000,0
        .set      noreorder
        .cpload   $25
        .set      reorder
        addiu     $sp,$sp,-24
        sw        $fp,16($sp)
        move      $fp,$sp
        sw        $4,24($fp)
        sw        $0,8($fp)
        lw        $2,24($fp)
        addiu     $2,$2,1
        sw        $2,8($fp)
        lw        $2,8($fp)
        move      $sp,$fp
        lw        $fp,16($sp)
        addiu     $sp,$sp,24
        j         $31
        .end      f_by_value
        .align    2
        .globl    f_by_reference
        .ent      f_by_reference
        .type     f_by_reference, @function
```

Stack Frame

Procedure/Function **(call) frame** (aka **activation record**)

- Used by (some) compilers to manage stack storage
- In addition to Stack Pointer register `$sp`,
  use **Frame Pointer** register **`$fp`**
  to keep track of ***all*** *pertinent information on the* ***stack***
  pertaining to a procedure/function **invocation** (aka **activation**).

Caller side:

- Caller pushes arguments on the stack
  (or passes them via `$a0 – $a3` if not more than 4 arguments)
- Caller reserves space on the stack for return values
  (or they are returned via `$v0 – $v1`)
- Caller passes “static link” (address on the stack of the nearest occurrence of the next lexically enclosing procedure/function) via `$v0`

The call:

- `jal label` jumps to label and puts the return address in `$ra`

Stack Frame

Callee side:

- Callee puts the old `$fp` at `0($sp)`.
  This is the “dynamic link” (chain).
  (note that here we chose to let `$sp` point beyond the stack).
- Callee sets `$fp` to `$sp` (beginning of the callee's frame)
- Callee reserves space on the stack for the entire frame (decrease `$sp` by frame size as the stack grows “downwards”)
- Callee saves the return address `$ra` at `-4($fp)` (in case the callee calls another function and overwrites `$ra`).
- Callee saves the “static link” from `$v0` at `-8($fp)`.
- Callee reserves the following words in the frame for local variables. To be initialized if appropriate.
- Callee uses words in the frame after the local variable to save temporary variables etc.
- If the callee in turns calls another function, see Caller side

Stack Frame

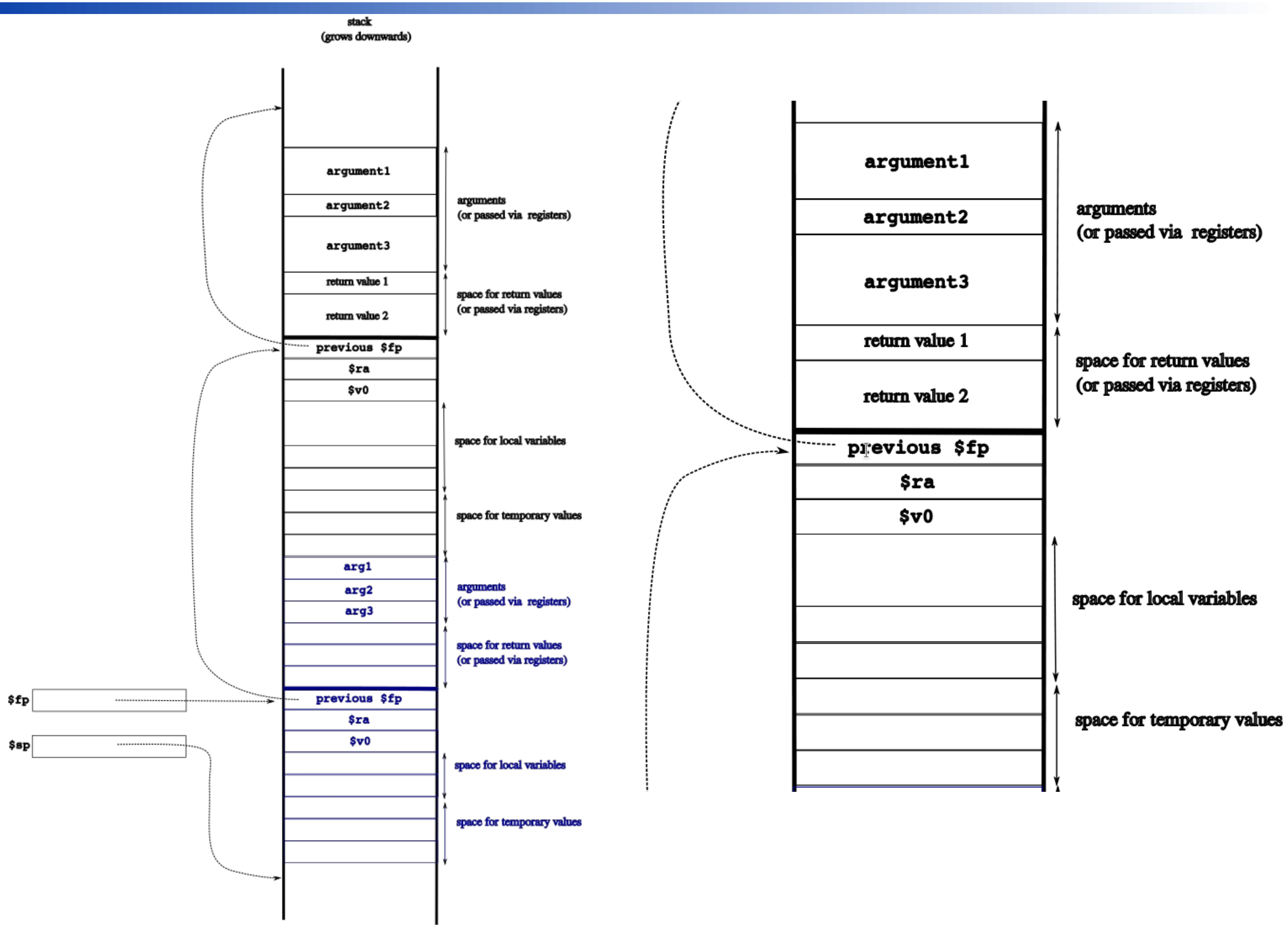
Callee side, returning from the call:

- Callee puts function result(s) in appropriate place(s) above the frame pointer (or returns via `$v0 – $v1`).
- Callee restores `$ra` from `–4($fp)`
- Callee sets `$sp` to `$fp` (*i.e.*, pops the entire frame)
- Callee sets `$fp` to `0($fp)` (the frame pointer of the caller)
- Callee returns with `jr $ra`

Caller side, returning from the call:

- Caller pops the function result(s) (if any) from the stack (unless passed via `$v0 – $v1`).
- Caller pops the function arguments (if any) from the stack.

Frame Structure (with dynamic links)



Factorial with Frames

```
/* factorial function

   fact.c

   based on an example by Leonidas Fegaras
   for CSE 5317 at UTA in Spring 1998

 */

  int fact ( int n ) {
   if (n<=1)
     return 1;
   else
     return n*fact(n-1);
  };
```

Factorial with Frames

```
      .text
fact: # Allocate a frame for the dynamic link, return address,
      # and static link (total: 3 words * 4 bytes/word = 12 bytes)
      # not more space needed as no local variables, temporaries, ...
      sw        $fp, ($sp)       # push old frame pointer (dynamic link);
                                 #($sp) means 0($sp)
      move      $fp, $sp # frame pointer now points to the top of the stack
      subu      $sp, $sp, 12     # allocate 3 words (12 bytes) on the stack

      # Save return address in the frame
      sw        $ra, -4($fp)

      # Save static link in the frame
      # $v0 is provided by caller
      # Note that $v0 is also used to pass a function return value
      sw        $v0, -8($fp)
```

Factorial with Frames

```
# if n>0 goto recurs: (the argument n is located at 4($fp))
#
# Note 1: if the result of fact() were not returned via $v0, but rather via
# the stack, 4($fp) would be reserved for this return value
# and the argument n of fact() would be found at 8($fp)
#
# Note 2: argument n of fact() could be passed via the argument register $a0.
# In that case, no space should be reserved for it on the stack
# and the following instruction
#    lw      $a0, 4($fp)
# would not be needed.
# There are two reasons for passing argument(s) via the stack:
#   1. if there are more than 4 arguments, there are not enough
#      argument registers $a0 - $a3 to pass all arguments directly;
#   2. if inside fact(), a function is called (in this case, fact() calls
#      itself recursively), the content of the argument register(s) need to be
#      saved (on the stack) as they may be overwritten.
#      Arguments passed via the stack are already saved.

      lw          $a0, 4($fp)
      bgt         $a0, 1, recurs

      # otherwise return 1
      li          $v0, 1
      b           return   # unconditional relative branch
                           # (beq $zero, $zero, return)
```

Factorial with Frames

```
recurs:
          lw      $a0, 4($fp)      # get n from the stack
          subu    $a0, $a0, 1      # calculate n-1
          # push n-1 on the stack as the argument to fact()
          sw      $a0, ($sp)
          subu    $sp, $sp, 4

          # load static link (was passed in $v0 by the caller
          # and stored in frame at -8($fp))
          lw      $v0, -8($fp)
          # call fact()
          jal     fact

          # multiply the result of fact(n-1), found in the result
          # register $v0, by n, found on the stack at 4($fp),
          # and place the result in the result register $v0
          lw      $a0, 4($fp)
          mul     $v0, $v0, $a0

return:   # return from fact (restore registers and pop the frame)
          lw      $ra, -4($fp)      # get return address from frame
          move    $sp, $fp          # get old frame pointer from current frame
          lw      $fp, ($sp)        # restore old frame pointer
          jr      $ra
```

Factorial with Frames

```
/* The main program calling the factorial function fact()

  main.c

 */

#include <stdio.h>

void main (void) {
   int i, res;
   while (1) {
      printf(" Number? ");
      scanf("%d", &i);
      if (i<=0) break;
      res = fact(i);
      printf("The factorial is: %d\n", res);
    };
 };
```

Factorial with Frames

```
        .text

        # main must be global (visible to functions in other files)
        .globl  main
main:   # allocate a frame for the dynamic link, return address, static link,
        # and for the integer local variables i and res (total: 5*4 = 20 bytes)
        sw      $fp, ($sp)       # push old frame pointer (dynamic link)
        move    $fp, $sp         # frame pointer now points
                                 # to the top of the stack
        subu    $sp, $sp, 20     # allocate 20 bytes on the stack

        # save return address in frame
        sw      $ra, -4($fp)

        # save static link in frame
        sw      $v0, -8($fp)
```

Factorial with Frames

```
loop:    # print prompt
         # note the interleaving of .data and .text

          .data
prompt:   .asciiz  "Number? "     # \0 terminated ASCII string

          .text
          li       $v0, 4         # print string service
          la       $a0, prompt    # address of the string to be printed
          syscall

          # read i
          li       $v0, 5         # read integer service
          syscall
          sw       $v0, -12($fp)  # result of read is local variable i,
                                  # located in frame, at  -12($fp)

          # if i<=0 goto exit:
          blez     $v0, exit
```

Factorial with Frames

```
# push i on the stack, as argument for the fact() call
lw      $a0, -12($fp)  # where in the frame local variable i is found
sw      $a0, ($sp)     # push i
subu    $sp, $sp, 4    # update stack pointer

# the static link of fact() is the frame pointer of main(),
# the nearest enclosing “lexical scope”
move    $v0, $fp

# call fact
jal     fact

# store the result of fact(i) (returned via result register $v0)
# in res (local variable res is located at -16($fp))
sw      $v0, -16($fp)
```

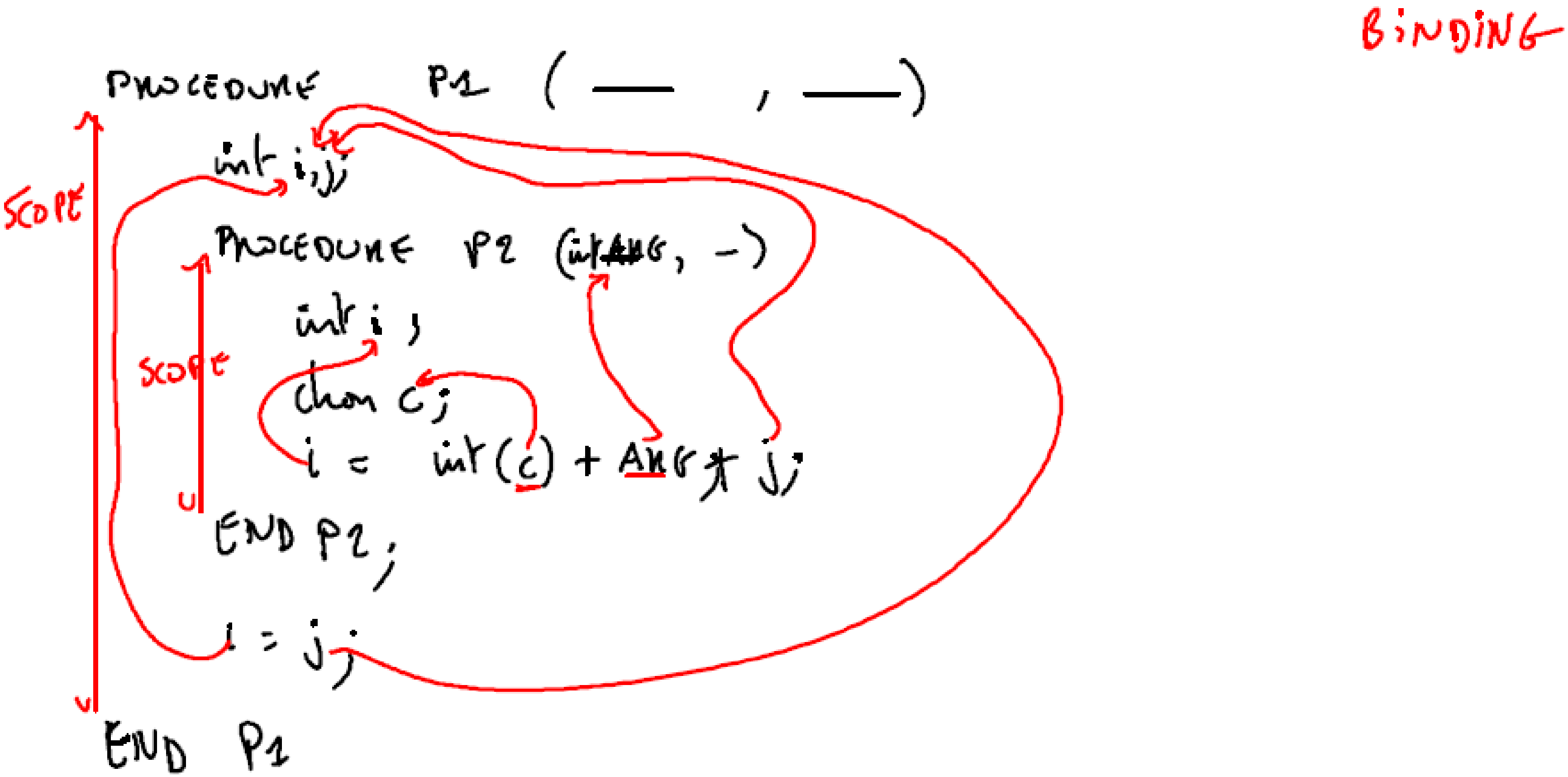
Factorial with Frames

```
        # print answer
        .data
answer: .asciiz "The factorial is: "
        .text
        li      $v0, 4          # print string service
        la      $a0, answer     # address of the string to be printed
        syscall

        # print res
        li      $v0, 1          # print integer service
        lw      $a0, -16($fp)   # the value of the string to be printed
        syscall

        # print end of line
        .data
endl:   .asciiz "\n"
        .text
        li      $v0, 4
        la      $a0, endl
        syscall

