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
- Many modern computers also have **simple** instruction sets
  - → easier hardware and compiler optimization
  - **RISC (R**educed **I**nstruction **S**et **C**omputer**)**
    load-store architecture aka register-register architecture
  - **CISC** (**C**omplex **I**nstruction **S**et **C**omputer)
    register-memory architecture

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

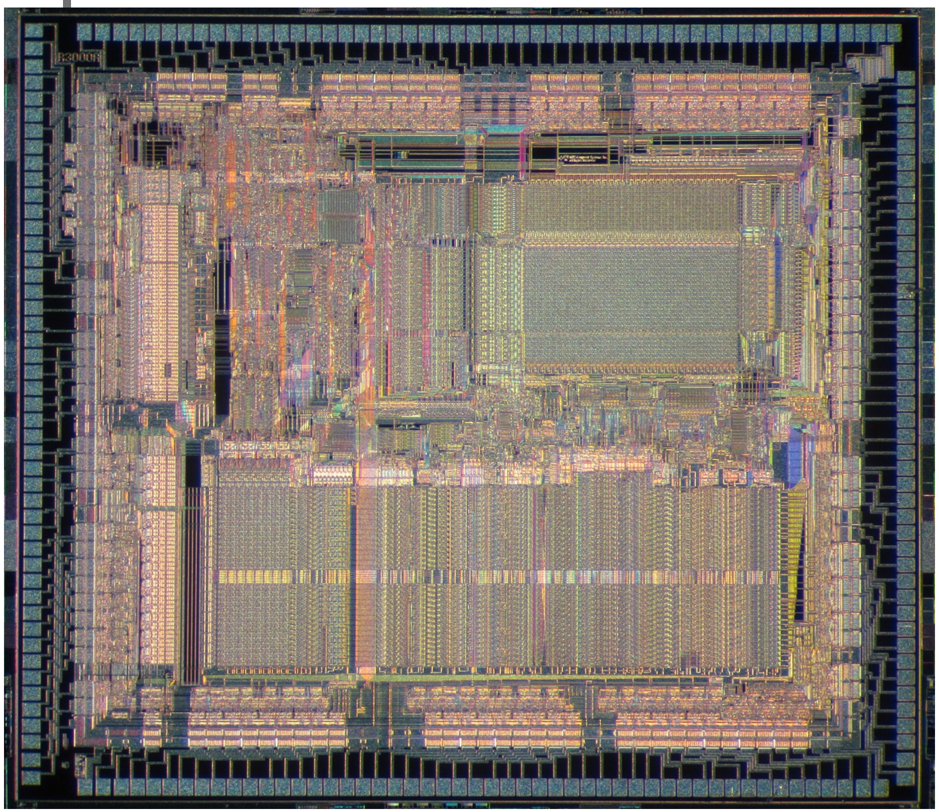
The MIPS Instruction Set

- 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

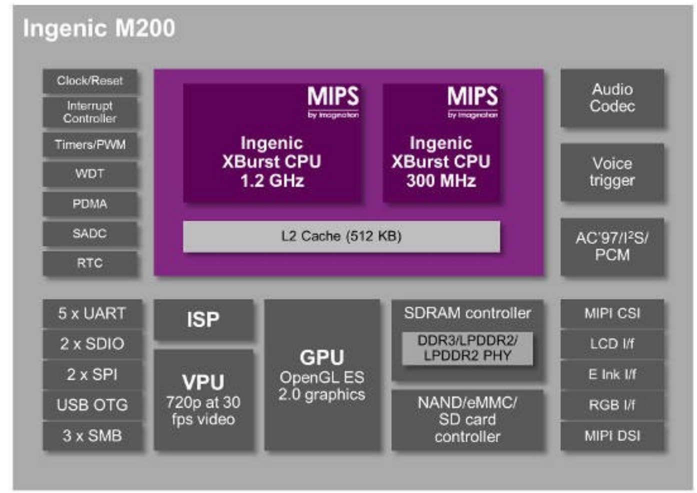
https://www.mips.com/blog/five-most-iconic-devices-to-use-mips-cpus/

MIPS
by Imagination

The future of MIPS?