        # loop back
        b       loop
```

Factorial with Frames

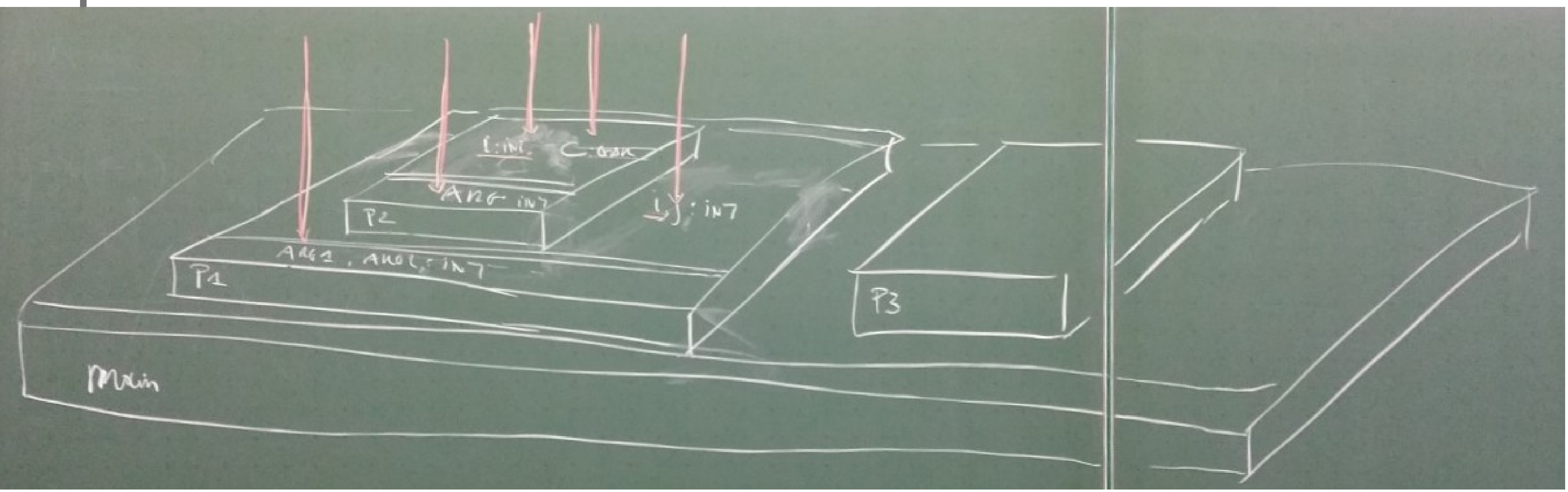
```
exit:    # return from main (restore registers and pop frame)
         lw      $ra, -4($fp)      # the old frame pointer
         move    $sp, $fp          # pop frame off the stack
         lw      $fp, ($sp)        # restore old frame pointer
         jr      $ra               # return to caller (of the fact program)
```

Frame Structure (with static links)



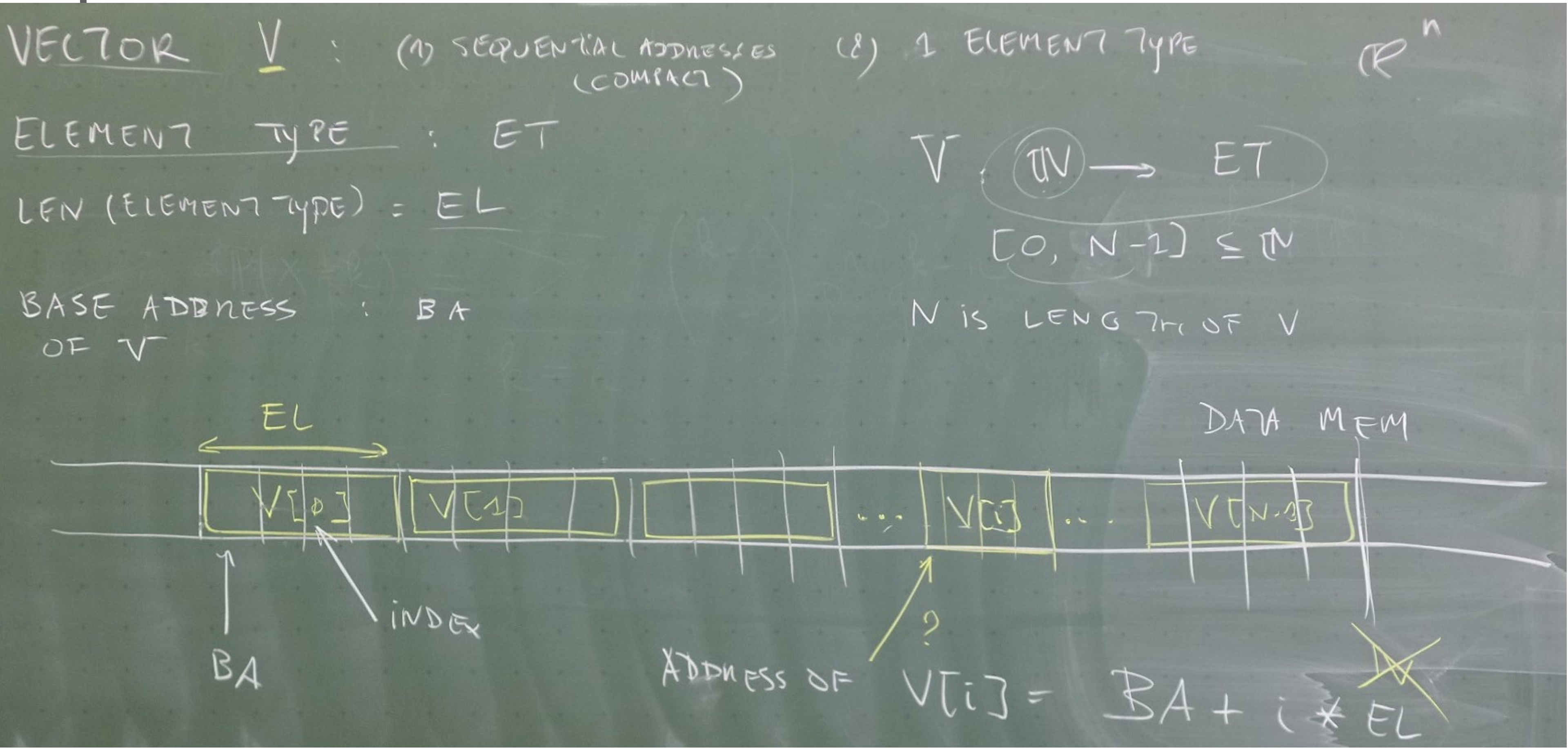
Not in CSA, but in Compilers (Ba2)

“nested scope”

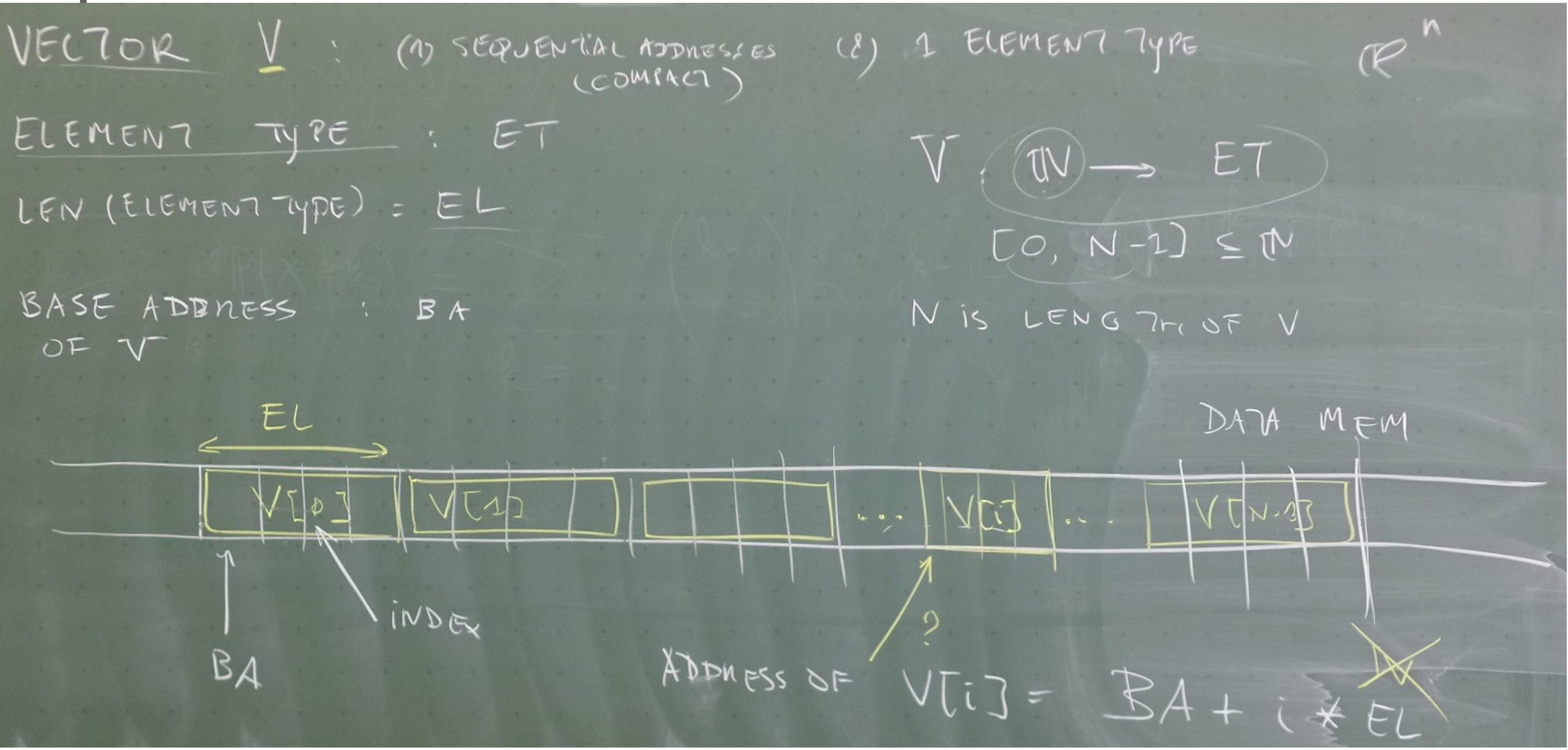


Not in CSA, but in Compilers (Ba2)

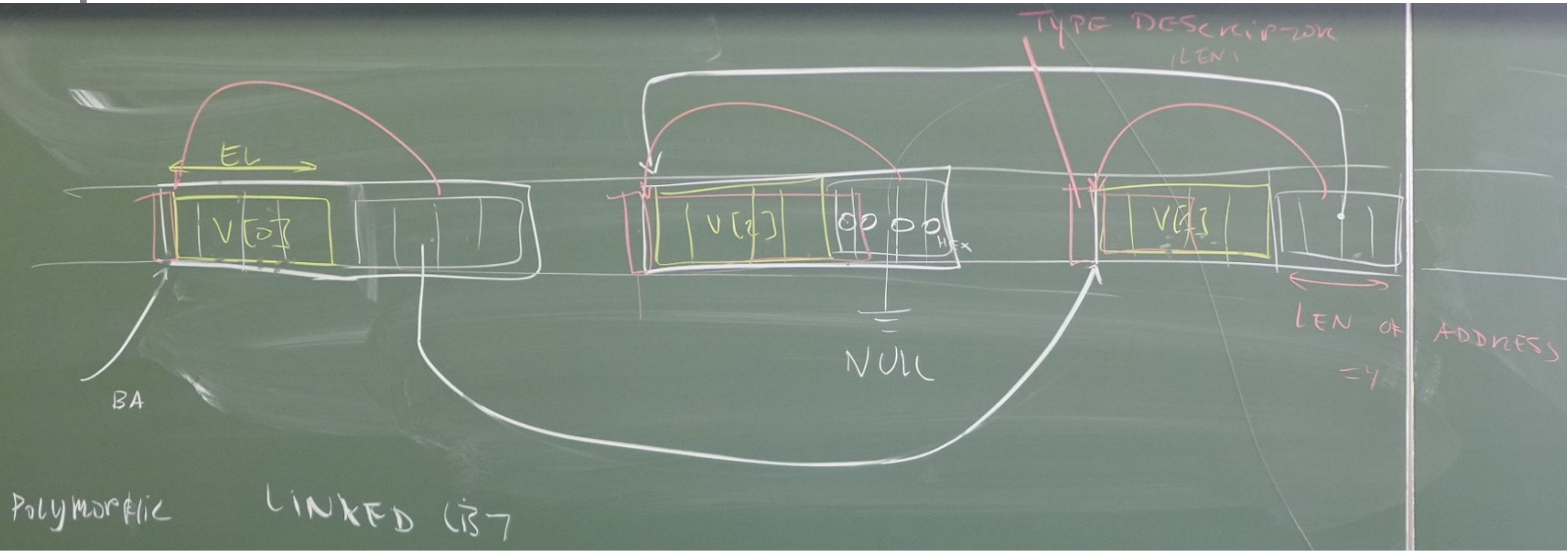
Matrix 1D (vector), static size, single type



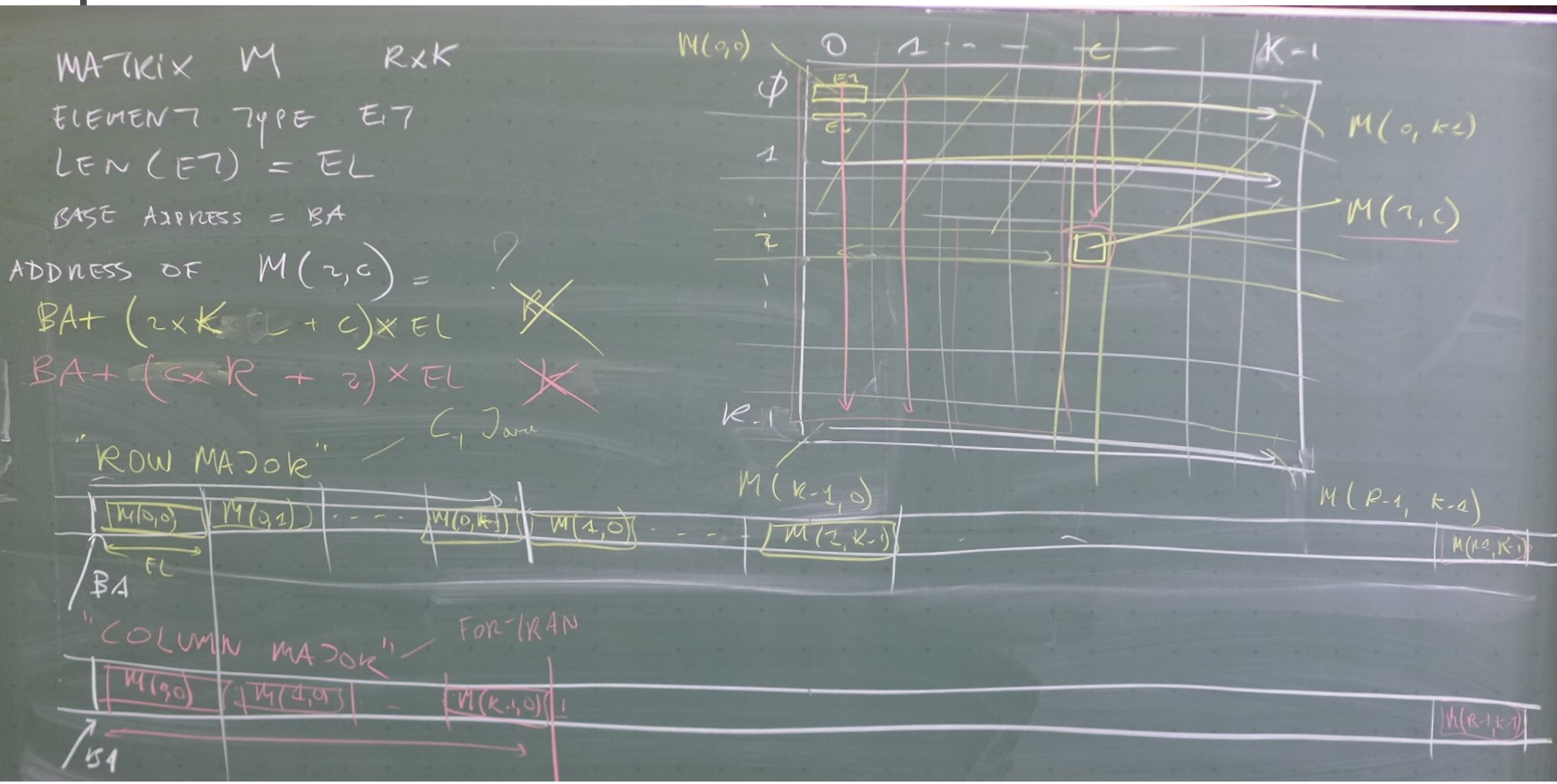
Matrix 1D (vector), dynamic size, single type



Matrix 1D (vector), dynamic size(, multi-type)



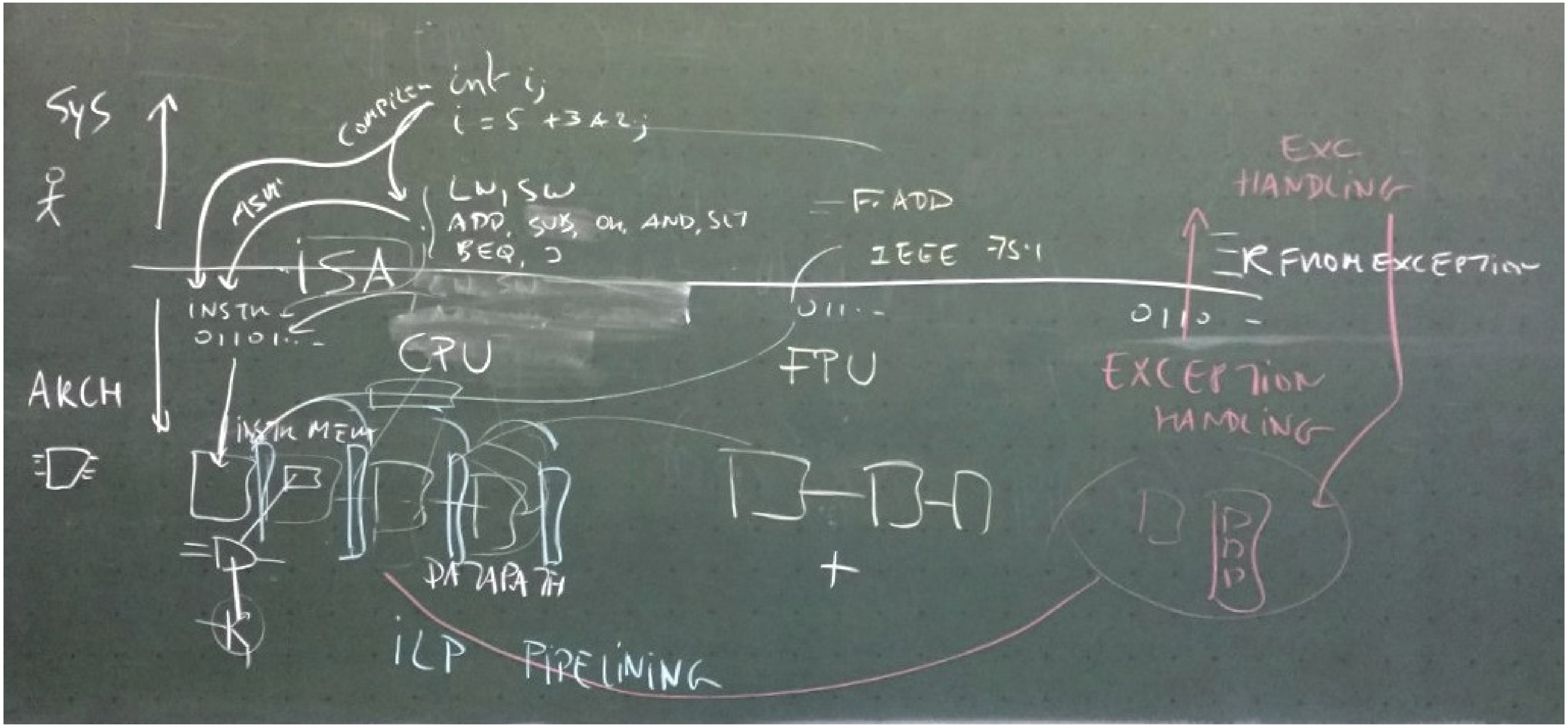
Matrix 2D, static size, single type



Character Data ... operations

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: universal character set
  - Used in `Java`, `C++` wide characters, …
  - Most of the world's alphabets, plus symbols
  - **UTF-8**, UTF-16: *variable-length* encodings

"big picture" (snapshot)



Byte/Halfword Operations

- Could use words + **bitwise** operations
- MIPS: **b**yte/**h**alfword load/store

  String processing is a common case

`lb rt, offset(rs)`   `lh rt, offset(rs)`

  - **Sign** extend to 32 bits in `rt`

`lbu rt, offset(rs)`   `lhu rt, offset(rs)`

  - **Zero** extend to 32 bits in `rt`

`sb rt, offset(rs)`   `sh rt, offset(rs)`

  - Store just rightmost byte/halfword

String Copy Example

- C code (naïve):
  - Null-terminated ASCII string (`.asciiz`)

```
void strcpy (char x[], char y[])
{ int i;
   i = 0;
   while ((x[i]=y[i])!='\0')
      i += 1;
}
```

  - Addresses of `x,y` in `$a0,$a1`
  - `i` in `$s0`

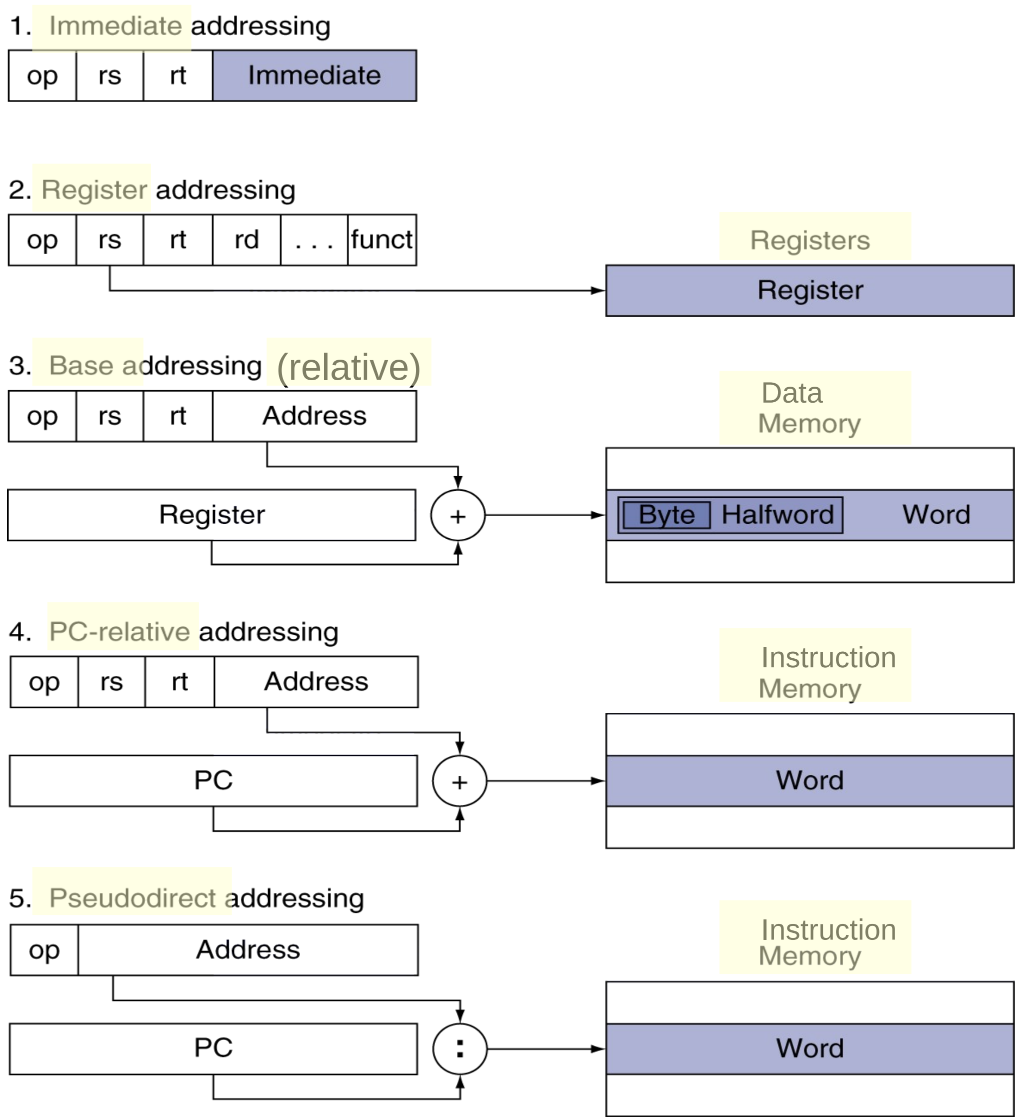
```
/* Caveat: (b &&  (x[i]=y[i])) evaluation in C */
```

String Copy Example

- MIPS code:

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    add  $s0, $zero, $zero# i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i] (includes \0)
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1 … what if int[] args?
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

Addressing Modes (data/instr)



6. indirect

JR $R

Branch Addressing

- Branch instructions specify
  - opcode, two registers, target address
- Most branch targets are **near** branch instruction ("locality")
  - forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- **PC-relative** addressing
  - Target address = **PC + offset × 4**
  - PC already incremented by 4 by this time

Jump Addressing

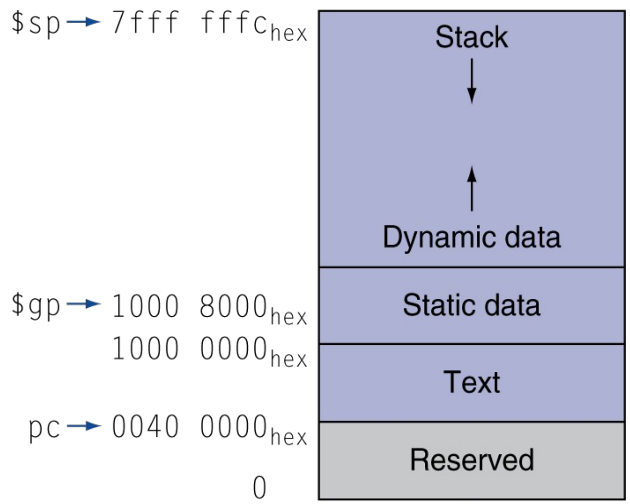
- Jump (`j` and `jal`) targets could be (almost) **anywhere** in **text segment**
  - encode (almost) full address in instruction

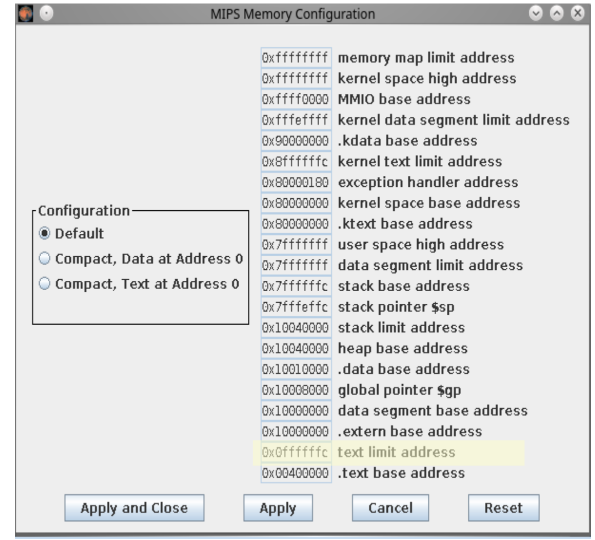
op	address
6 bits	26 bits

- (pseudo)**Direct** jump addressing
  - Target address = $\mathbf{PC_{31\ldots28} : (address \times 4)}$

Assembling Example (J)

```
      .text   0x1C50083C
loop: sll     $t1, $s3, 2
      add     $t2, $t1, $s6
      lw      $t0, -4($t2)
      bne     $t0, $s5, exit
      addi    $s3, $s4, -1
      j       loop
exit:
```






```
      .text   0x1C50083C
```

Address not allowed!
This is a synthetic example to demonstrate use of top 4 bits in address calculation in the `J` instruction

Assembling Example (J)

```
       .text   0x1C50083C
loop:  sll     $t1, $s3, 2
       add     $t2, $t1, $s6
       lw      $t0, -4($t2)
       bne     $t0, $s5, exit
       addi    $s3, $s4, -1
       j       loop
exit:
```

```
       .text   0x1C50083C
loop:  sll     $9, $19, 2
       add     $10, $9, $22
       lw      $8, 0xFFFC($10)
       bne     $8, $21, 0x0002
       addi    $19, $20, 0xFFFF
       j       loop
exit:
```

Assembling Example (J)

```
        .text   0x1C50083C
loop:   sll     $9, $19, 2
        add     $10, $9, $22
        lw      $8, 0xFFFC($10)
        bne     $8, $21, 0x0002
        addi    $19, $20, 0xFFFF
        j       loop
exit:
```

```
0x1C50083C  sll   $9, $19, 2
         →  0x00  0 19 9  2  0
0x1C500840  add   $10, $9, $22
         →  0x00  9 22 10 0 0x20
0x1C500844  lw    $8, 0xFFFC($10)
         →  0x23  10 8  0xFFFC
0x1C500848  bne   $8, $21, 0x0002
         →  0x05  8 21  0x0002
0x1C50084C  addi  $19, $20, 0xFFFF
         →  0x08  20 19  0xFFFF
0x1C500850  j     loop
         →  0x02  0x314020F
```

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

	31–26	25–21	20–16	15–11	10–6	5–0
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	address				

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

	31–26	25–21	20–16	15–11	10–6	5–0
FR	opcode	fmt	ft	fs	fd	funct
FI	opcode	fmt	ft	immediate		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Assembling Example (J)

```
0x1C50083C  sll   $9, $19, 2
   →        0x00  0 19 9 2 0
0x1C500840  add   $10, $9, $22
   →        0x00  9 22 10 0 0x20
0x1C500844  lw    $8, 0xFFFC($10)
   →        0x23  10 8 0xFFFC
0x1C500848  bne   $8, $21, 0x0002
   →        0x05  8 21 0x0002
0x1C50084C  addi  $19, $20, 0xFFFF
   →        0x08  20 19 0xFFFF
0x1C500850  j     loop
   →        0x02  0x314020F
0x1C500854  …                             # next sequential address
```

```
Jump target   =   1    C    5    0    0    8    3    C(hex)
              =   0001 1100 0101 0000 0000 1000 0011 1100(bin)
 (1C500854(hex) = 0001 1100 0101 0000 0000 1000 0101 0100(bin))
              =   0001 11 0001 0100 0000 0010 0000 1111 00(bin)

26 bit address         11 0001 0100 0000 0010 0000 1111(bin)

                       3  1    4    0    2    0    F(hex)
```

Assembling Example (J)

```
0x1C50083C  sll  $9, $19, 2
    →       0x00 0 19 9 2 0          000000 00000 10011 01001 00010 000000bin
0x1C500840  add  $10, $9, $22
    →       0x00 9 22 10 0 0x20      000000 01001 10110 01010 00000 100000bin
0x1C500844  lw   $8, 0xFFFC($10)
    →       0x23 10 8 0xFFFC         100011 01010 01000 1111 1111 1111 1100bin
0x1C500848  bne  $8, $21, 0x0002
    →       0x05 8 21 0x0002         000101 01000 10101 0000 0000 0000 0010bin
0x1C50084C  addi $19, $20, 0xFFFF
    →       0x08 20 19 0xFFFF        001000 10100 10011 1111 1111 1111 1111bin
0x1C500850  j    loop
    →       0x02 0x314020F           000010 11 0001 0100 0000 0010 0000 1111bin
                    26 bit
0x1C500854  …
```

Assembling Example (J)

000000 00000 10011 01001 00010 000000$_{bin}$	0000 0000 0001 0011 0100 0100 1000 0000$_{bin}$
000000 01001 10110 01010 00000 100000$_{bin}$	0000 0001 0011 0110 0101 0000 0010 0000$_{bin}$
100011 01010 01000 1111 1111 1111 1100$_{bin}$	1000 1101 0100 1111 1111 1111 1111 1100$_{bin}$
000101 01000 10101 0000 0000 0000 0010$_{bin}$	0001 0101 0001 0101 0000 0000 0000 0010$_{bin}$
001000 10100 10011 1111 1111 1111 1111$_{bin}$	0010 0010 1001 0011 1111 1111 1111 1111$_{bin}$
000010 11 0001 0100 0000 0010 0000 1111$_{bin}$	0000 1011 0001 0100 0000 0010 0000 1111$_{bin}$

Assembling Example (J)

0000 0000 0001 0011 0100 0100 1000 0000_{bin}	0x1C50083C	00134880_{hex}
0000 0001 0011 0110 0101 0000 0010 0000_{bin}	0x1C500840	01365020_{hex}
1000 1101 0100 1111 1111 1111 1111 1100_{bin}	0x1C500844	$8D48FFFC_{hex}$
0001 0101 0001 0101 0000 0000 0000 0010_{bin}	0x1C500848	15150002_{hex}
0010 0010 1001 0011 1111 1111 1111 1111_{bin}	0x1C50084C	$2293FFFF_{hex}$
0000 1011 0001 0100 0000 0010 0000 1111_{bin}	0x1C500850	**$0B14020F_{hex}$**
	0x1C500854	…

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x00134880	sll $9,$19,0x00000002	2: loop: sll $t1, $s3, 2
	0x00400004	0x01365020	add $10,$9,$22	3: add $t2, $t1, $s6
	0x00400008	0x8d48fffc	lw $8,0xfffffffc($10)	4: lw $t0, -4($t2)
	0x0040000c	0x15150002	bne $8,$21,0x00000002	5: bne $t0, $s5, exit
	0x00400010	0x2293ffff	addi $19,$20,0xfff...	6: addi $s3, $s4, -1
	0x00400014	0x08100000	j 0x00400000	7: j loop

disAssembling Example (J)

Decoding machine code (aka “disassembling”)