MIPS Goes Open Source

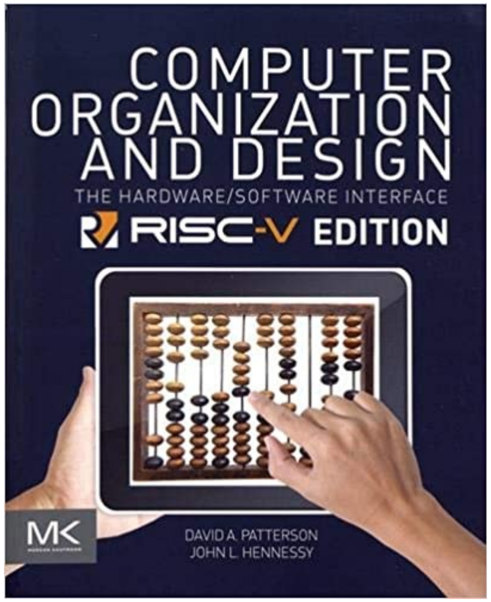
https://www.cdrinfo.com/d7/content/mips-goes-open-source



Smart watch **SoC** has dual MIPS cores

open source ISA of the future

RISC-V®



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)

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

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*
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/--/--/--

* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

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

© 2014 by Elsevier, Inc. All rights reserved. From Patterson and Hennessy, *Computer Organization and Design*, 5th ed.

Arithmetic Operations

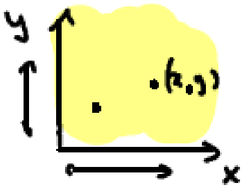
- Add and subtract, **three operands**
  - Two sources and one destination

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

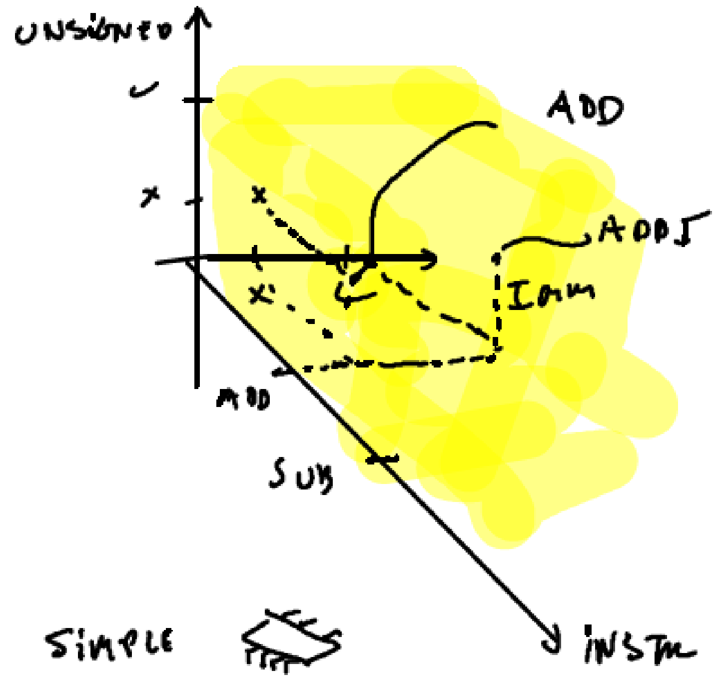
- All arithmetic operations have this "Three-Address Code" (TAC, 3AC) form
- ***Design Principle 1:***

  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



- 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`

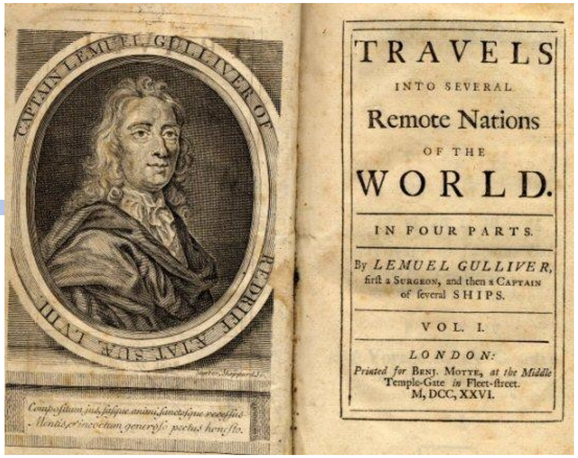
- Compiled MIPS code:

```
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** (load-store architecture)
  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

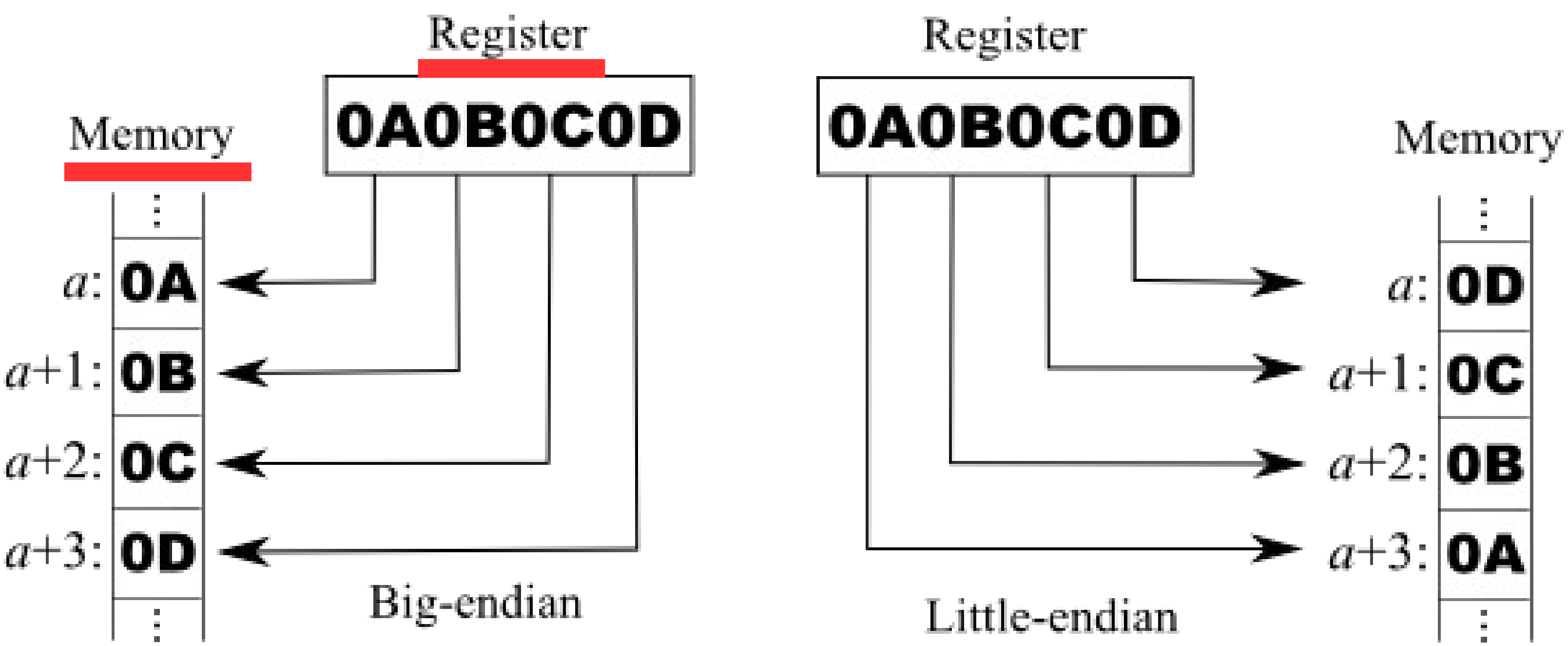