```
0x1C500850   0B14020F          0x1C500850   0000 1011 0001 0100 0000 0010 0000 1111bin
0x1C500854   …                 0x1C500854   …
```

Opcode of Jump (**J**)

```
          1C500854hex =   0001 1100 0101 0000 0000 1000 0101 0100bin
 (26 bit) 314020Fhex  =          11 0001 0100 0000 0010 0000 1111bin
     Jump target      =   0001 11 0001 0100 0000 0010 0000 1111 00bin
                      =   0001 1100 0101 0000 0000 1000 0011 1100bin
                      =      1    C    5    0    0    8    3    Chex
```

Branching Far Away

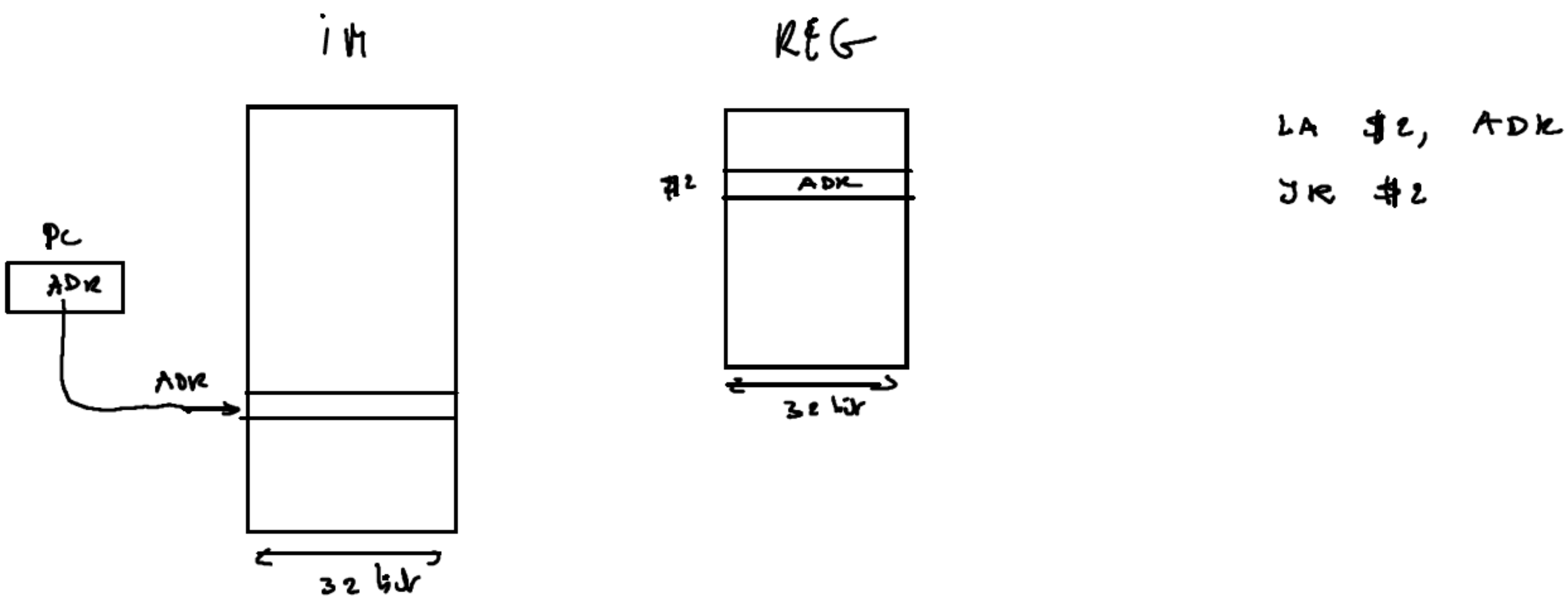
- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
            ↓
        bne $s0,$s1, L2
        j L1
    L2: …
```

Even Farther Away: JR

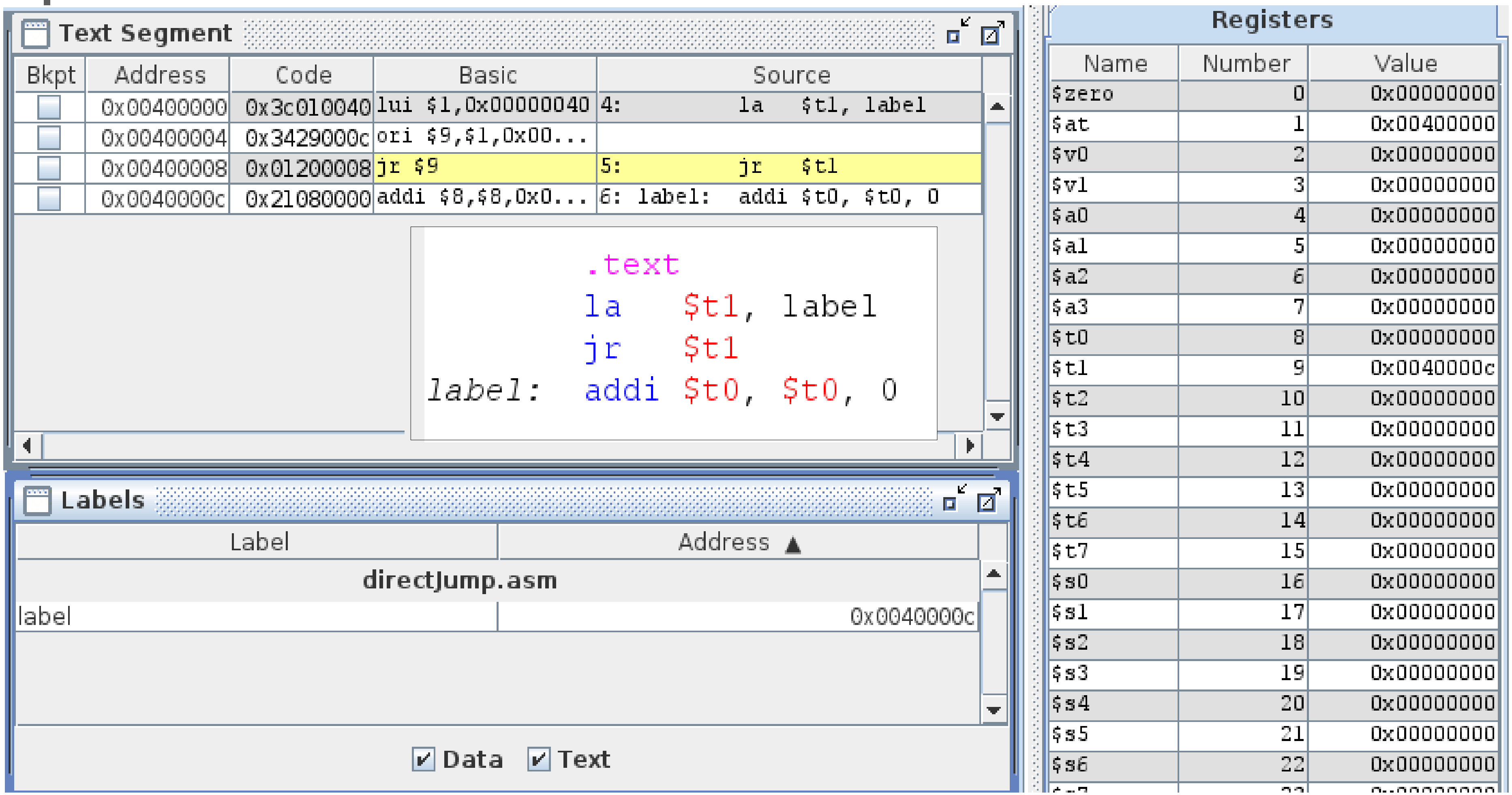
Full 32 bit address

ABSOLUTE, INDIRECT Jump

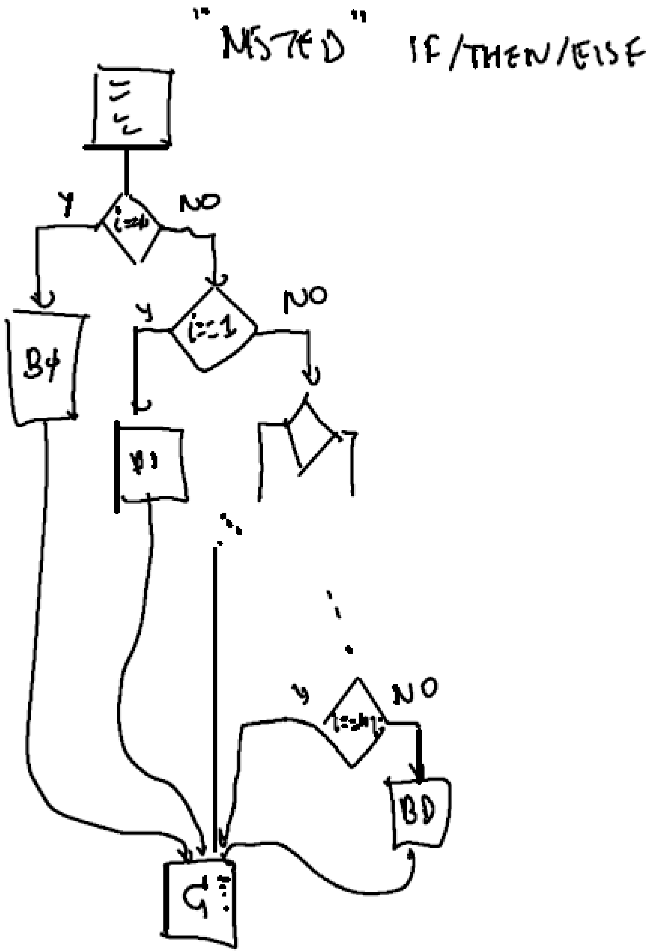


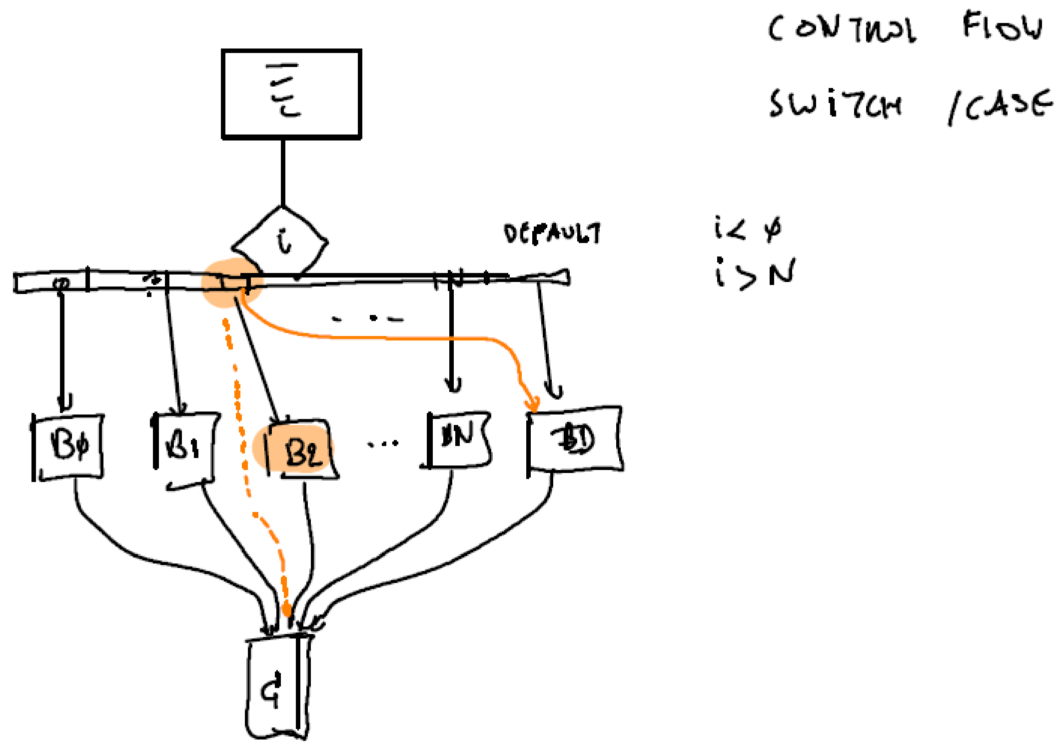
Even Farther Away: JR

Full 32 bit address



Switch statement (aka "computed jump")





"control (flow) **indirection**"

Data Indirection: example

```
        .data
        .align  2               # .align n: align on 2^n boundary
ptrTbl: .space  16          # allocate 4 consecutive words,
                            # with storage uninitialized,
                            # to store 4 pointers
value0: .word   1           # 4 bytes, value 1, aligned on word boundary
        .space  16
value1: .word   2           # 4 bytes, value 2, aligned on word boundary
        .space  7           # does not end on word boundary
value2: .word   -1          # 4 bytes, value -1, aligned on word boundary
        .space  32
value3: .word   3           # 4 bytes, value 2, aligned on word boundary


        .text

#       fill ptrTbl with adresses of value0 .. value3
        la      $s0, ptrTbl    # $s0 contains the address of ptrTbl
        la      $t0, value0
        sw      $t0, 0($s0)    # dataMEM[ADDRESS(ptrTbl)+  0]  = ADDRESS(value0)
        la      $t0, value1
        sw      $t0, 4($s0)    # dataMem[ADDRESS(ptrTbl)+  4]  = ADDRESS(value1)
        la      $t0, value2
        sw      $t0, 8($s0)    # dataMEM[ADDRESS(ptrTbl)+  8]  = ADDRESS(value2)
        la      $t0, value3
        sw      $t0, 12($s0)   # dataMEM[ADDRESS(ptrTbl)+ 12]  = ADDRESS(value3)

# more compact: let the assembler figure out the addresses in ptrTbl
#       .data
#       .word value0, value1, value2, value3
```

Data Indirection: example

```
#       logic to encode:
#
#       for i in 0..3:
#         address = dataMEM[ADDRESS(ptrTbl) + 4*i]
#         dataMEM[address] += 1
#
#       with only (assembler) primitive if and goto:
#
#       address = ADDRESS(ptrTbl) + 4*3
# for:  dataMEM[address] += 1
#       address -= 4
#       if address >= ADDRESS(ptrTbl) goto: for


        addi    $t1, $s0, 12  # $t1 is pointer to elements (words) of ptrTbl (starting with the last)
for:    lw      $t2, 0($t1)   # $t2 is the data in the elements of ptrTbl:
                              # the address of the data to be incremented
        lw      $t3, 0($t2)   # the data to be incremented
        addi    $t3, $t3, 1   # increment
        sw      $t3, 0($t2)   # put incremented value back in memory
        subi    $t1, $t1, 4
        bge     $t1, $s0, for

#   cleanly exit to OS
        li      $v0, 10
        syscall
```

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- **Pseudoinstructions**:
  “expanded” by the assembler

```
move $t0, $t1        → add $t0, $zero, $t1
blt  $t0, $t1, L     → slt $at, $t0, $t1
                       bne $at, $zero, L
```

`$at` (register 1): **a**ssembler **t**emporary

Assembler "macro"s

- User-defined patterns,
  "expanded" by the assembler
- Increased readability (but harder to debug)

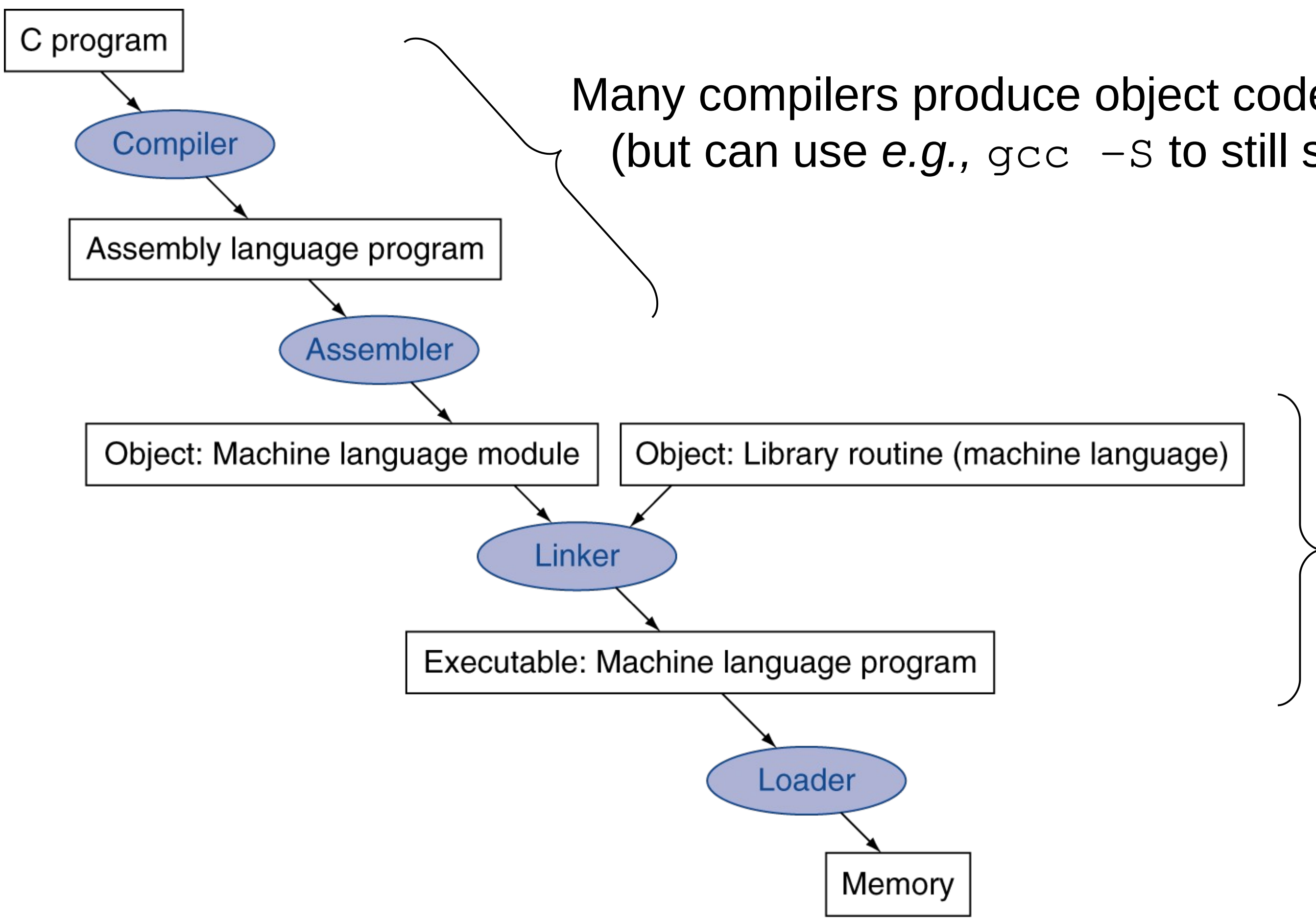
  Don't go overboard as others may not understand your new "language"!
- **macros**: "expanded" by the assembler

```
.eqv   INCR      100
.eqv   CTR       $t2

addi CTR, CTR, INCR
```

```
.macro done
li $v0,10
syscall
.end_macro

done
```

```
.macro terminate (%termination_value)
li $a0, %termination_value
li $v0, 17
syscall
.end_macro

terminate (1)
```

Translation and Startup



Many compilers produce object code modules directly (but can use *e.g.*, `gcc -S` to still see assembly)

Static linking

Translation and Startup

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

Translation and Startup

```
        .text
        .align   2                # .align n: align on 2^n boundary
        .globl   main
main:
        subu     $sp, $sp, 32
        sw       $ra, 20($sp)
        sd       $a0, 32($sp)
        sw       $0,  24($sp)
        sw       $0,  28($sp)
loop:
        lw       $t6, 28($sp)
        mul      $t7, $t6, $t6
        lw       $t8, 24($sp)
        addu     $t9, $t8, $t7
        sw       $t9, 24($sp)
        addu     $t0, $t6, 1
        sw       $t0, 28($sp)
        ble      $t0, 100, loop
        la       $a0, str
        lw       $a1, 24($sp)
        jal      printf
        move     $v0, $0
        lw       $ra, 20($sp)
        addu     $sp, $sp, 32
        jr       $ra

        .data
        .align   0                # align on byte boundary
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

Translation and Startup

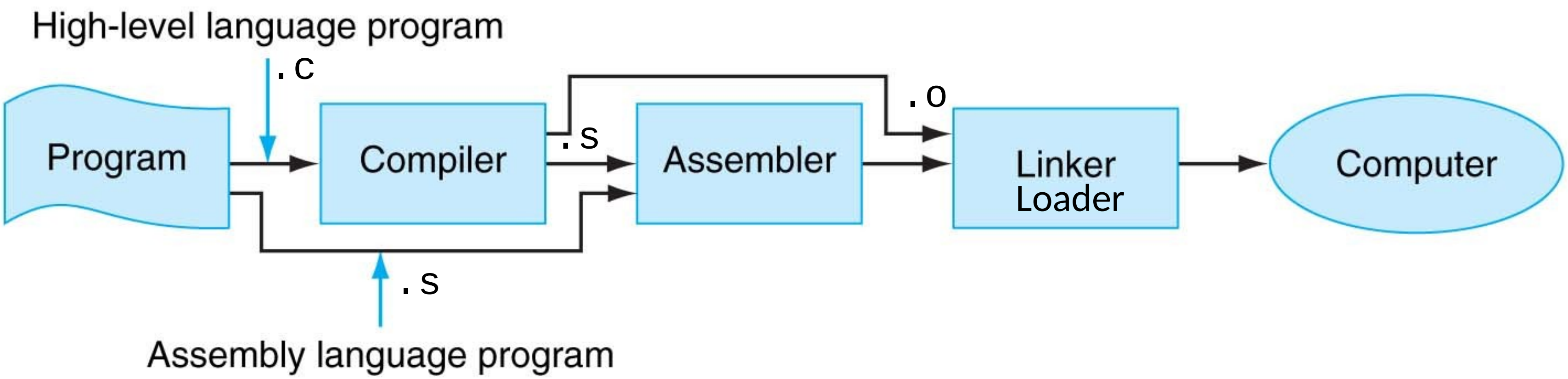
```
addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4,  32($29)
sw       $5,  36($29)
sw       $0,  24($29)
sw       $0,  28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu    $14, $14
addiu    $8,  $14, 1
slti     $1,  $8, 101
sw       $8,  28($29)
mflo     $15
addu     $25, $24, $15
bne      $1,  $0, -9
sw       $25, 24($29)
lui      $4,  4096
lw       $5,  24($29)
jal      1048812
addiu    $4,  $4, 1072
lw       $31, 20($29)
addiu    $29, $29, 32
jr       $31
move     $2,  $0
```

Translation and Startup

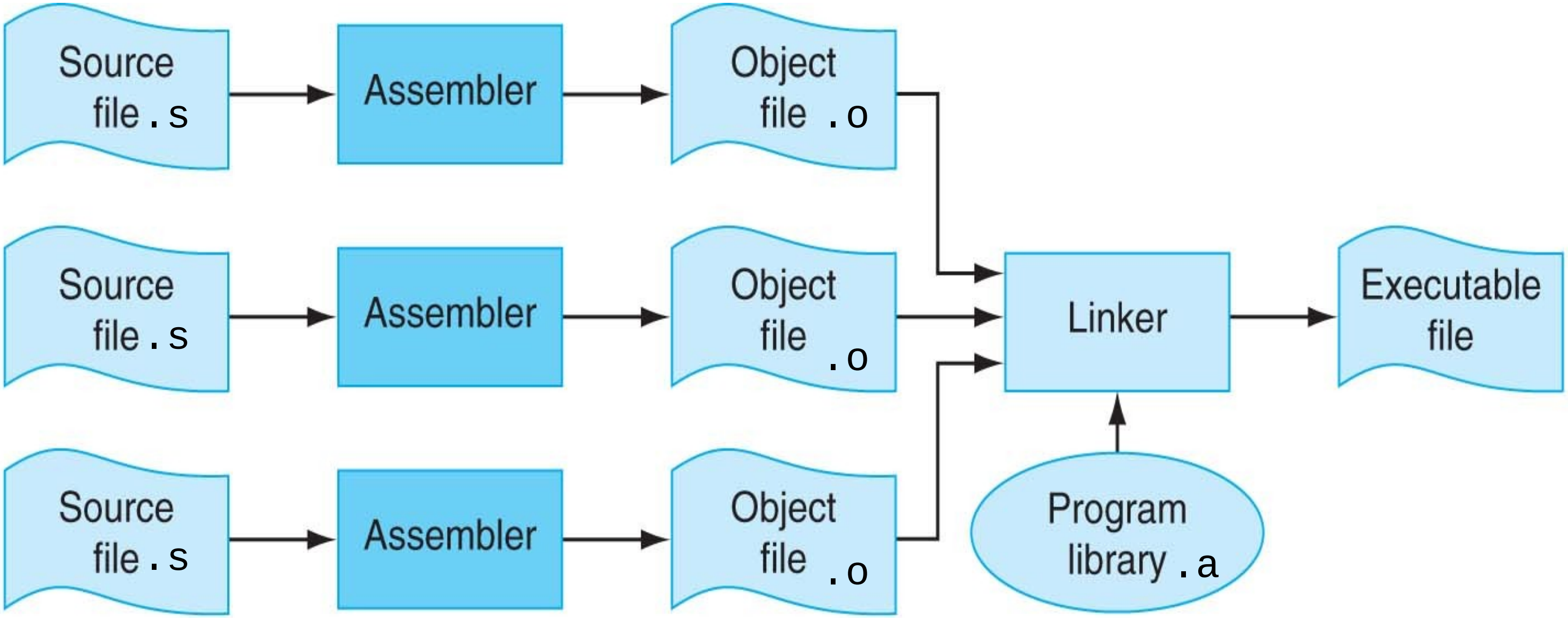
1 MIPS instruction – 32 bit

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

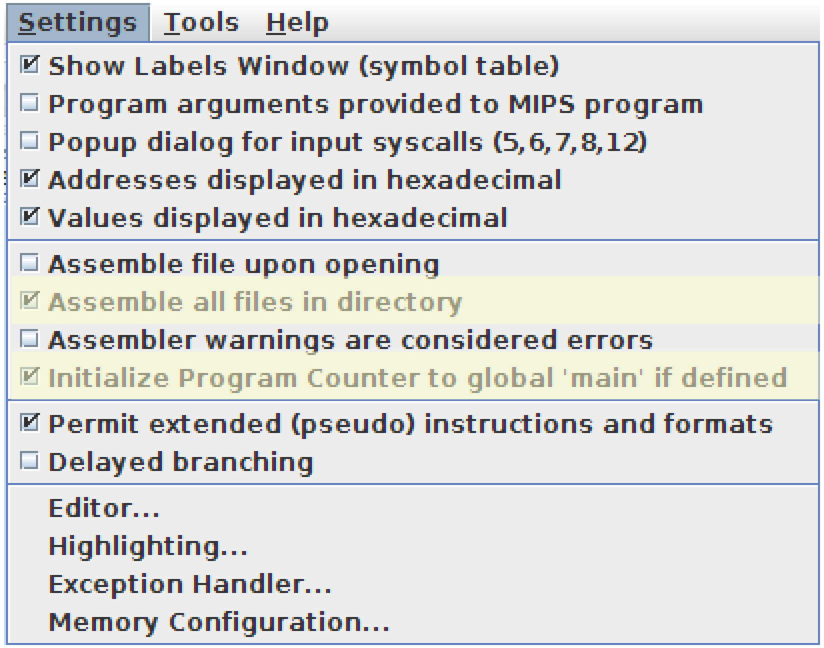
Translation and Startup

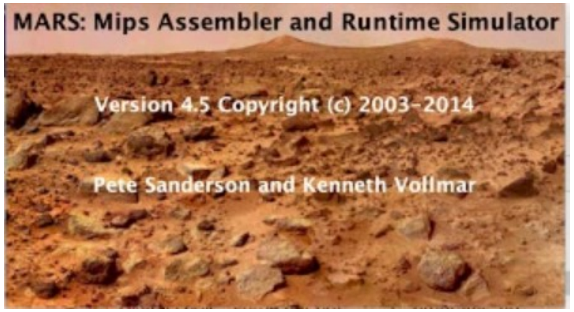


Translation and Startup, in pieces



Static Linking (local, global visibility) in MARS





This is called the "**entry point**",
By default, this is the first line in the .text segment.

Static Linking in MARS

main.asm | program1.asm | program2.asm | printInProg.asm | macros.asm

```
.macro cleanProgramExit
        # clean exit of program
        li $v0, 10
        syscall       # exit()
.end_macro

.macro push(%register)
        subiu  $sp, $sp, 4  # stack (of words) grows downwards
        sw     %register, 0($sp) # $sp points to last stack entry
.end_macro

.macro pop(%register)
        lw     %register, 0($sp) # $sp points to last stack entry
        addiu  $sp, $sp, 4  # stack (of words) grows downwards
.end_macro
```

Static Linking in MARS

main.asm | program1.asm | program2.asm | printInProg.asm | macros.asm

```
#
# main.asm
#
# main program (entry point main:) which calls subroutines
#     prog1
#     prog2
#
# note how main has no local data defined

.include "macros.asm"

# text (code) segment

         .text #0x00400000

defaultEntry:   # execution starts here if 'main' is not explicitly made entry point in MARS Settings
                # main must also be globally visible for this to work

         # call printProgX, with argument 0 -> will print prog0
         li      $a0, 0
         jal     printProgX

         .globl  main  # make symbol (instruction address main) globally visible for linking
         # when commented, 'main' is not globally visible and execution will start at defaultEntry:
main:
         jal      prog1 # call prog1
         jal      prog2 # call prog2

         cleanProgramExit
```

Static Linking in MARS

main.asm | program1.asm | program2.asm | printInProg.asm | macros.asm

```
#
# program1.asm
#
#    holds integer variable w1 with value 255 (0x000000ff)
#    makes (the address of) w1 globally visible for referencing from other linked binaries
#
#    defines subroutine prog1
#    makes (the address of) prog1 globally visible for referencing from other linked binaries
#    prog1 prints integer variables w1 and w2 by calling printProgX
#

.include "macros.asm"

# text (code) segment

        .text   #0x00420000
        .globl  prog1     # make symbol (instruction address prog1) globally visible for linking
prog1:
         # save $ra by pushing on the stack
         push($ra)

         # call printProgX, with argument 1 (in prog1)
         lw      $a0, progNr
         jal     printProgX

         # restore $ra by popping from the stack
         pop($ra)

         # return to prog1's caller
         jr      $ra

# data segment

         .data   # 0x10010010

progNr:  .word   1          # only visible locally

         .globl w1          # make symbol (data address w1) globally visible for linking
w1:      .word   255
```

Static Linking in MARS

main.asm | program1.asm | program2.asm | printlnProg.asm | macros.asm

```
#
# program2.asm
#
#    holds integer variable w2 with value -1 (0xffffffff)
#    makes (the address of) w2 globally visible for referencing from other linked binaries
#
#    defines subroutine prog2
#    makes (the address of) prog2 globally visible for referencing from other linked binaries
#    prog2 prints integer variables w1 and w2 by calling printProgX
#

.include "macros.asm"

# text (code) segment

        .text   #0x00440000
        .globl  prog2     # make symbol (instruction address prog2) globally visible for linking
prog2:
         # save $ra by pushing on the stack
         push($ra)

         # call printProgX, with argument 2 (in prog2)
         lw      $a0, progNr
         jal     printProgX

         # restore $ra by popping from the stack
         pop($ra)

         # return to prog2's caller
         jr      $ra

# data segment

         .data   # 0x10010020

progNr:  .word   2          # only visible locally

         .globl w2          # make symbol (data address w2) globally visible for linking
w2:      .word   -1
```

Static Linking in MARS

main.asm | program1.asm | program2.asm* | printInProg.asm | macros.asm

```
#
# printInProg.asm
#
#    implements the function printProgX
#      takes one integer argument (passed in $a0) indicating whether we're in prog1 or prog2
#      prints the values of w1 and w2 (which are externally defined and only available through linking)
#

# text (code) segment

        .text   #0x00430000
        .globl  printProgX     # make symbol (instruction address printProgX) globally visible for linking
printProgX:
         # print "in Prog X:", with X from function argument (in $a0)

         # fill prognr: with the appropriate digit (computed from $a0)
         #              to complete the inProgX: "template"
         addiu  $t1, $a0, '0' # ascii value of the digit = integer value of digit + '0'
         sb     $t1, prognr

         la     $a0, inProgX
         li     $v0, 4    # print string syscall
         syscall
```

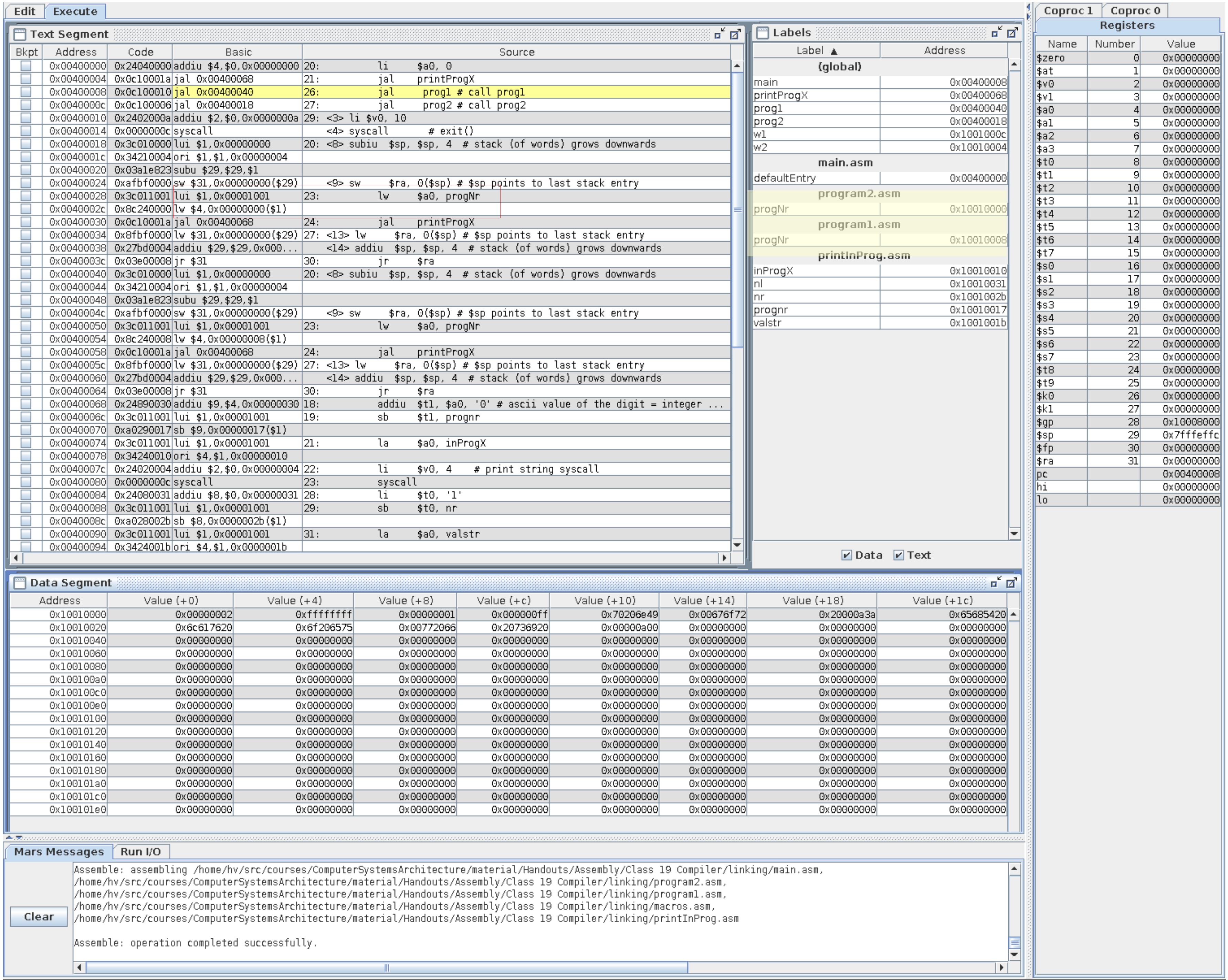
Static Linking in MARS

main.asm | program1.asm | program2.asm* | printlnProg.asm | macros.asm

```
        # print w1

        # fill nr: with the digit '1' to complete the valstr: "template"
        li      $t0, '1'
        sb      $t0, nr

        la      $a0, valstr
        li      $v0, 4      # print string syscall
        syscall

        lw      $a0, w1     # w1 is external (defined in program1.asm)   <---
        li      $v0, 1      # print integer syscall
        syscall

        la      $a0, nl     # end with newline
        li      $v0, 4      # print string syscall
        syscall

        # print w2

        # fill nr: with the digit '2' to complete the valstr: "template"
        li      $t0, '2'
        sb      $t0, nr

        la      $a0, valstr
        li      $v0, 4      # print string syscall
        syscall

        lw      $a0, w2     # w2 is external (defined in program2.asm)   <---
        li      $v0, 1      # print integer syscall
        syscall

        la      $a0, nl     # end with newline
        li      $v0, 4      # print string syscall
        syscall

        # return to printProgX's caller
        jr      $ra
```

Static Linking in MARS

```
# data segment
         .data # 0x10010030

# the following are only locally visible (for printing informative messages)

inProgX: .ascii  "In prog"
prognr:  .space  1        # 1 byte placeholder for the ASCII code of digits '1' or '2' (prog1 or prog2)
         .asciiz ":\n"

valstr:  .ascii  "  The value of w"
nr:      .space  1        # 1 byte placeholder for the ASCII code of digits '1' or '2' (w1 or 2w)
         .asciiz " is "
nl:      .asciiz "\n"
```



Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24040000	addiu $4,$0,0x00000000	20: li $a0, 0
	0x00400004	0x0c10001a	jal 0x00400068	21: jal printProgX
	0x00400008	0x0c100010	jal 0x00400040	26: jal prog1 # call prog1
	0x0040000c	0x0c100006	jal 0x00400018	27: jal prog2 # call prog2
	0x00400010	0x2402000a	addiu $2,$0,0x0000000a	29: <3> li $v0, 10
	0x00400014	0x0000000c	syscall	<4> syscall # exit()
	0x00400018	0x3c010000	lui $1,0x00000000	20: <8> subiu $sp, $sp, 4 # stack (of words) grows downwards
	0x0040001c	0x34210004	ori $1,$1,0x00000004	
	0x00400020	0x03a1e823	subu $29,$29,$1	
	0x00400024	0xafbf0000	sw $31,0x00000000($29)	<9> sw $ra, 0($sp) # $sp points to last stack entry
	0x00400028	0x3c011001	lui $1,0x00001001	23: lw $a0, progNr
	0x0040002c	0x8c240000	lw $4,0x00000000($1)	
	0x00400030	0x0c10001a	jal 0x00400068	24: jal printProgX
	0x00400034	0x8fbf0000	lw $31,0x00000000($29)	27: <13> lw $ra, 0($sp) # $sp points to last stack entry
	0x00400038	0x27bd0004	addiu $29,$29,0x000...	<14> addiu $sp, $sp, 4 # stack (of words) grows downwards
	0x0040003c	0x03e00008	jr $31	30: jr $ra
	0x00400040	0x3c010000	lui $1,0x00000000	20: <8> subiu $sp, $sp, 4 # stack (of words) grows downwards
	0x00400044	0x34210004	ori $1,$1,0x00000004	
	0x00400048	0x03a1e823	subu $29,$29,$1	
	0x0040004c	0xafbf0000	sw $31,0x00000000($29)	<9> sw $ra, 0($sp) # $sp points to last stack entry
	0x00400050	0x3c011001	lui $1,0x00001001	23: lw $a0, progNr
	0x00400054	0x8c240008	lw $4,0x00000008($1)	
	0x00400058	0x0c10001a	jal 0x00400068	24: jal printProgX
	0x0040005c	0x8fbf0000	lw $31,0x00000000($29)	27: <13> lw $ra, 0($sp) # $sp points to last stack entry
	0x00400060	0x27bd0004	addiu $29,$29,0x000...	<14> addiu $sp, $sp, 4 # stack (of words) grows downwards
	0x00400064	0x03e00008	jr $31	30: jr $ra
	0x00400068	0x24890030	addiu $9,$4,0x00000030	18: addiu $t1, $a0, '0' # ascii value of the digit = integer ...
	0x0040006c	0x3c011001	lui $1,0x00001001	19: sb $t1, prognr
	0x00400070	0xa0290017	sb $9,0x00000017($1)	
	0x00400074	0x3c011001	lui $1,0x00001001	21: la $a0, inProgX
	0x00400078	0x34240010	ori $4,$1,0x00000010	
	0x0040007c	0x24020004	addiu $2,$0,0x00000004	22: li $v0, 4 # print string syscall
	0x00400080	0x0000000c	syscall	23: syscall
	0x00400084	0x24080031	addiu $8,$0,0x00000031	28: li $t0, '1'
	0x00400088	0x3c011001	lui $1,0x00001001	29: sb $t0, nr
	0x0040008c	0xa028002b	sb $8,0x0000002b($1)	
	0x00400090	0x3c011001	lui $1,0x00001001	31: la $a0, valstr
	0x00400094	0x3424001b	ori $4,$1,0x0000001b	

Labels

Label ▲	Address
(global)	
main	0x00400008
printProgX	0x00400068
prog1	0x00400040
prog2	0x00400018
w1	0x1001000c
w2	0x10010004
main.asm	
defaultEntry	0x00400000
program2.asm	
progNr	0x10010000
program1.asm	
progNr	0x10010008
printInProg.asm	
inProgX	0x10010010
nl	0x10010031
nr	0x1001002b
prognr	0x10010017
valstr	0x1001001b

☑ Data ☑ Text

Coproc 1 | Coproc 0

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400008
hi		0x00000000
lo		0x00000000

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000002	0xffffffff	0x00000001	0x000000ff	0x70206e49	0x00676f72	0x20000a3a	0x65685420
0x10010020	0x6c617620	0x6f206575	0x00772066	0x20736920	0x00000a00	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Mars Messages | Run I/O

Clear

```
Assemble: assembling /home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/main.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/program2.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/program1.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/macros.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/printInProg.asm

Assemble: operation completed successfully.
```

Mars Messages | Run I/O

Clear

```
Assemble: assembling /home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/main.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/program2.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/program1.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/macros.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/printInProg.asm