(Jonathan) Swift's point is that the difference between breaking the egg at the little-end and breaking it at the big-end is trivial. Therefore, he suggests, that everyone does it in his own preferred way.

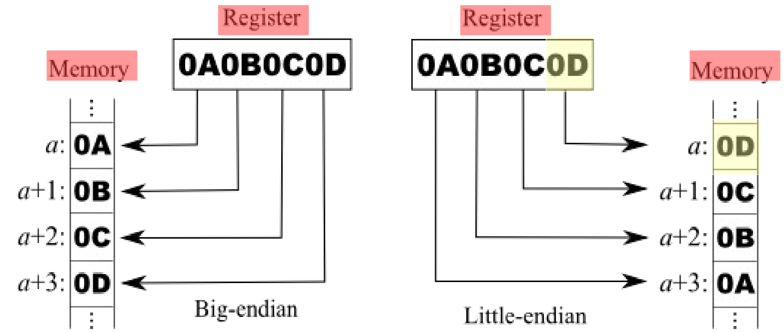
Danny Cohen IEN 137 **1 April** 1980

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

IEN == Internet Experiment Note
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

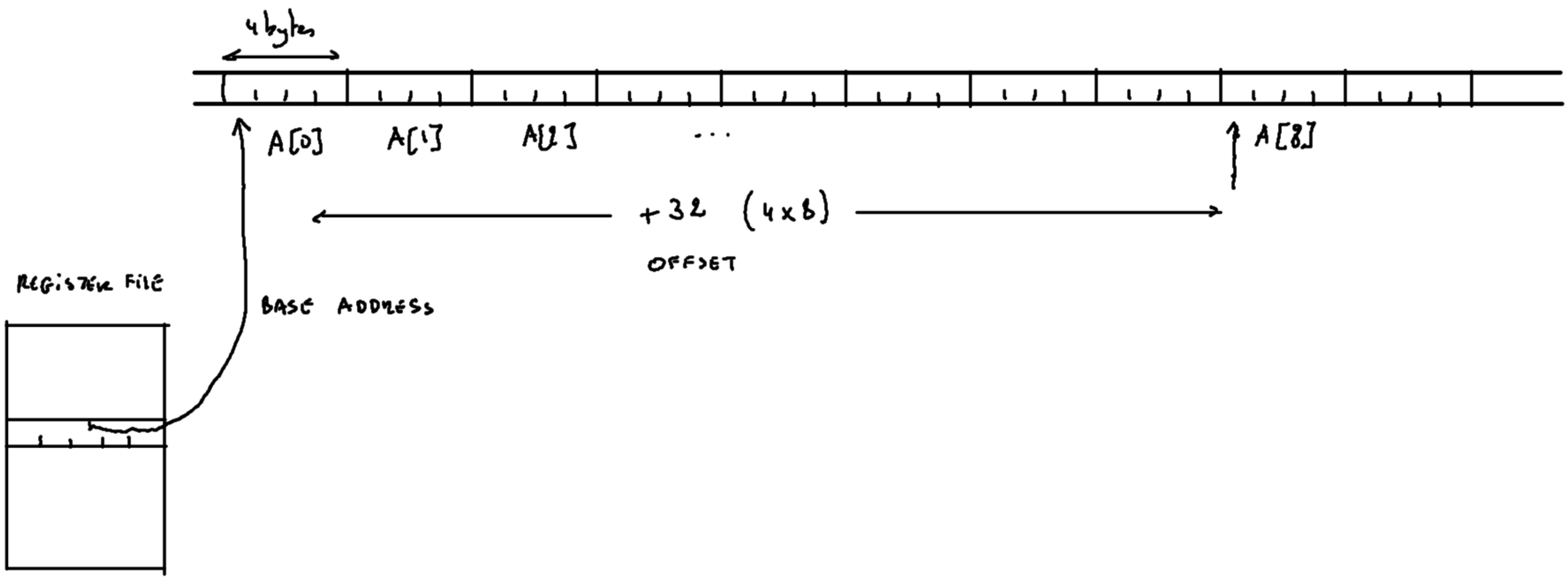
  (4 bytes per word)