Assemble: operation completed successfully.
```

Labels

Label ▲	Address
(global)	
main	0x00400008
printProgX	0x00400068
prog1	0x00400040
prog2	0x00400018
w1	0x1001000c
w2	0x10010004
main.asm	
defaultEntry	0x00400000
program2.asm	
progNr	0x10010000
program1.asm	
progNr	0x10010008
printInProg.asm	
inProgX	0x10010010
nl	0x10010031
nr	0x1001002b
prognr	0x10010017
valstr	0x1001001b

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000002	0xffffffff	0x00000001	0x000000ff	0x70206e49	0x00676f72	0x20000a3a	0x65685420
0x10010020	0x6c617620	0x6f206575	0x00772066	0x20736920	0x00000a00	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Data Segment

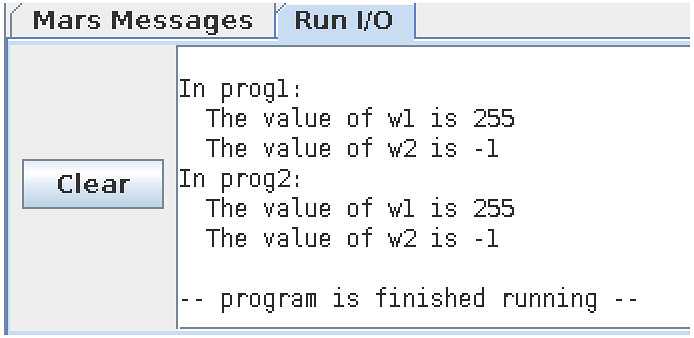
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	\0 \0 \0 .	. . . .	\0 \0 \0 .	\0 \0 \0 .	p n I	\0 g o r	\0 \n :	e h T
0x10010020	l a v	o e u	\0 w f	s i	\0 \0 \n \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10010000 (.data) | ☑ Hexadecimal Addresses | ☑ Hexadecimal Values | ☑ ASCII

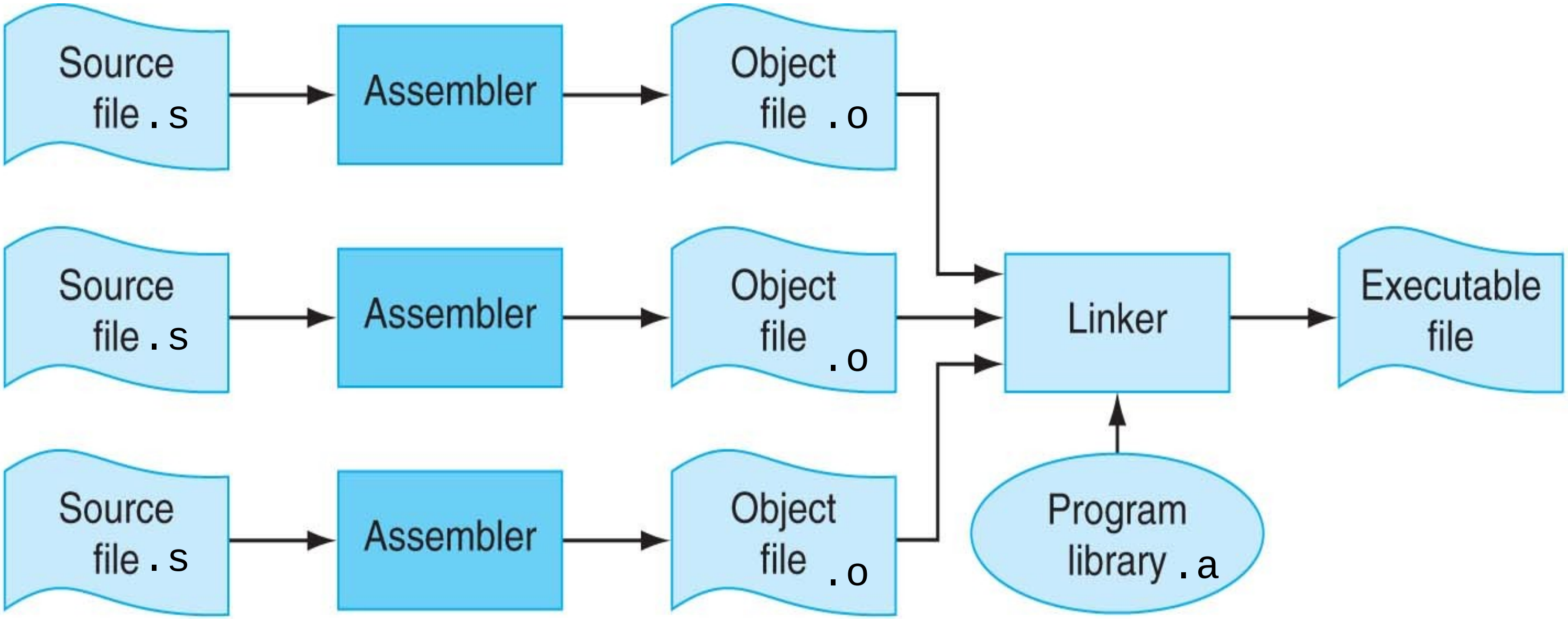
Text Segment

Label	Bkpt	Address	Code	Basic	Source	Annotation
defaultEntry:		0x00400000	0x24040000	addiu $4,$0,0x00000000	20: li $a0, 0	
main:		0x00400004	0x0c10001a	jal 0x00400068	21: jal printProgX	
		0x00400008	0x0c100010	jal 0x00400040	26: jal prog1 # call prog1	main.asm
		0x0040000c	0x0c100006	jal 0x00400018	27: jal prog2 # call prog2	
		0x00400010	0x2402000a	addiu $2,$0,0x0000000a	29: <3> li $v0, 10	cleanProgramExit
		0x00400014	0x0000000c	syscall	<4> syscall # exit()	
prog2:		0x00400018	0x3c010000	lui $1,0x00000000	20: <8> subiu $sp, $sp, 4 # stack (of words) grows downwards	
		0x0040001c	0x34210004	ori $1,$1,0x00000004		
		0x00400020	0x03a1e823	subu $29,$29,$1		push
		0x00400024	0xafbf0000	sw $31,0x00000000($29)	<9> sw $ra, 0($sp) # $sp points to last stack entry	
		0x00400028	0x3c011001	lui $1,0x00001001	23: lw $a0, progNr	
		0x0040002c	0x8c240000	lw $4,0x00000000($1)		program2.asm
		0x00400030	0x0c10001a	jal 0x00400068	24: jal printProgX	
		0x00400034	0x8fbf0000	lw $31,0x00000000($29)	27: <13> lw $ra, 0($sp) # $sp points to last stack entry	
		0x00400038	0x27bd0004	addiu $29,$29,0x000...	<14> addiu $sp, $sp, 4 # stack (of words) grows downwards	pop
		0x0040003c	0x03e00008	jr $31	30: jr $ra	
prog1:		0x00400040	0x3c010000	lui $1,0x00000000	20: <8> subiu $sp, $sp, 4 # stack (of words) grows downwards	
		0x00400044	0x34210004	ori $1,$1,0x00000004		
		0x00400048	0x03a1e823	subu $29,$29,$1		
		0x0040004c	0xafbf0000	sw $31,0x00000000($29)	<9> sw $ra, 0($sp) # $sp points to last stack entry	
		0x00400050	0x3c011001	lui $1,0x00001001	23: lw $a0, progNr	
		0x00400054	0x8c240008	lw $4,0x00000008($1)		program1.asm
		0x00400058	0x0c10001a	jal 0x00400068	24: jal printProgX	
		0x0040005c	0x8fbf0000	lw $31,0x00000000($29)	27: <13> lw $ra, 0($sp) # $sp points to last stack entry	
		0x00400060	0x27bd0004	addiu $29,$29,0x000...	<14> addiu $sp, $sp, 4 # stack (of words) grows downwards	
		0x00400064	0x03e00008	jr $31	30: jr $ra	
printProgX:		0x00400068	0x24890030	addiu $9,$4,0x00000030	18: addiu $t1, $a0, '0' # ascii value of the digit = integer ...	
		0x0040006c	0x3c011001	lui $1,0x00001001	19: sb $t1, prognr	
		0x00400070	0xa0290017	sb $9,0x00000017($1)		
		0x00400074	0x3c011001	lui $1,0x00001001	21: la $a0, inProgX	
		0x00400078	0x34240010	ori $4,$1,0x00000010		
		0x0040007c	0x24020004	addiu $2,$0,0x00000004	22: li $v0, 4 # print string syscall	
		0x00400080	0x0000000c	syscall	23: syscall	
		0x00400084	0x24080031	addiu $8,$0,0x00000031	28: li $t0, '1'	
		0x00400088	0x3c011001	lui $1,0x00001001	29: sb $t0, nr	
		0x0040008c	0xa028002b	sb $8,0x0000002b($1)		
		0x00400090	0x3c011001	lui $1,0x00001001	31: la $a0, valstr	
		0x00400094	0x3424001b	ori $4,$1,0x0000001b		
		0x00400098	0x24020004	addiu $2,$0,0x00000004	32: li $v0, 4 # print string syscall	
		0x0040009c	0x0000000c	syscall	33: syscall	
		0x004000a0	0x3c011001	lui $1,0x00001001	35: lw $a0, w1 # w1 is external (defined in program1.asm...	
		0x004000a4	0x8c24000c	lw $4,0x0000000c($1)		
		0x004000a8	0x24020001	addiu $2,$0,0x00000001	36: li $v0, 1 # print integer syscall	
		0x004000ac	0x0000000c	syscall	37: syscall	
		0x004000b0	0x3c011001	lui $1,0x00001001	39: la $a0, nl # end with newline	
		0x004000b4	0x34240031	ori $4,$1,0x00000031		programInProg.asm
		0x004000b8	0x24020004	addiu $2,$0,0x00000004	40: li $v0, 4 # print string syscall	
		0x004000bc	0x0000000c	syscall	41: syscall	
		0x004000c0	0x24080032	addiu $8,$0,0x00000032	46: li $t0, '2'	
		0x004000c4	0x3c011001	lui $1,0x00001001	47: sb $t0, nr	
		0x004000c8	0xa028002b	sb $8,0x0000002b($1)		
		0x004000cc	0x3c011001	lui $1,0x00001001	49: la $a0, valstr	
		0x004000d0	0x3424001b	ori $4,$1,0x0000001b		
		0x004000d4	0x24020004	addiu $2,$0,0x00000004	50: li $v0, 4 # print string syscall	
		0x004000d8	0x0000000c	syscall	51: syscall	
		0x004000dc	0x3c011001	lui $1,0x00001001	53: lw $a0, w2 # w2 is external (defined in program2.asm...	
		0x004000e0	0x8c240004	lw $4,0x00000004($1)		
		0x004000e4	0x24020001	addiu $2,$0,0x00000001	54: li $v0, 1 # print integer syscall	
		0x004000e8	0x0000000c	syscall	55: syscall	
		0x004000ec	0x3c011001	lui $1,0x00001001	57: la $a0, nl # end with newline	
		0x004000f0	0x34240031	ori $4,$1,0x00000031		
		0x004000f4	0x24020004	addiu $2,$0,0x00000004	58: li $v0, 4 # print string syscall	
		0x004000f8	0x0000000c	syscall	59: syscall	
		0x004000fc	0x03e00008	jr $31	62: jr $ra	

(linked) program execution



Translation and Startup, in pieces



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions (binary), stored on disk
- Provides **information** for building a **complete** program from the pieces (through “linking”)
  - **Header**: described contents/layout of object module
  - **Text segment**: translated instructions
  - **Static data segment**: data allocated for the life of the program
  - **Relocation info**: for contents that depend on absolute location of loaded program
  - **Symbol table**: global definitions and external refs
  - **Debug info**: for trace-ability to source code

Application Binary Interface

Interacting with the Operating System: ABI

SYSTEM V
APPLICATION BINARY INTERFACE

MIPS® RISC Processor Supplement
3rd Edition

http://math-atlas.sourceforge.net/devel/assembly/mipsabi32.pdf

Table of Contents

byReferenceVSbyValue.c

```
#
# “call by value” vs. “call by reference”
#

#include <stdio.h>          /* extern int fprintf(FILE *stream, const char *format, …) */


int f_by_value(int arg)
{
 int calc_res = 0;
 calc_res = arg+1;
 return calc_res;
}


void f_by_reference(int *arg_address)
{
 int calc_res = 0;
 calc_res = *arg_address+1;
 *arg_address = calc_res;
}
```

byReferenceVSbyValue.c

```
void main()
{
 int i = 10; int ret_val = 999;

 printf(" before f_by_value():     (i, ret_val) is  (%d, %d)\n", i, ret_val);

 /* call by value */
 ret_val = f_by_value(i);

 printf(" after  f_by_value():     (i, ret_val) is  (%d, %d)\n\n", i, ret_val);


 /* re-inialize values */
 i = 10;  ret_val = 999;


 printf(" before f_by_reference(): (i, ret_val) is  (%d, %d)\n", i, ret_val);

 /* call by reference */
 f_by_reference(&i);

 printf(" after  f_by_reference(): (i, ret_val) is  (%d, %d)\n", i, ret_val);
}
```

Executable and Linking Format (ELF)

http://refspecs.linuxbase.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html

```
% gcc -c byReferenceVSbyValue.c
% file byReferenceVSbyValue.o
byReferenceVSbyValue.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

Executable and Linking Format (ELF)

http://refspecs.linuxbase.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html

```
% gcc -c byReferenceVSbyValue.c
% file byReferenceVSbyValue.o
byReferenceVSbyValue.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

% hexdump -C -n 64 byReferenceVSbyValue.o
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  d0 05 00 00 00 00 00 00  |................|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 0d 00 0c 00  |....@.....@.....|
00000040
```

"magic number" = Special data located at the beginning of a binary data file to indicate its type to a utility. Under Unix, the system and various applications programs (especially the linker) distinguish between types of executable file by looking for a magic number.

From http://www.catb.org/jargon/html/go01.html

Executable and Linking Format (ELF)

http://refspecs.linuxbase.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html

```
% gcc -c byReferenceVSbyValue.c
% file byReferenceVSbyValue.o
byReferenceVSbyValue.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

% readelf -h byReferenceVSbyValue.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          800 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

Executable and Linkable Format (ELF)

```
% readelf -S -W byReferenceVSbyValue.o
There are 13 section headers, starting at offset 0x320:
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 0000d6 00  AX  0   0  4
  [ 2] .rela.text        RELA            0000000000000000 0007e0 0000c0 18     11   1  8
  [ 3] .data             PROGBITS        0000000000000000 000118 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          0000000000000000 000118 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        0000000000000000 000118 0000fa 00   A  0   0  8
  [ 6] .comment          PROGBITS        0000000000000000 000212 00002d 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        0000000000000000 00023f 000000 00      0   0  1
  [ 8] .eh_frame         PROGBITS        0000000000000000 000240 000078 00   A  0   0  8
  [ 9] .rela.eh_frame    RELA            0000000000000000 0008a0 000048 18     11   8  8
  [10] .shstrtab         STRTAB          0000000000000000 0002b8 000061 00      0   0  1
  [11] .symtab           SYMTAB          0000000000000000 000660 000138 18     12  11  8
  [12] .strtab           STRTAB          0000000000000000 000798 000042 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Executable and Linkable Format (ELF)

-g generates debugging information

```
% gcc -c -g byReferenceVSbyValue.c
% readelf -S -W byReferenceVSbyValue.o
There are 21 section headers, starting at offset 0x720:
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 0000d6 00  AX  0   0  4
  [ 2] .rela.text        RELA            0000000000000000 000e50 0000f0 18     19   1  8
  [ 3] .data             PROGBITS        0000000000000000 000118 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          0000000000000000 000118 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        0000000000000000 000118 0000fa 00   A  0   0  8
  [ 6] .debug_info       PROGBITS        0000000000000000 000212 000126 00      0   0  1
  [ 7] .rela.debug_info  RELA            0000000000000000 000f40 0002b8 18     19   6  8
  [ 8] .debug_abbrev     PROGBITS        0000000000000000 000338 0000bb 00      0   0  1
  [ 9] .debug_aranges    PROGBITS        0000000000000000 0003f3 000030 00      0   0  1
  [10] .rela.debug_aranges RELA          0000000000000000 0011f8 000030 18     19   9  8
  [11] .debug_line       PROGBITS        0000000000000000 000423 00005e 00      0   0  1
  [12] .rela.debug_line  RELA            0000000000000000 001228 000018 18     19  11  8
  [13] .debug_str        PROGBITS        0000000000000000 000481 000144 01  MS  0   0  1
  [14] .comment          PROGBITS        0000000000000000 0005c5 00002d 01  MS  0   0  1
  [15] .note.GNU-stack   PROGBITS        0000000000000000 0005f2 000000 00      0   0  1
  [16] .eh_frame         PROGBITS        0000000000000000 0005f8 000078 00   A  0   0  8
  [17] .rela.eh_frame    RELA            0000000000000000 001240 000048 18     19  16  8
  [18] .shstrtab         STRTAB          0000000000000000 000670 0000b0 00      0   0  1
  [19] .symtab           SYMTAB          0000000000000000 000c60 0001b0 18     20  14  8
  [20] .strtab           STRTAB          0000000000000000 000e10 00003c 00      0   0  1
```

Executable and Linkable Format (ELF)

```
% gcc -c byReferenceVSbyValue.c
% readelf -s byReferenceVSbyValue.o

Symbol table '.symtab' contains 13 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS byReferenceVSbyValue.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000    28 FUNC    GLOBAL DEFAULT    1 f_by_value
    10: 000000000000001c    31 FUNC    GLOBAL DEFAULT    1 f_by_reference
    11: 000000000000003b   155 FUNC    GLOBAL DEFAULT    1 main
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

Executable and Linkable Format (ELF)

In `byReferenceVSbyValue_local.c`: use **`static`** to make functions `f_by_value` and `f_by_reference` **invisible globally (to the linker)**, *i.e.,* **local** to this binary object file.

```
static int f_by_value(int arg)
static void f_by_reference(int *arg_address)
```

```
% readelf -s byReferenceVSbyValue_local.o

Symbol table '.symtab' contains 13 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS byReferenceVSbyValue_local.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000    28 FUNC    LOCAL  DEFAULT    1 f_by_value
     6: 000000000000001c    31 FUNC    LOCAL  DEFAULT    1 f_by_reference
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    11: 000000000000003b   155 FUNC    GLOBAL DEFAULT    1 main
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

Executable and Linkable Format (ELF)

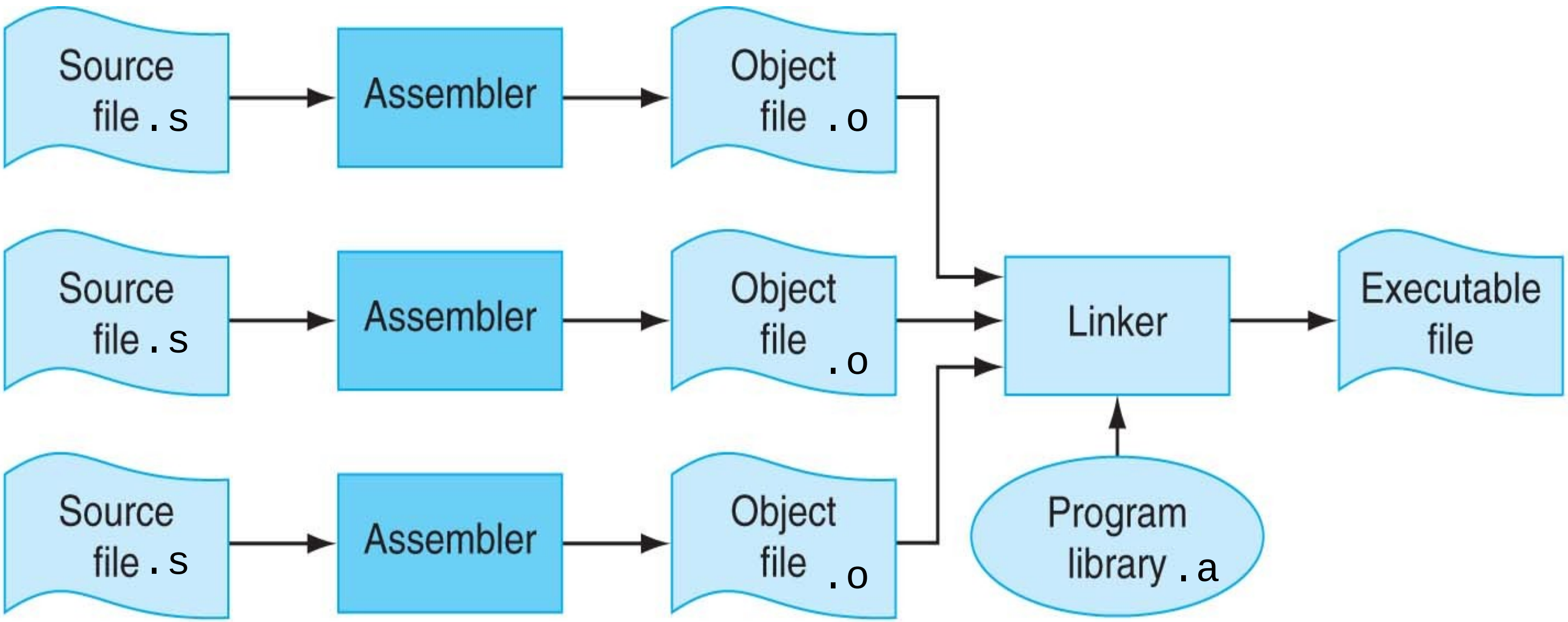
In `byReferenceVSbyValue_local_global.c`:

additionally, use `static` to make **local function variable** `calc_res` in `f_by_reference` **global** (from the point of view of `f_by_reference` but **local** to the linker).

```
 static void f_by_reference(int *arg_address){
   static int calc_res;
   calc_res = *arg_address+1;
   *arg_address = calc_res;
 }
```

```
% readelf -s byReferenceVSbyValue_local_global.o
Symbol table '.symtab' contains 14 entries:
Num:         Value       Size Type     Bind   Vis      Ndx Name
  0: 0000000000000000     0 NOTYPE   LOCAL  DEFAULT UND
  1: 0000000000000000     0 FILE     LOCAL  DEFAULT ABS byReferenceVSbyValue.loca
  2: 0000000000000000     0 SECTION  LOCAL  DEFAULT   1
  3: 0000000000000000     0 SECTION  LOCAL  DEFAULT   3
  4: 0000000000000000     0 SECTION  LOCAL  DEFAULT   4
  5: 0000000000000000    28 FUNC     LOCAL  DEFAULT   1 f_by_value
  6: 000000000000001c    38 FUNC     LOCAL  DEFAULT   1 f_by_reference
  7: 0000000000000000     4 OBJECT   LOCAL  DEFAULT   4 calc_res.2254
  8: 0000000000000000     0 SECTION  LOCAL  DEFAULT   5
  9: 0000000000000000     0 SECTION  LOCAL  DEFAULT   7
 10: 0000000000000000     0 SECTION  LOCAL  DEFAULT   8
 11: 0000000000000000     0 SECTION  LOCAL  DEFAULT   6
 12: 0000000000000042   156 FUNC     GLOBAL DEFAULT   1 main
 13: 0000000000000000     0 NOTYPE   GLOBAL DEFAULT UND printf
```

Linking Object Files



Statically Linking Object Files

Statically link the object file `byReferenceVSbyValue.o`

with all necessary libraries (e.g., resolve `printf`) to make it executable.

```
% gcc -o byRefenceVSbyValue byReferenceVSbyValue.o
% readelf -h byReferenceVSbyValue
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4003e0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          3272 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

Statically Linking Object Files

Split up `byReferenceVSbyValue.c` into:

byReferenceVSbyValue_functions.h

```
#include <stdio.h>
extern int f_by_value(int arg);
extern void f_by_reference(int *arg_address);
```

declare (signature) vs. **define/implement**

byReferenceVSbyValue_functions.c

```
#include "byReferenceVSbyValue_functions.h"

int f_by_value(int arg)
{
 int calc_res = 0;
 calc_res = arg+1;
 return calc_res;
}

void f_by_reference(int *arg_address)
{
 int calc_res;
 calc_res = *arg_address+1;
 *arg_address = calc_res;
}
```

Statically Linking Object Files

byReferenceVSbyValue_main.c

```
#include "byReferenceVSbyValue_functions.h"
void main()
{
 int i;
 int return_value;
 /* inialize values */
 i = 10;
 return_value = 999;
 printf(" before f_by_value():     (i, return_value) is  (%d, %d)\n", i, return_value);
 /* call by value */
 return_value = f_by_value(i);
 printf(" after  f_by_value():     (i, return_value) is  (%d, %d)\n\n", i, return_value);
 /* re-inialize values */
 i = 10;
 return_value = 999;
 printf(" before f_by_reference(): (i, return_value) is  (%d, %d)\n", i, return_value);
 /* call by reference */
 f_by_reference(&i);
 printf(" after  f_by_reference(): (i, return_value) is  (%d, %d)\n", i, return_value);
}
```

Statically Linking Object Files

```
% gcc -c byReferenceVSbyValue_functions.c

% readelf -s -W byReferenceVSbyValue_functions.o

Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS byReferenceVSbyValue_functions.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     8: 0000000000000000    28 FUNC    GLOBAL DEFAULT    1 f_by_value
     9: 000000000000001c    31 FUNC    GLOBAL DEFAULT    1 f_by_reference
```

Statically Linking Object Files

```
% gcc -c byReferenceVSbyValue_main.c

% readelf -s -W byReferenceVSbyValue_main.o

Symbol table '.symtab' contains 13 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS byReferenceVSbyValue_main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000   155 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND f_by_value
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND f_by_reference
```

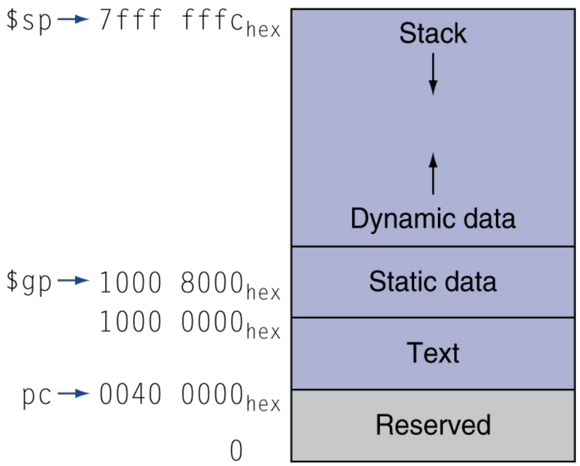
Statically Linking Object Files

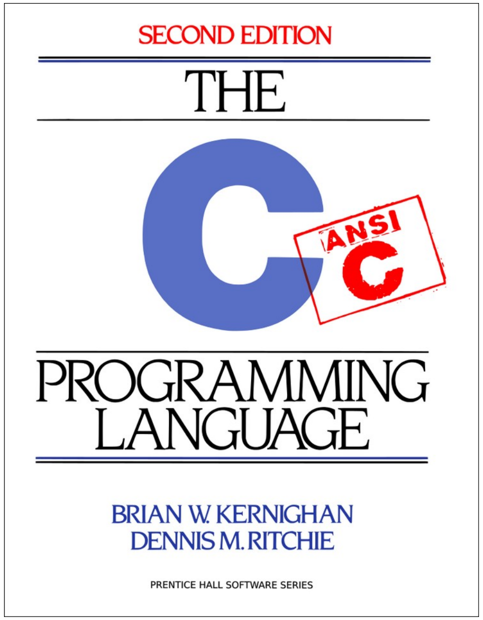
```
% gcc -o byReferenceVSbyValue byReferenceVSbyValue_functions.o byReferenceVSbyValue_main.o

% readelf -s -W byReferenceVSbyValue

Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 68 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400200     0 SECTION LOCAL  DEFAULT    1
...
    49: 00000000004004dc    28 FUNC    GLOBAL DEFAULT   13 f_by_value
...
    52: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    53: 00000000004004f8    31 FUNC    GLOBAL DEFAULT   13 f_by_reference
...
    63: 0000000000400518   155 FUNC    GLOBAL DEFAULT   13 main
...
```

Visibility and Bindings in C

```
/* bindings.c *
#include <stdio.h>
#include <stdlib.h> /* for malloc() */
int global = -1; /* visible for external linking */
extern int fromOtherCompilationUnit;
static int global2thisFile; /* not visible to linker, initialized to 0 */
int *f(int argByValue, int *argByReference){
  /* similar to local vars: int argByValue, int *argByReference; */
  int local = 5; int *array; static int scopeLife = 2;
  *argByReference = local + argByValue * (*argByReference) +
                    global + global2thisFile + fromOtherCompilationUnit;
  scopeLife++; printf("%d\n", scopeLife);
  array = malloc(10 * sizeof(int)); /* dynamically allocated */
  /* to be relinquished (not cleared) with "free(array);" if not: "memory leak" */
  for (int index=0; index<10; index++) {array[index] = index;}
  return array; /* “return &local;” yields nonsense, compiler warning */
}
void main(){
  int *tenIntsRef; fromOtherCompilationUnit = 9;
  tenIntsRef = f(global2thisFile, &fromOtherCompilationUnit);
  printf("%d\n", tenIntsRef[5]); /* free(tenIntsRef); memory leak! */
  tenIntsRef = f(global2thisFile, &fromOtherCompilationUnit);
  printf("%d\n", tenIntsRef[5]); free(tenIntsRef);
}

/* otherCompilationUnit.c */
int fromOtherCompilationUnit = 5;
```

```
% gcc -o bindings bindings.c otherCompilationUnit.c
% ./bindings
3
5
4
5
```





Intermezzo: C++ and symbol tables

```
//
// overloading.cpp
//

#include <iostream>
using namespace std;

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}

void print(char* str) {
  cout << " Here is char* \"" << str << "\"" << endl;
}

int main() {
  print(10);
  print(10.10);
  print((char *)"ten"); // ISO C++ forbids converting a string constant to char*, so explicitly type-cast
}
```

```
% ./overloading
 Here is int 10
 Here is float 10.1
 Here is char* "ten"
```

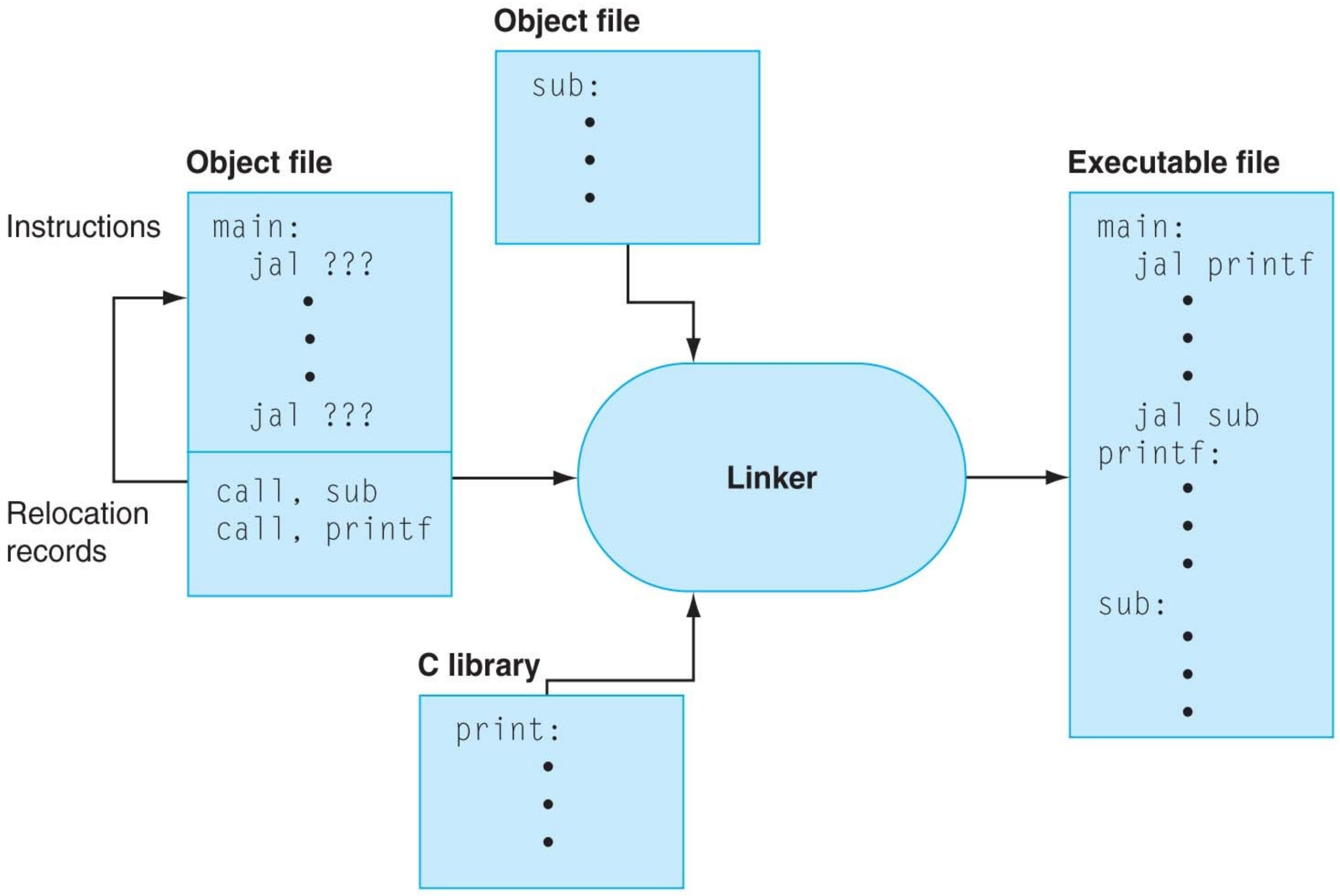
C++ compiler “mangles” names

```
% g++ -c overloading.cpp

% readelf -s -W overloading.o
Symbol table '.symtab' contains 27 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS overloading.cpp
      ...
    13: 0000000000000000    54 FUNC    GLOBAL DEFAULT    1 _Z5printi
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZSt4cout
    15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSolsEi
    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
    18: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSolsEPFRSoS_E
    19: 0000000000000036    67 FUNC    GLOBAL DEFAULT    1 _Z5printd
    20: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSolsEd
    21: 0000000000000079    57 FUNC    GLOBAL DEFAULT    1 _Z5printPc
    22: 00000000000000b2    59 FUNC    GLOBAL DEFAULT    1 main
    23: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSt8ios_base4InitC1Ev
    24: 0000000000000000     0 NOTYPE  GLOBAL HIDDEN   UND __dso_handle
    25: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSt8ios_base4InitD1Ev
    26: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND __cxa_atexit
```

Names of overloaded functions are made unique for the C-convention linker … beware … **mangling is compiler-specific!**

Solution: apart from looking into symbol tables, and only when C++ source code is available: use `extern “C” int f (int)`

Linking Object Modules (Editor)



(static) Linking of Object Modules

- Independently compiled/assembled procedures

  (no need to **re-**compile/assemble everything)
- Produces an **executable image**
  1. **Merges** text/data segments
  2. **Resolves** text/data labels
     (determines their addresses)
  3. **Patches** *location-dependent* and *external* refs

  “edit”
- Could leave location dependencies for fixing at run-time by a *relocating* **loader**
  - But with *virtual memory*, no need to do this,
  - program can be loaded (faulted) into *absolute* location in *virtual* memory space

  not in CSA, but in Operating Systems (Ba2)

Linking Object Modules (Editor)

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	sw $a1, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Linking Object Modules (Editor)

result of static linking:

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal 40 0100_{hex}
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal 40 0000_{hex}
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

Label	Bkpt	Address	Code	Basic	Source	Annotation
defaultEntry:		0x00400000	0x24040000	addiu $4,$0,0x00000000	20: li $a0, 0	
		0x00400004	0x0c10001a	jal 0x00400068	21: jal printProgX	
main:		0x00400008	0x0c100010	jal 0x00400040	26: jal prog1 # call prog1	main.asm
		0x0040000c	0x0c100006	jal 0x00400018	27: jal prog2 # call prog2	
		0x00400010	0x2402000a	addiu $2,$0,0x0000000a	29: <3> li $v0, 10	cleanProgramExit
		0x00400014	0x0000000c	syscall	<4> syscall # exit()	
prog2:		0x00400018	0x3c010000	lui $1,0x00000000	20: <8> subiu $sp, $sp, 4 # stack (of words) grows downwards	
		0x0040001c	0x34210004	ori $1,$1,0x00000004		push
		0x00400020	0x03a1e823	subu $29,$29,$1		
		0x00400024	0xafbf0000	sw $31,0x00000000($29)	<9> sw $ra, 0($sp) # $sp points to last stack entry	
		0x00400028	0x3c011001	lui $1,0x00001001	23: lw $a0, progNr	
		0x0040002c	0x8c240000	lw $4,0x00000000($1)		program2.asm
		0x00400030	0x0c10001a	jal 0x00400068	24: jal printProgX	
		0x00400034	0x8fbf0000	lw $31,0x00000000($29)	27: <13> lw $ra, 0($sp) # $sp points to last stack entry	
		0x00400038	0x27bd0004	addiu $29,$29,0x000...	<14> addiu $sp, $sp, 4 # stack (of words) grows downwards	pop
		0x0040003c	0x03e00008	jr $31	30: jr $ra	
prog1:		0x00400040	0x3c010000	lui $1,0x00000000	20: <8> subiu $sp, $sp, 4 # stack (of words) grows downwards	
		0x00400044	0x34210004	ori $1,$1,0x00000004		
		0x00400048	0x03a1e823	subu $29,$29,$1		
		0x0040004c	0xafbf0000	sw $31,0x00000000($29)	<9> sw $ra, 0($sp) # $sp points to last stack entry	
		0x00400050	0x3c011001	lui $1,0x00001001	23: lw $a0, progNr	program1.asm
		0x00400054	0x8c240008	lw $4,0x00000008($1)		
		0x00400058	0x0c10001a	jal 0x00400068	24: jal printProgX	
		0x0040005c	0x8fbf0000	lw $31,0x00000000($29)	27: <13> lw $ra, 0($sp) # $sp points to last stack entry	
		0x00400060	0x27bd0004	addiu $29,$29,0x000...	<14> addiu $sp, $sp, 4 # stack (of words) grows downwards	
		0x00400064	0x03e00008	jr $31	30: jr $ra	
printProgX:		0x00400068	0x24890030	addiu $9,$4,0x00000030	18: addiu $t1, $a0, '0' # ascii value of the digit = integer ...	
		0x0040006c	0x3c011001	lui $1,0x00001001	19: sb $t1, prognr	
		0x00400070	0xa0290017	sb $9,0x00000017($1)		
		0x00400074	0x3c011001	lui $1,0x00001001	21: la $a0, inProgX	
		0x00400078	0x34240010	ori $4,$1,0x00000010		
		0x0040007c	0x24020004	addiu $2,$0,0x00000004	22: li $v0, 4 # print string syscall	
		0x00400080	0x0000000c	syscall	23: syscall	
		0x00400084	0x24080031	addiu $8,$0,0x00000031	28: li $t0, '1'	
		0x00400088	0x3c011001	lui $1,0x00001001	29: sb $t0, nr	
		0x0040008c	0xa028002b	sb $8,0x0000002b($1)		
		0x00400090	0x3c011001	lui $1,0x00001001	31: la $a0, valstr	
		0x00400094	0x3424001b	ori $4,$1,0x0000001b		
		0x00400098	0x24020004	addiu $2,$0,0x00000004	32: li $v0, 4 # print string syscall	
		0x0040009c	0x0000000c	syscall	33: syscall	
		0x004000a0	0x3c011001	lui $1,0x00001001	35: lw $a0, w1 # w1 is external (defined in program1.asm...	
		0x004000a4	0x8c24000c	lw $4,0x0000000c($1)		
		0x004000a8	0x24020001	addiu $2,$0,0x00000001	36: li $v0, 1 # print integer syscall	
		0x004000ac	0x0000000c	syscall	37: syscall	
		0x004000b0	0x3c011001	lui $1,0x00001001	39: la $a0, nl # end with newline	
		0x004000b4	0x34240031	ori $4,$1,0x00000031		programInProg.asm
		0x004000b8	0x24020004	addiu $2,$0,0x00000004	40: li $v0, 4 # print string syscall	
		0x004000bc	0x0000000c	syscall	41: syscall	
		0x004000c0	0x24080032	addiu $8,$0,0x00000032	46: li $t0, '2'	
		0x004000c4	0x3c011001	lui $1,0x00001001	47: sb $t0, nr	
		0x004000c8	0xa028002b	sb $8,0x0000002b($1)		
		0x004000cc	0x3c011001	lui $1,0x00001001	49: la $a0, valstr	
		0x004000d0	0x3424001b	ori $4,$1,0x0000001b		
		0x004000d4	0x24020004	addiu $2,$0,0x00000004	50: li $v0, 4 # print string syscall	
		0x004000d8	0x0000000c	syscall	51: syscall	
		0x004000dc	0x3c011001	lui $1,0x00001001	53: lw $a0, w2 # w2 is external (defined in program2.asm...	
		0x004000e0	0x8c240004	lw $4,0x00000004($1)		
		0x004000e4	0x24020001	addiu $2,$0,0x00000001	54: li $v0, 1 # print integer syscall	
		0x004000e8	0x0000000c	syscall	55: syscall	
		0x004000ec	0x3c011001	lui $1,0x00001001	57: la $a0, nl # end with newline	
		0x004000f0	0x34240031	ori $4,$1,0x00000031		
		0x004000f4	0x24020004	addiu $2,$0,0x00000004	58: li $v0, 4 # print string syscall	
		0x004000f8	0x0000000c	syscall	59: syscall	
		0x004000fc	0x03e00008	jr $31	62: jr $ra	

Mars Messages | Run I/O

Clear

```
Assemble: assembling /home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/main.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/program2.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/program1.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/macros.asm,
/home/hv/src/courses/ComputerSystemsArchitecture/material/Handouts/Assembly/Class 19 Compiler/linking/printInProg.asm

Assemble: operation completed successfully.
```

Labels

Label ▲	Address
(global)	
main	0x00400008
printProgX	0x00400068
prog1	0x00400040
prog2	0x00400018
w1	0x1001000c
w2	0x10010004
main.asm	
defaultEntry	0x00400000
program2.asm	
progNr	0x10010000
program1.asm	
progNr	0x10010008
printInProg.asm	
inProgX	0x10010010
nl	0x10010031
nr	0x1001002b
prognr	0x10010017
valstr	0x1001001b

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000002	0xffffffff	0x00000001	0x000000ff	0x70206e49	0x00676f72	0x20000a3a	0x65685420
0x10010020	0x6c617620	0x6f206575	0x00772066	0x20736920	0x00000a00	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	\0 \0 \0 .	. . . .	\0 \0 \0 .	\0 \0 \0 .	p n I	\0 g o r	\0 \n :	e h T
0x10010020	l a v	o e u	\0 w f	s i	\0 \0 \n \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☑ ASCII

Relocation and PIC

- **Relocatable** Code:
  > requires fixing/editing of address references

- **Position Independent Code** (PIC):
  > does not require fixing/editing
  > use *relative* addressing only

```
% gcc -fPIC -c byReferenceVSbyValue_functions.c
```

Re-entrant Code

Can be interrupted (see “exceptions”) in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution

Requirements:

- Should not hold any static (or global) non-constant data (on stack is OK)
- Should not modify its own code
- Should not call non-re-entrant routines

Loading a Program

- Load **program** from **image** file on **disk** into **memory**
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - or set page table entries so they can be faulted in (cfr. virtual memory)
  4. Set up arguments on stack
  5. Initialize registers (including `$sp, $fp, $gp`)
  6. Jump to startup routine
     - Copies arguments to `$a0,` … and calls `main`
     - When `main` returns, do `exit` syscall

Intermezzo: passing command-line args

https://gcc.gnu.org/onlinedocs/gcc/Standards.html

```
/* argcArgv.c
 * compile with:  gcc -std=c99 -o argcArgv argcArgv.c
 * call: `pwd`/argcArgv -a b -c - d e */
#include <stdio.h>
int main(int argc, char **argv){
  printf("number of arguments, including command name = %d\n", argc);

  /* vector style */
  for (int arg_nr = 0; arg_nr < argc; arg_nr++)
    printf("argument %d = %s\n", arg_nr, argv[arg_nr]);

  /* pointer style */
  for (int arg_nr = 0; arg_nr < argc; arg_nr++){
    char *arg_str = *argv;
    printf("argument %d = %s\n", arg_nr, arg_str);
    argv++;
  }
}
```

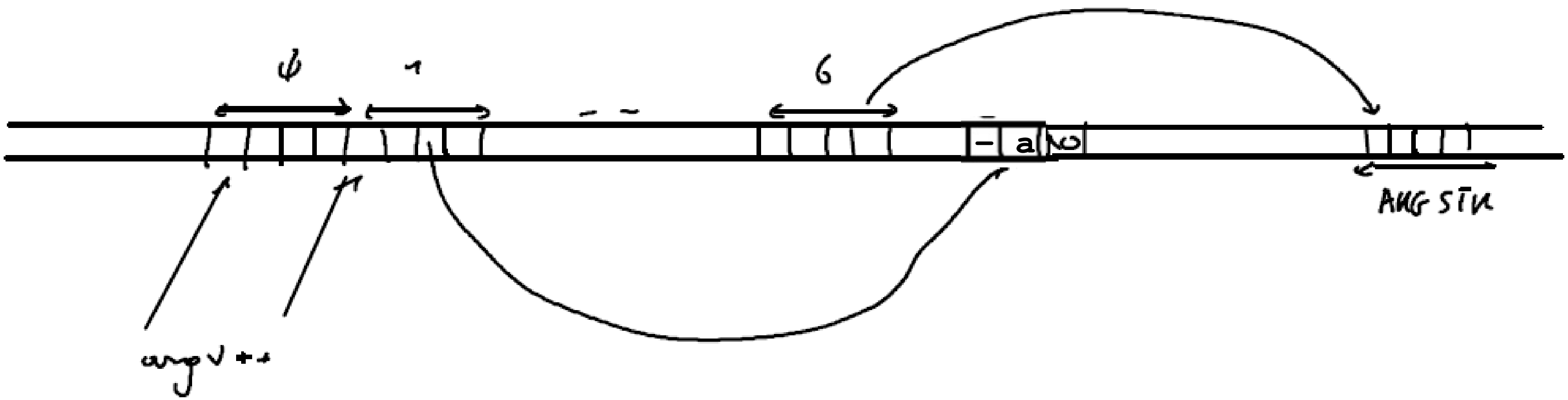
Intermezzo: passing command-line args

```
% `pwd`/argcArgv -a b -c -- d e
number of arguments, including command name = 7
argument 0 =
/home/hv/src/courses/ComputerSystemsArchitecture/lectures/argcArgv
argument 1 = -a
argument 2 = b
argument 3 = -c
argument 4 = --
argument 5 = d
argument 6 = e
argument 0 =
/home/hv/src/courses/ComputerSystemsArchitecture/lectures/argcArgv
argument 1 = -a
argument 2 = b
argument 3 = -c
argument 4 = --
argument 5 = d
argument 6 = e
```

Intermezzo: passing command-line args

```
/* argcArgv.c
 * compile with:  gcc -std=c99 -o argcArgv argcArgv.c
 * call: `pwd`/argcArgv -a b -c -- d e */
```

https://gcc.gnu.org/onlinedocs/gcc/Standards.html



Dynamic Linking and Loading

Only **link/load** library procedure **when** it is **called**
aka “lazy” linking

- Avoids image **bloat** caused by static linking of **all** (transitively) referenced libraries (but not necessarily used in all execution paths)
- Automatically picks up **new** (latest) library **versions** (no need to re-link)

  http://www.iecc.com/linker/linker10.html

- Requires loaded code to be **relocatable**

Shared Object (`.so`) / Dynamic-Link Library (`.dll`)

```
% gcc -fPIC -c byReferenceVSbyValue_functions.c
% gcc -shared -o libbyReferenceVSbyValue_functions.so byReferenceVSbyValue_functions.o
% readelf -h libbyReferenceVSbyValue_functions.so
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
...
  Type:                                  DYN (Shared object file)
...
```

Shared Object (.so) / Dynamic-Link Library (.dll)

```
% gcc -c byReferenceVSbyValue_main.c

% gcc -o byReferenceVSbyValue byReferenceVSbyValue_main.o -L. -
lbyReferenceVSbyValue_functions

% readelf -s byReferenceVSbyValue

Symbol table '.dynsym' contains 14 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND f_by_value
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND f_by_reference
     …

% ./byReferenceVSbyValue
./byReferenceVSbyValue: error while loading shared libraries:
libbyReferenceVSbyValue_functions.so: cannot open shared object file: No such file or
directory

% setenv LD_LIBRARY_PATH .
% ./byReferenceVSbyValue
```

Static Linkage (of `.o`)

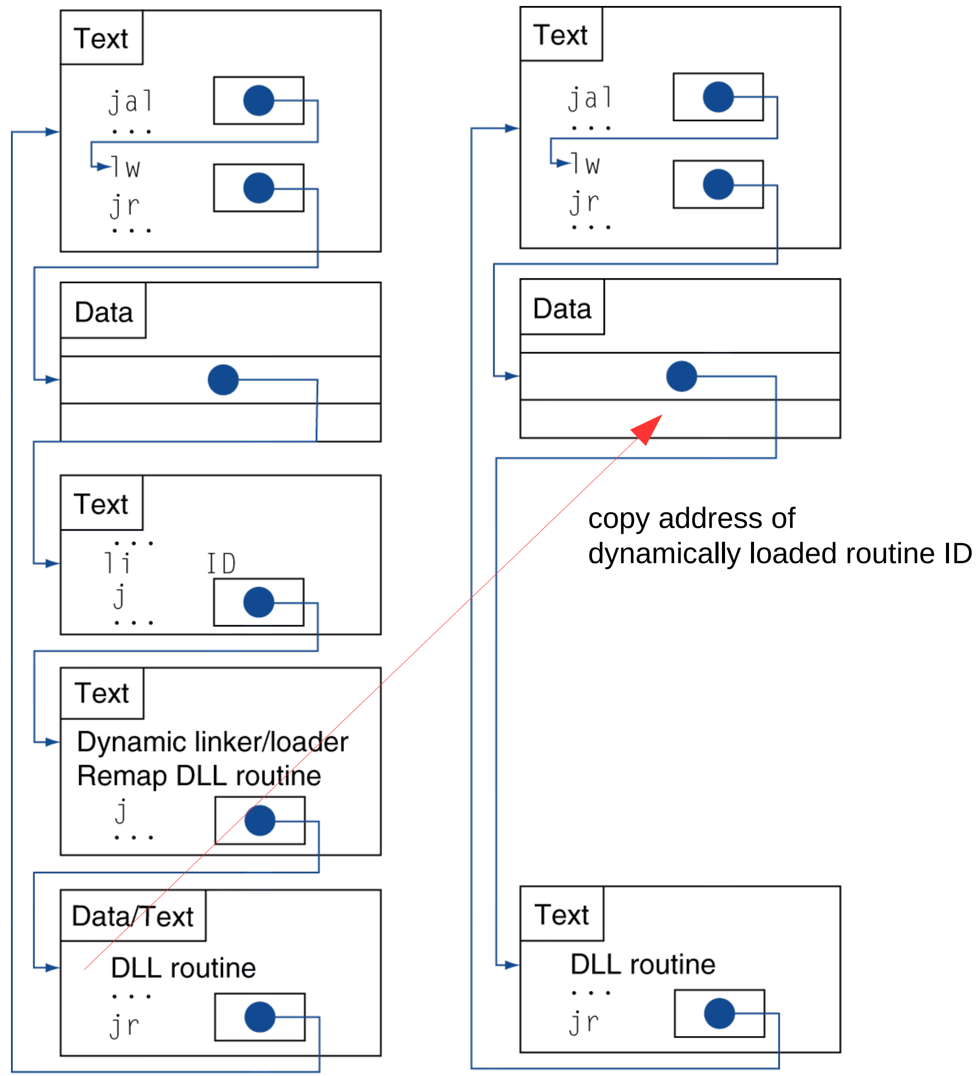
Statically linked object

Lazy Linkage (of `.dll/.so`)

Indirection table:
Upon first call to routine ID:
  holds **address of dynamic linker loader**
Upon subsequent calls to routine ID:
  holds **address of dynamically loaded ID**

Find (in search path) a `dll/so` object file which provides a routine ID;
Dynamic linker/loader **loads** the found object file and **copies** its address to the indirection table;
Jump to the loaded object file.

Dynamically loaded object



a. First call to DLL routine

b. Subsequent calls to DLL routine

Calling C from Python

ctypes — A foreign function library for Python

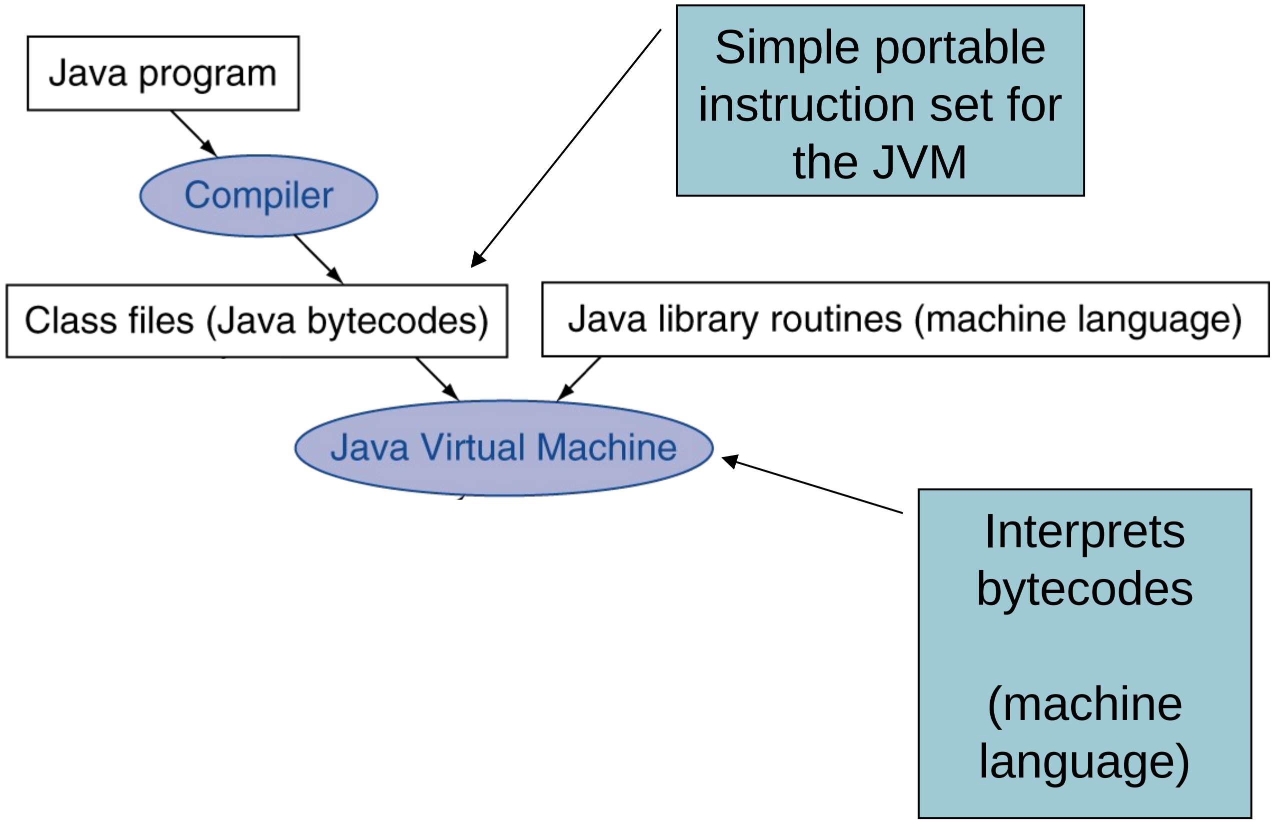
ctypes is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