```
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 (cfr. memory hierarchy)
  - 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 datapath
  - `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
  - signed values: extend with `1`s
- Examples: 8-bit to 16-bit
  - `+2: 0000 0010  →  0000 0000 0000 0010`
  - `–2: 1111 1110  →  1111 1111 1111 1110`

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 (on the right) 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 (on the left) 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) aka “blitting”

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

Byte/Halfword Operations

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

  String processing is a common case

`l`**`b`**` rt, offset(rs)`   `l`**`h`**` rt, offset(rs)`

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

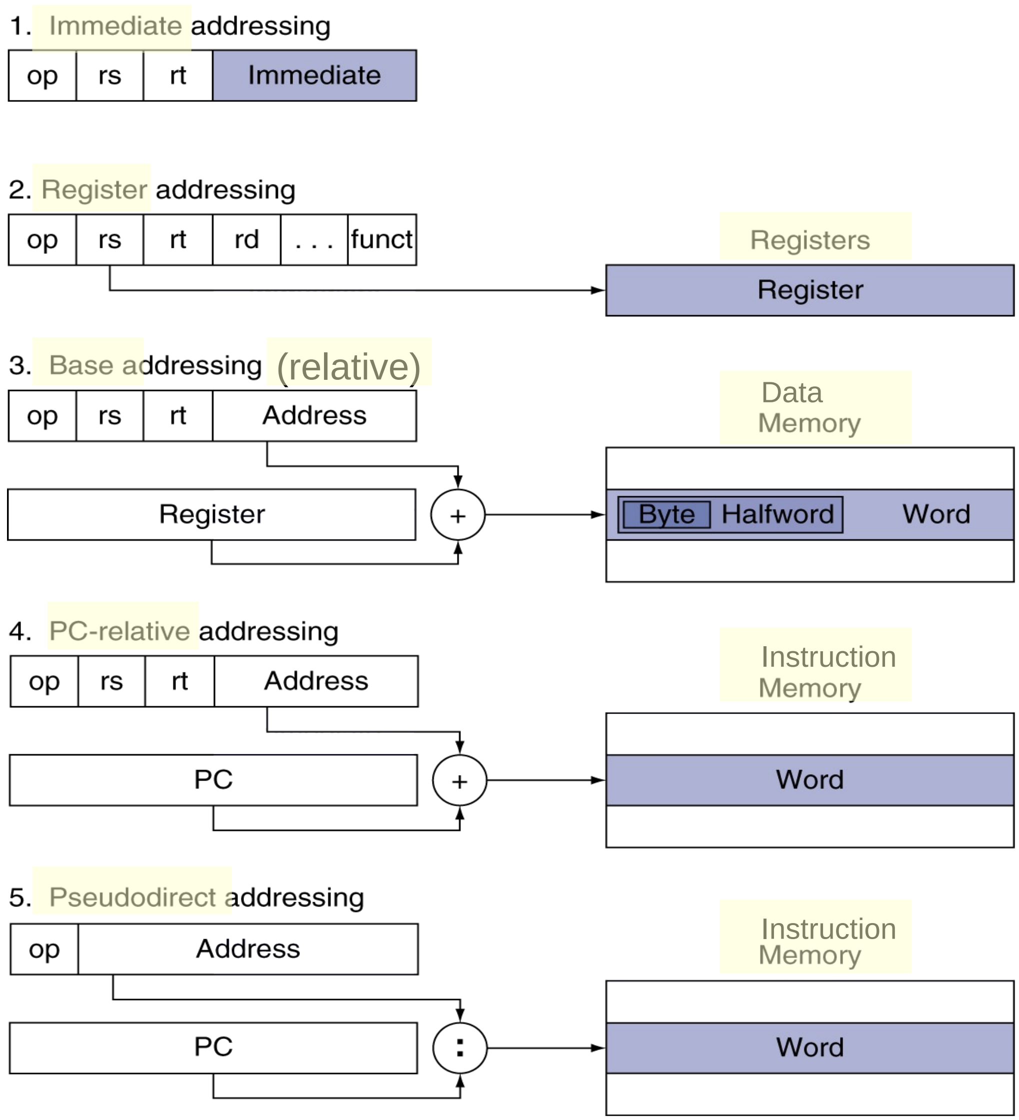
`l`**`bu`**` rt, offset(rs)`   `l`**`hu`**` rt, offset(rs)`