```
>>> from ctypes import *
>>> dir()
['ARRAY', 'ArgumentError', 'Array', 'BigEndianStructure', 'BigEndianUnion', 'CDLL',
'CFUNCTYPE', 'DEFAULT_MODE', 'LibraryLoader', 'LittleEndianStructure',
'LittleEndianUnion', 'POINTER', 'PYFUNCTYPE', 'PyDLL', 'RTLD_GLOBAL', 'RTLD_LOCAL',
'SIZEOF_TIME_T', 'SetPointerType', 'Structure', 'Union', '__annotations__',
'__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'addressof', 'alignment', 'byref', 'c_bool', 'c_buffer', 'c_byte', 'c_char',
'c_char_p', 'c_double', 'c_float', 'c_int', 'c_int16', 'c_int32', 'c_int64',
'c_int8', 'c_long', 'c_longdouble', 'c_longlong', 'c_short', 'c_size_t',
'c_ssize_t', 'c_time_t', 'c_ubyte', 'c_uint', 'c_uint16', 'c_uint32', 'c_uint64',
'c_uint8', 'c_ulong', 'c_ulonglong', 'c_ushort', 'c_void_p', 'c_voidp', 'c_wchar',
'c_wchar_p', 'cast', 'cdll', 'create_string_buffer', 'create_unicode_buffer',
'get_errno', 'memmove', 'memset', 'pointer', 'py_object', 'pydll', 'pythonapi',
'resize', 'set_errno', 'sizeof', 'string_at', 'wstring_at']
>>> c_char()
c_char(b'\x00')

>>> libc=CDLL("libc.so.6")
>>> libc.time(None)
1701201608
```

https://docs.python.org/3/library/ctypes.html

Interpreted Languages (e.g., Java)

compile to **bytecode** once, **interpret Virtual Machine bytecode**



Java Virtual Machine (JVM)

```
do
    atomically calculate PC and fetch opcode at PC;
    if (operands) fetch operands;
    execute the action for the opcode;
while (there is more to do);
```

https://docs.oracle.com/javase/specs/jvms/se12/html/index.html

https://www.informit.com/articles/article.aspx?p=2024315

bytecode (JVM instruction set)

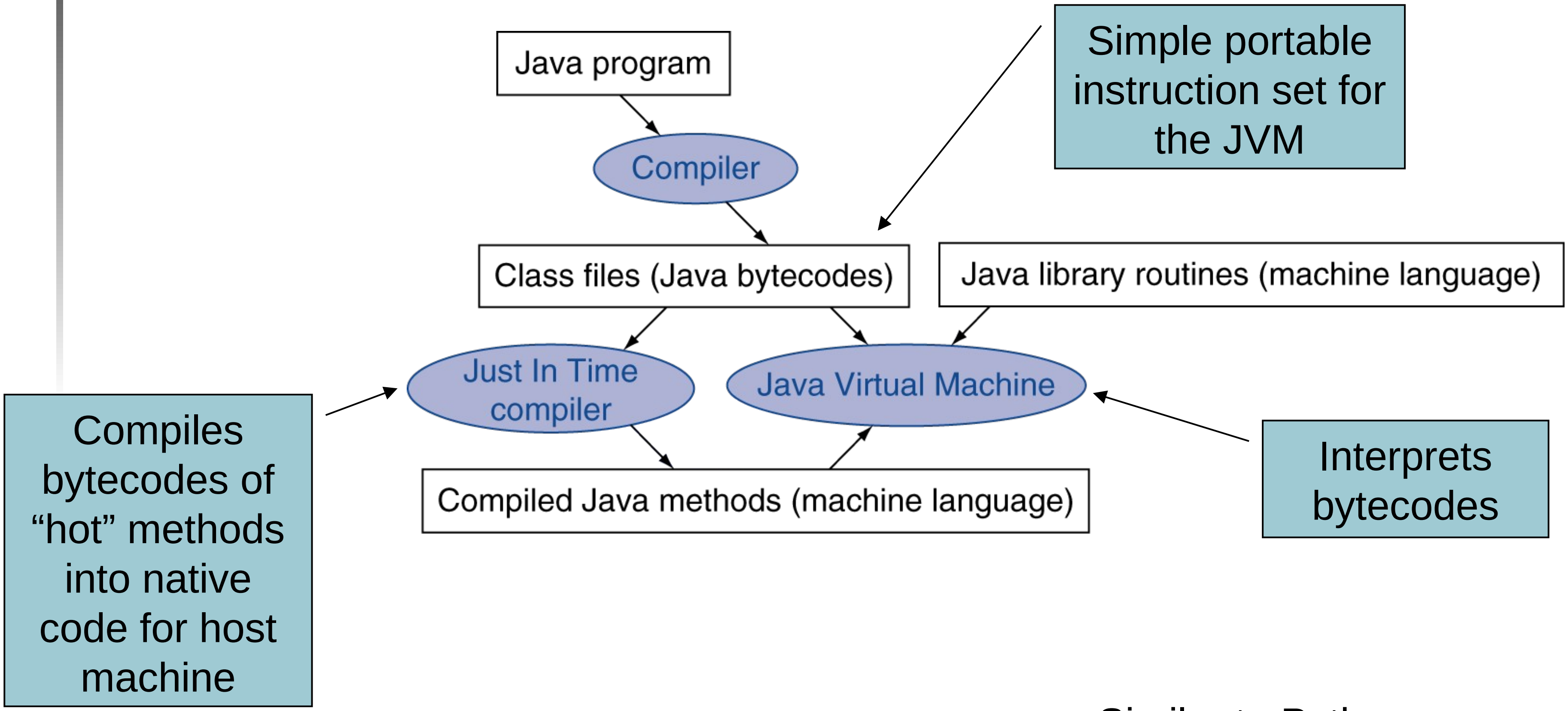
Mnemonic	Opcode (in hex)	Opcode (in binary)	Other bytes [count]: [operand labels]	Stack [before]→[after]	Description
aaload	32	0011 0010		arrayref, index → value	load onto the stack a reference from an array
aastore	53	0101 0011		arrayref, index, value →	store a reference in an array
aconst_null	01	0000 0001		→ null	push a *null* reference onto the stack
aload	19	0001 1001	1: index	→ objectref	load a reference onto the stack from a local variable *#index*
aload_0	2a	0010 1010		→ objectref	load a reference onto the stack from local variable 0
aload_1	2b	0010 1011		→ objectref	load a reference onto the stack from local variable 1
aload_2	2c	0010 1100		→ objectref	load a reference onto the stack from local variable 2
aload_3	2d	0010 1101		→ objectref	load a reference onto the stack from local variable 3
anewarray	bd	1011 1101	2: indexbyte1, indexbyte2	count → arrayref	create a new array of references of length *count* and component type identified by the class reference *index* (`indexbyte1 << 8 | indexbyte2`) in the constant pool
areturn	b0	1011 0000		objectref → [empty]	return a reference from a method
arraylength	be	1011 1110		arrayref → length	get the length of an array
astore	3a	0011 1010	1: index	objectref →	store a reference into a local variable *#index*
astore_0	4b	0100 1011		objectref →	store a reference into local variable 0

https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions

https://www.informit.com/articles/article.aspx?p=2024315&seqNum=11

Just In Time compilation (JIT)

compile to **bytecode** once, **interpret** Virtual Machine bytecode (**or JIT compile**)



Similar to Python
(PyPy is JIT)

C Sort Example (performance analysis)

- Illustrates use of assembly instructions for a `C` bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - `v` in `$a0`, `k` in `$a1`, `temp` in `$t0`

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls `swap`)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - `v` in `$a0`, `k` in `$a1`, `i` in `$s0`, `j` in `$s1`

The Procedure Body

```
         move $s2, $a0             # save $a0 into $s2
         move $s3, $a1             # save $a1 into $s3
```
Move params

```
         move $s0, $zero           # i = 0
for1tst: slt  $t0, $s0, $s3        # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
```
Outer loop

```
         beq  $t0, $zero, exit1    # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, –1         # j = i – 1
for2tst: slti $t0, $s1, 0          # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2    # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2          # $t1 = j * 4
         add  $t2, $s2, $t1        # $t2 = v + (j * 4)
         lw   $t3, 0($t2)          # $t3 = v[j]
         lw   $t4, 4($t2)          # $t4 = v[j + 1]
         slt  $t0, $t4, $t3        # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2    # go to exit2 if $t4 ≥ $t3
```
Inner loop

```
         move $a0, $s2             # 1st param of swap is v (old $a0)
         move $a1, $s1             # 2nd param of swap is j
         jal  swap                 # call swap procedure
```
Pass params & call

```
         addi $s1, $s1, –1         # j –= 1
         j    for2tst              # jump to test of inner loop
```
Inner loop

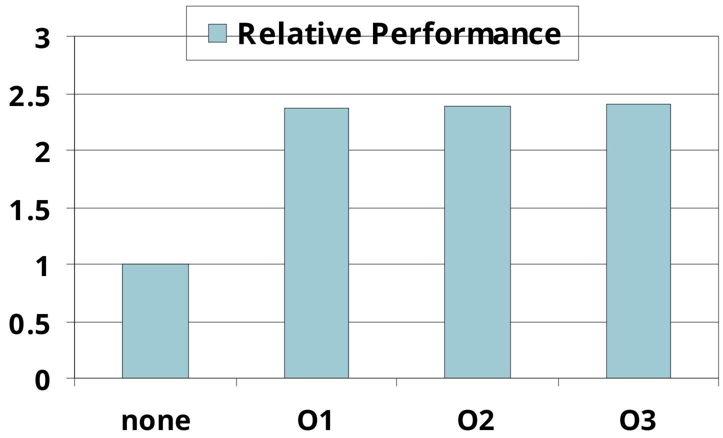
```
exit2:   addi $s0, $s0, 1          # i += 1
         j    for1tst              # jump to test of outer loop
```
Outer loop

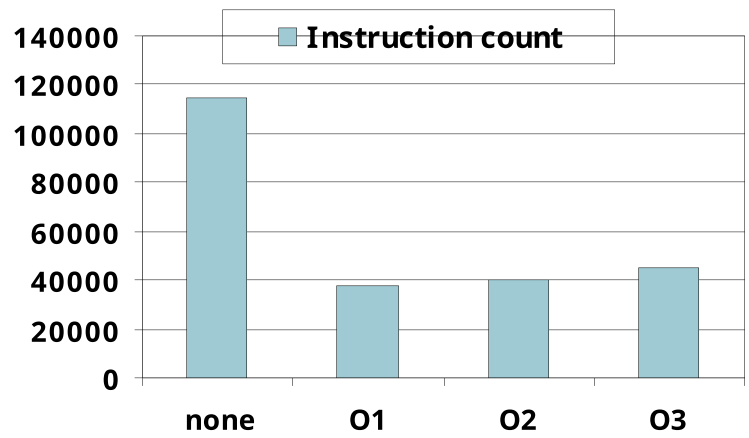
The Full Procedure

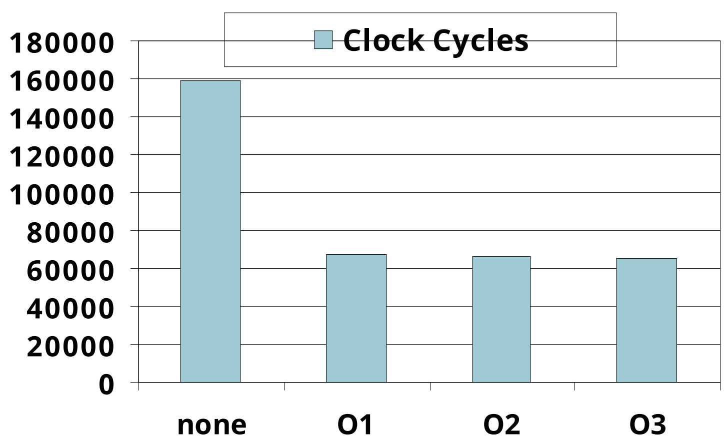
```
sort:    addi $sp,$sp, –20       # make room on stack for 5 registers
         sw $ra, 16($sp)         # save $ra on stack
         sw $s3,12($sp)          # save $s3 on stack
         sw $s2, 8($sp)          # save $s2 on stack
         sw $s1, 4($sp)          # save $s1 on stack
         sw $s0, 0($sp)          # save $s0 on stack
         …                       # procedure body
         …
exit1:   lw $s0, 0($sp)          # restore $s0 from stack
         lw $s1, 4($sp)          # restore $s1 from stack
         lw $s2, 8($sp)          # restore $s2 from stack
         lw $s3,12($sp)          # restore $s3 from stack
         lw $ra,16($sp)          # restore $ra from stack
         addi $sp,$sp, 20        # restore stack pointer
         jr $ra                  # return to calling routine
```

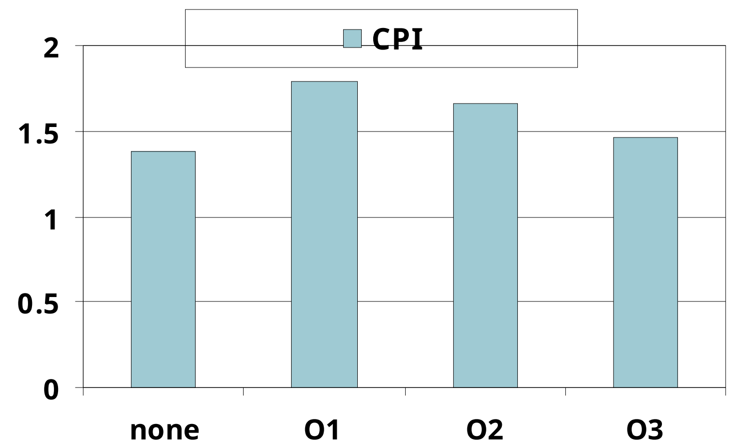
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

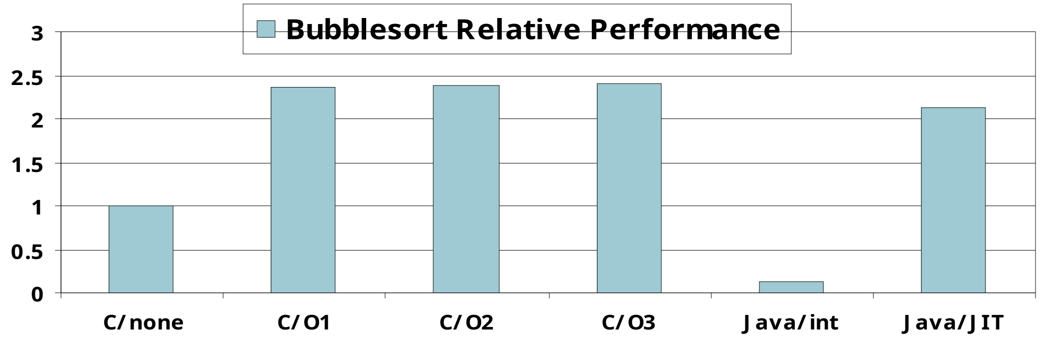


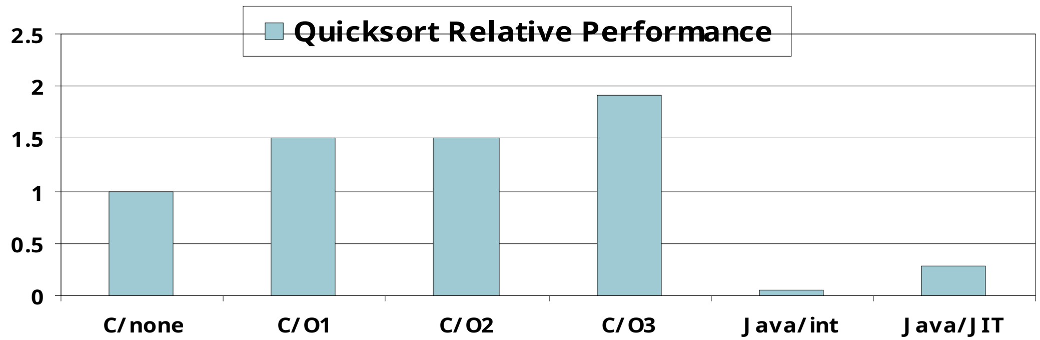


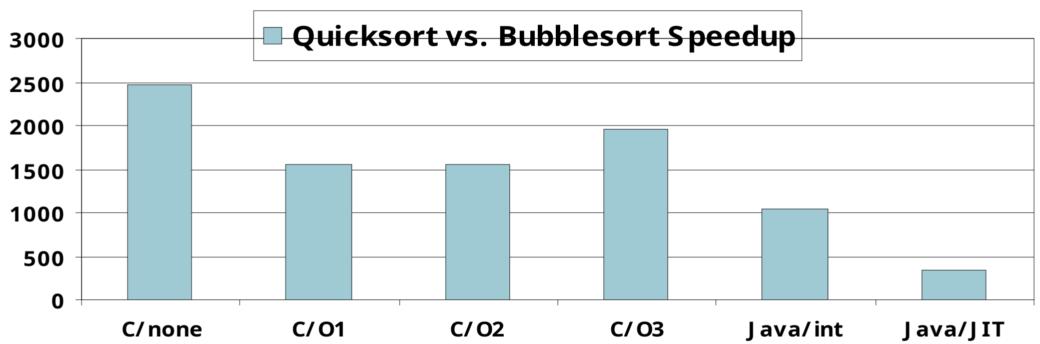




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good **performance indicators** in isolation
- Compiler **optimizations** are sensitive to the **algorithm**
- Java/**JIT** compiled code is significantly faster than JVM **interpreted**
  - Comparable to **optimized C** in some cases
- Nothing can fix a **dumb algorithm**!

more performance: Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid **indexing** complexity
  - “pointer arithmetic” (`T *p; p++`)

Example: Clearing an Array

```
clear1(int array[], int size) {
 int i;
 for (i = 0; i < size; i += 1)
   array[i] = 0;
}
```

```
clear2(int *array, int size) {
 int *p;
 for (p = &array[0]; p < &array[size];
      p = p +1)
   *p = 0;
}
```

```
      move $t0,$zero   # i = 0
loop1:sll $t1,$t0,2    # $t1 = i * 4
      add $t2,$a0,$t1  # $t2 =
                       #   &array[i]
      sw $zero, 0($t2) # array[i] = 0
      addi $t0,$t0,1   # i = i + 1
      slt $t3,$t0,$a1  # $t3 =
                       #   (i < size)
      bne $t3,$zero,loop1 # if (…)
                          # goto loop1
```

```
 move $t0,$a0           # p = & array[0]
      sll $t1,$a1,2     # $t1 = size * 4
      add $t2,$a0,$t1   # $t2 =
                        #   &array[size]
loop2:sw $zero,0($t0)   # Memory[p] = 0
      addi $t0,$t0,4    # p = p + 1
      slt $t3,$t0,$t2   # $t3 =
                        #(p<&array[size])
      bne $t3,$zero,loop2 # if (…)
                          # goto loop2
```

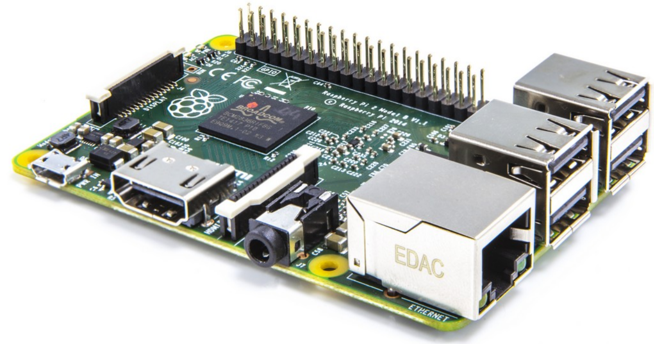
Comparison: Array vs. Pointer

- Multiply is "strength reduced" to shift
- Array version requires shift to be *inside* loop
  - part of index calculation for incremented `i`
  - versus incrementing pointer
- **Compiler** can achieve same effect as manual use of pointers
  - "Induction variable elimination"
  - Better to make program clearer and safer than to try to optimize. Modern compilers do a better optimization job than a human anyway.

Other architectures (than MIPS)

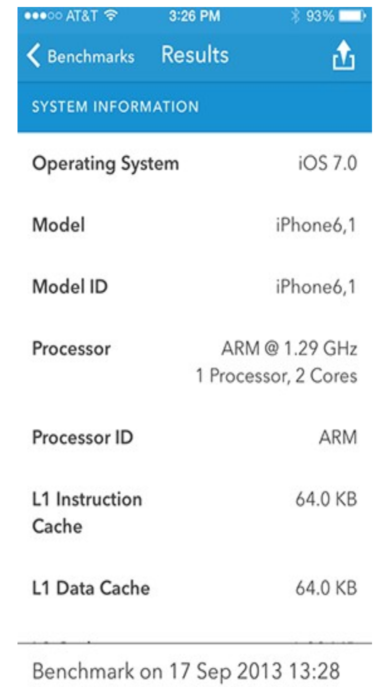
ARM

ARM: Advanced (Acorn) Risc Machine



Raspberry Pi 2 Model B single-board computer

900MHz quad-core ARM Cortex-A7 CPU



the most popular
embedded core

Galaxy Nexus
uses OMAP 4460 SoC

OMAP (Open Multimedia Applications Platform) is a series of image/video processors Systems on Chip from Texas Instruments.
Include a general-purpose ARM architecture processor core and one or more specialized co-processors.

ARM and MIPS Similarities

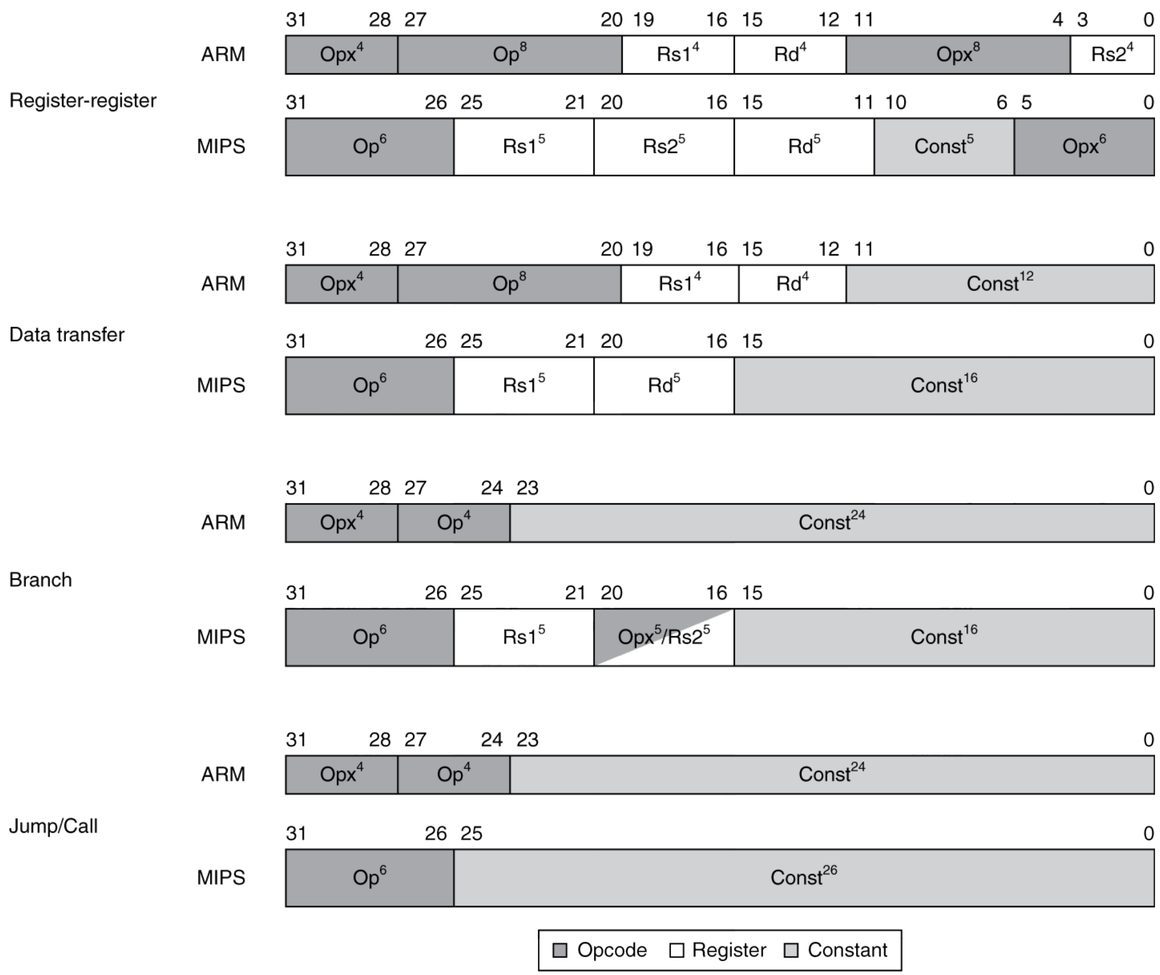
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Compare and Branch in ARM

- Uses **condition codes** for result of an arithmetic/logical instruction
  - **negative, zero, carry, overflow**
  - Compare instructions to set condition codes without keeping the result (crf. `slt` for MIPS)
- Each **instruction** can be **conditional**
  - **Top 4 bits** of instruction word: **condition** value
  - Can avoid branches over single instructions

Instruction Encoding



The Intel x86 ISA

Complex **I**nstruction **S**et **C**omputer (CISC)

Intel® 64 and IA-32 Architectures
Software Developer's Manual

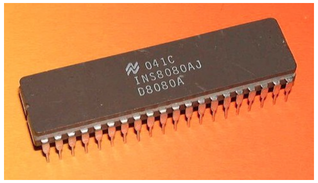
Volume 2 (2A, 2B, 2C & 2D):
Instruction Set Reference, A-Z

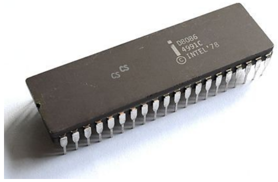
2198 pages !

https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - **Accumulator**, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (**CISC**)
  - 8087 (1980): floating-point coprocessor
    - Adds **FP** instructions and **register stack**





Register Stack

2 + (3 * 4)   + (2, *(3,4))   INFIX

3 4 (*) 2 +   POSTFIX

12

14

RPN (REVERSE POLISH NOTATION)

https://www.hpmuseum.org/hp41.htm

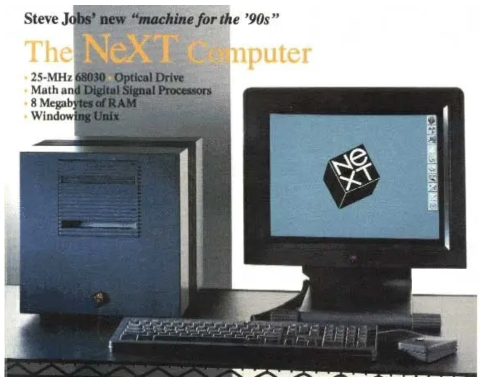
HP 41C

- FORTH
- POSTSCRIPT

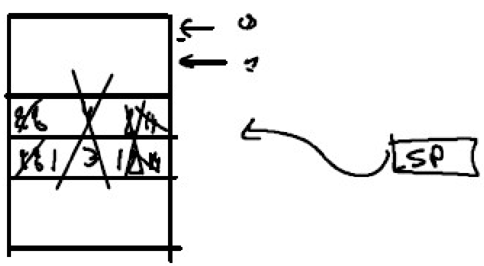
LINETO
MOVETO
STROKE

NEXT

DISPLAY POSTSCRIPT
NETWORK
SCREEN
X-WINDOW

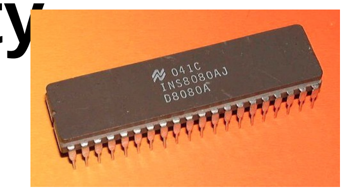


PUSH 3
PUSH 4
MUL → POP → A, POP → B, A*B → C, PUSH C
PUSH 2
ADD
DUP
ADD
LT 8
EQ
REPEAT 4



The Intel x86 ISA

- **Evolution** with **backward compatibility**
  - 8080 (1974): 8-bit microprocessor
    - **Accumulator**, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (**CISC**)
  - 8087 (1980): floating-point coprocessor
    - Adds **FP** instructions and **register stack**
  - 80286 (1982): 24-bit addresses, **MMU**
    - **Segmented memory** mapping and **protection**
  - 80386 (1985): **32-bit** extension (now IA-32)
    - Additional addressing modes and operations
    - **Paged** memory mapping as well as segments



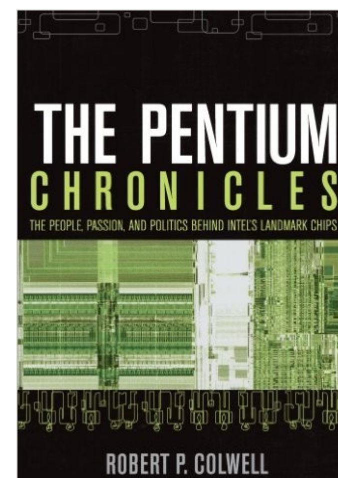



The Intel x86 ISA

Further evolution…

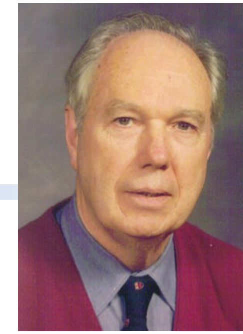
- i486 (1989): **pipelined**, **on-chip caches** and **FPU**
  - Compatible competitors: AMD, Cyrix, …
- Pentium (1993): **superscalar (ILP)**, **64-bit** datapath
  - Later versions added **MMX** (Multi-Media eXtension) instructions
  - The infamous **FDIV** bug

- Pentium Pro (1995), Pentium II (1997)
  - New **microarchitecture** (see Colwell, *The Pentium Chronicles*)

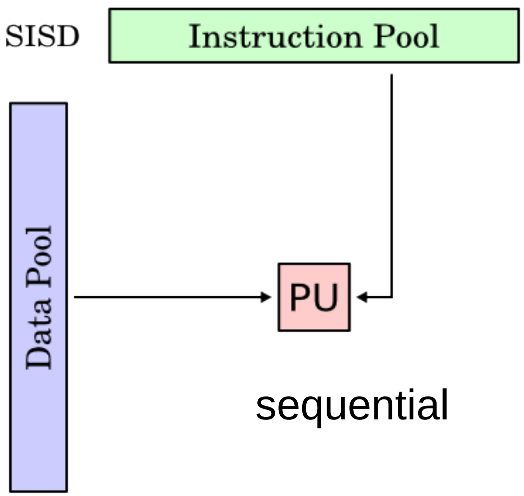


- Pentium III (1999)
  - Added SSE (**S**treaming **S**IMD –aka "**vector**"– **E**xtensions) and associated registers
- Pentium 4 (2001)
  - New microarchitecture
  - Added SSE2 instructions

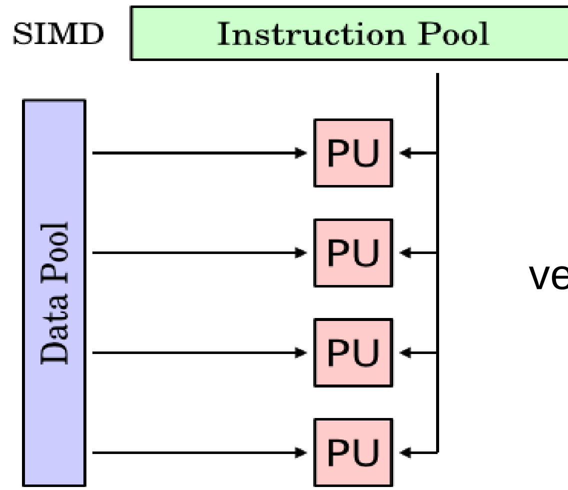
Flynn's Taxonomy (1966)



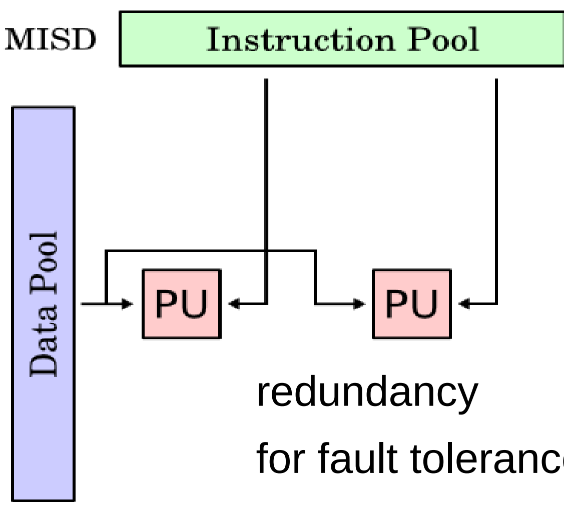
Michael J. Flynn



sequential



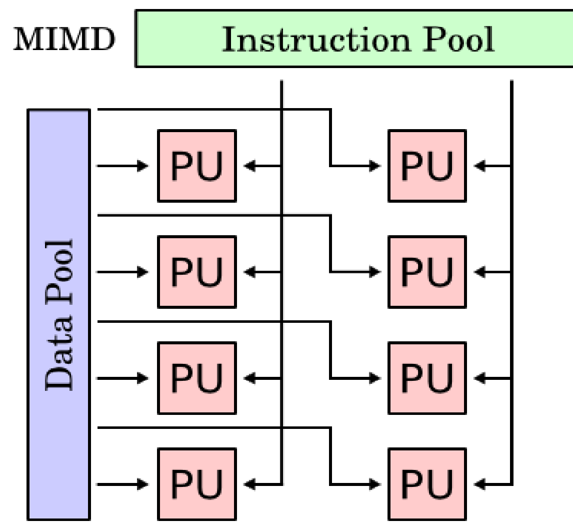
vector



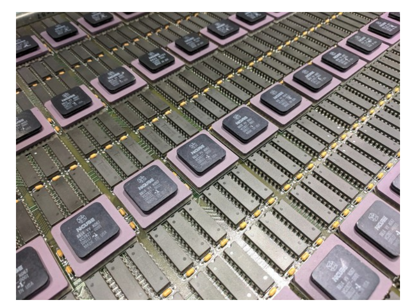
redundancy

for fault tolerance

(arguably still SISD)



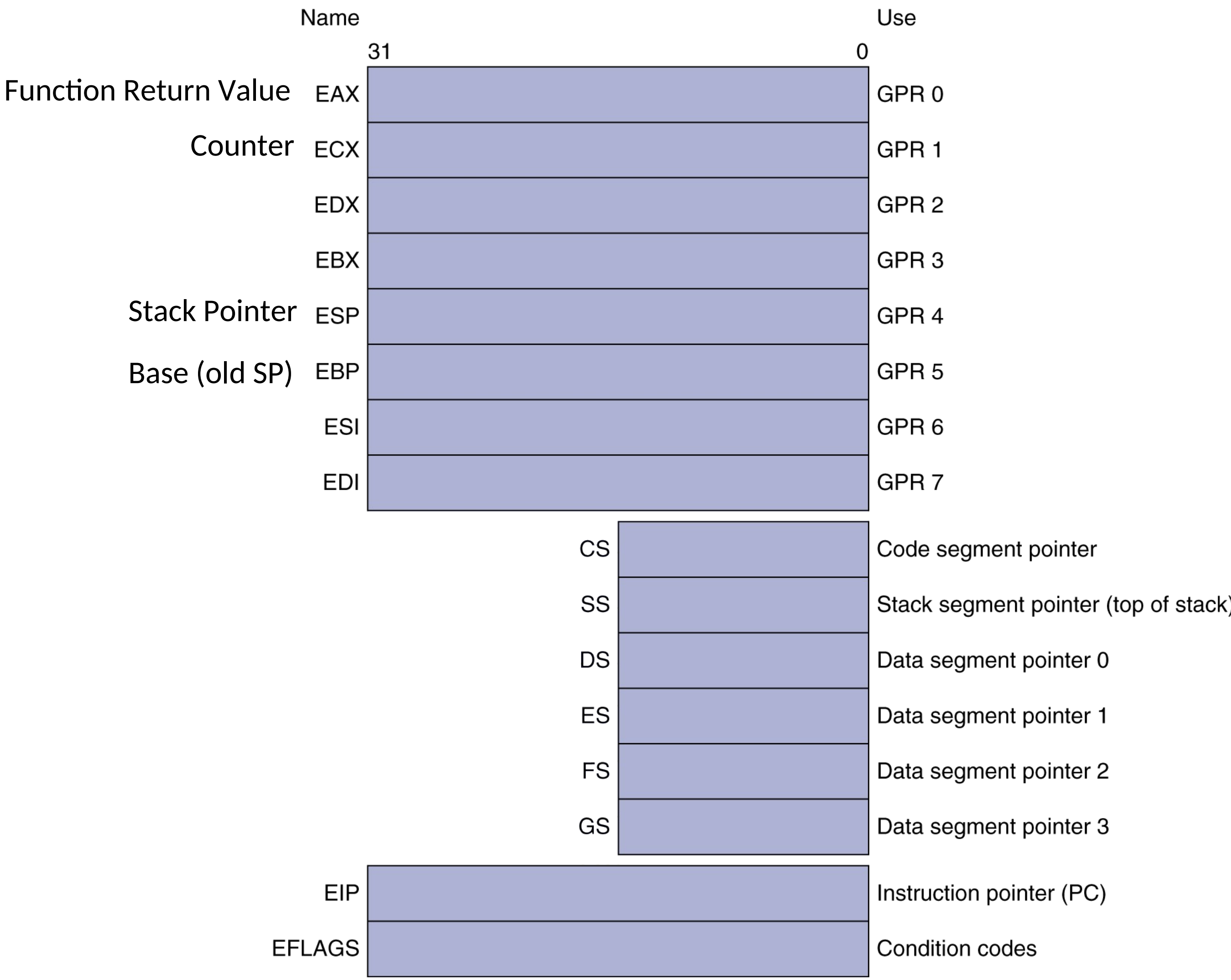
parallel



The Intel x86 ISA

- and further…
  - AMD64 (2003): extended architecture to **64 bits**
  - EM64T – Extended Memory 64 Technology (2004)
    - **AMD64** adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, **virtual machine** support
  - AMD64 (2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (2008)
    - Longer SSE registers, more instructions
- if Intel didn't extend *with compatibility*, its competitors would!
  - technical elegance ≠ market success

Basic x86 Registers



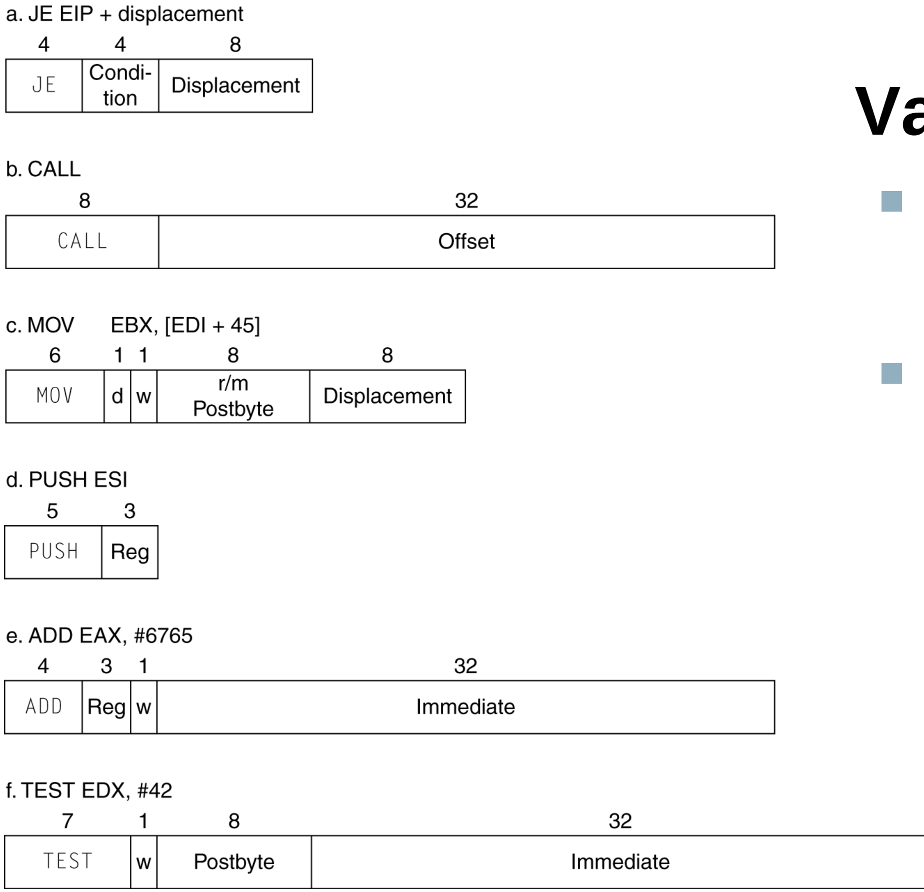
Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding



Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
  - Operand length, repetition, locking, …

Implementing IA-32

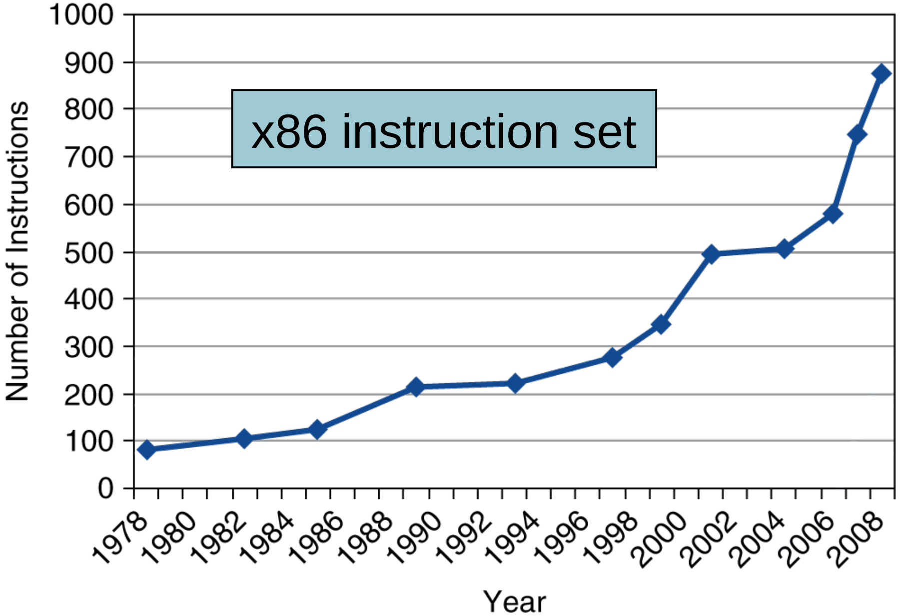
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler **micro-operations**
    - Simple instructions: 1 – 1
    - Complex instructions: 1 – many
  - Micro-engine similar to **RISC** (used for pipelining)
  - Market share makes this **economically viable**
- Comparable performance to RISC
  - Compilers avoid complex instructions

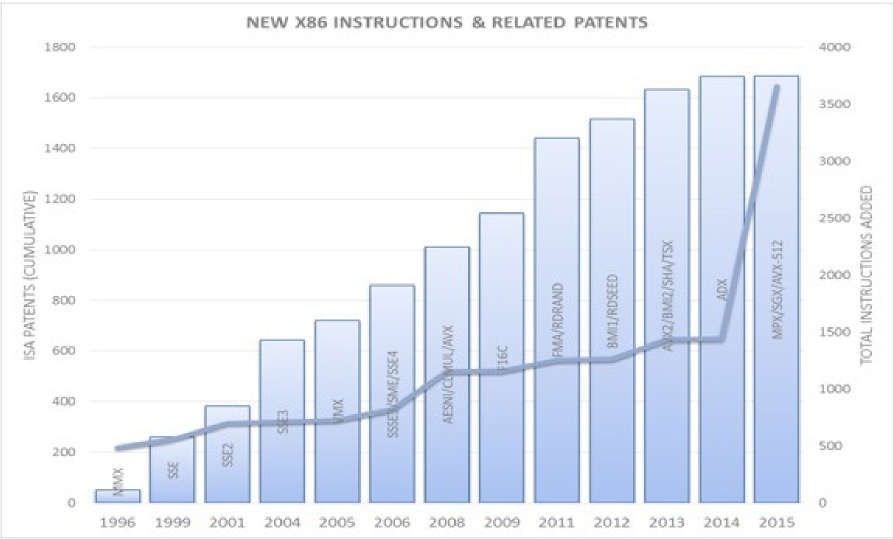
Fallacies

- Powerful (complex) instructions ⇒ higher performance
  - (+) Fewer instructions required
  - (+) Made for humans (readable)
  - (-) But complex instructions are hard to implement
    - May slow down **all** instructions, including simple ones
  - **Compilers** are good at making fast code from **simple** instructions
- Use assembly code for high performance
  - (-) More lines of code ⇒ more errors and less productivity
  - (-) But modern compilers are better at dealing with modern processors

Fallacies

- Backward compatibility ⇒
  accrete more (legacy) instructions





Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic (local to procedure) variable after procedure returns
  - e.g., passing pointer back via an argument
    → pointer becomes invalid when stack popped

Concluding Remarks

- **Design principles**
  1. **Smaller** is faster
  2. Simplicity favors **regularity**
  3. Make the **common case** fast
  4. Good design demands good **compromises**
- **Layers** of software/hardware
  - compiler, assembler, hardware
- MIPS: typical example of **RISC** ISAs
  - compare to **x86**, and to its **micro-architecture**