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

`s`**`b`**` rt, offset(rs)`   `s`**`h`**` rt, offset(rs)`

  - Store just rightmost byte/halfword

Addressing Modes (data/instr)



6. indirect

JR $R

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
```

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 = $\mathbf{PC_{31\ldots28} : (address \times 4)}$

Branching Far Away

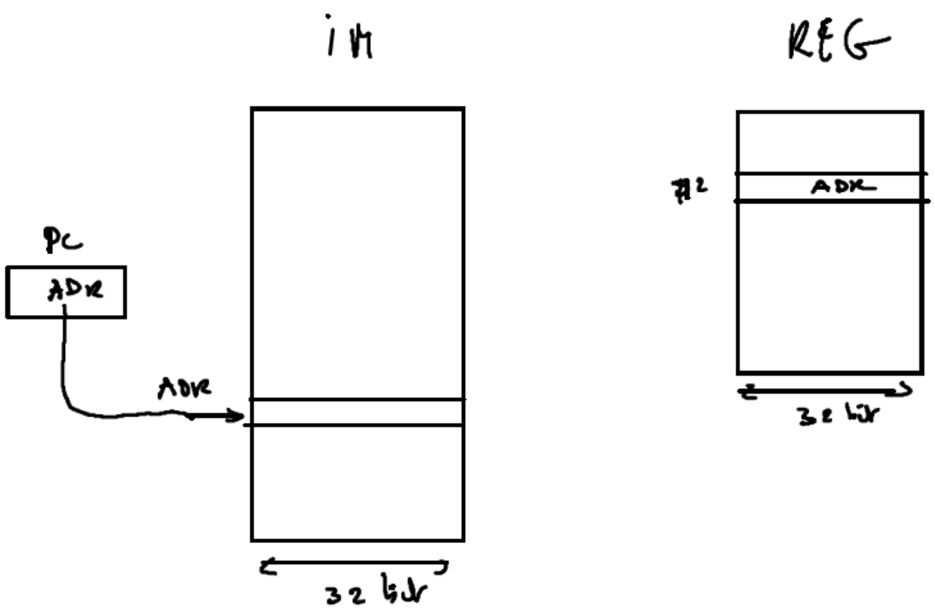
- If branch target is too far to encode with 16-bit offset of `beq`
  → assembler rewrites the code
- Example

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

Even Farther Away: JR

Full 32 bit address

ABSOLUTE, INDIRECT JUMP

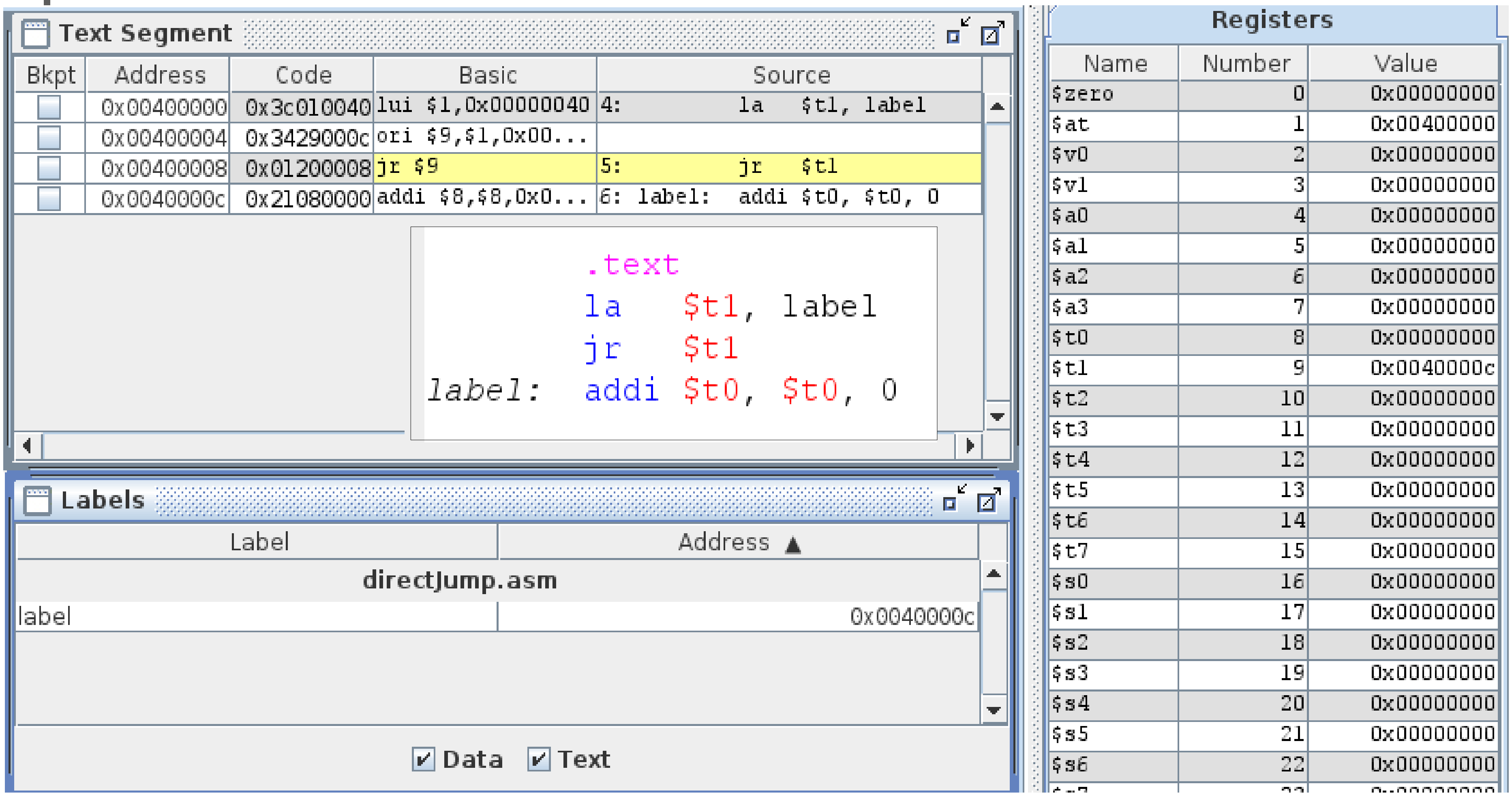


LA $2, ADR

JR $2

Even Farther Away: JR

Full 32 bit address



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c010040	lui $1,0x00000040	4: la $t1, label
	0x00400004	0x3429000c	ori $9,$1,0x00...	
	0x00400008	0x01200008	jr $9	5: jr $t1
	0x0040000c	0x21080000	addi $8,$8,0x0...	6: label: addi $t0, $t0, 0

```
        .text
        la    $t1, label
        jr    $t1
label:  addi  $t0, $t0, 0
```

Labels

Label	Address ▲
directJump.asm	
label	0x0040000c

☑ Data ☑ Text

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00400000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x0040000c
$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

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)
```