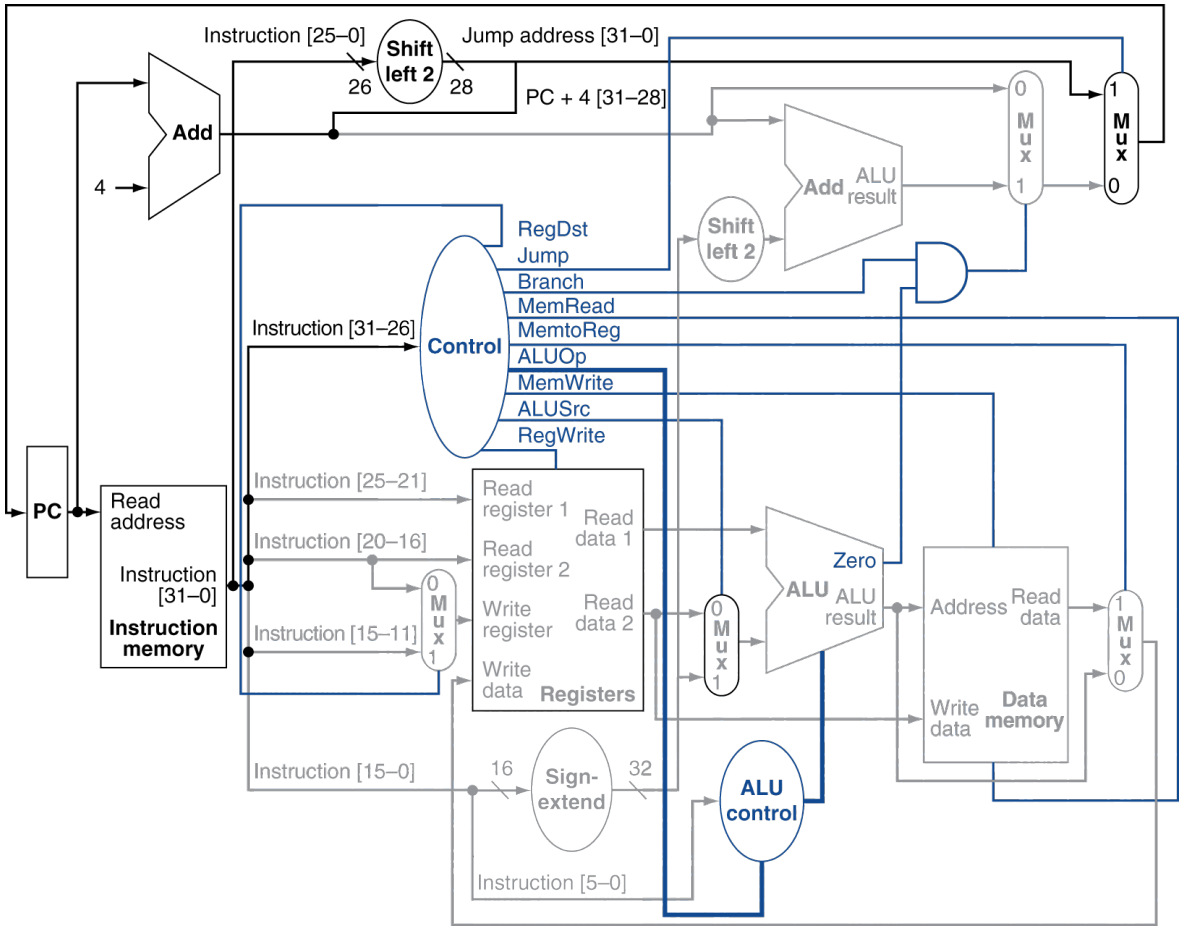
The Processor

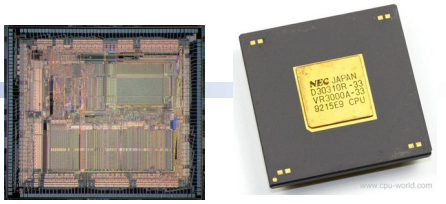
Designing the **datapath** (aka **architecture**)



Program Execution (performance)

- Algorithm
  - Determines **number of operations** executed
- Programming language, compiler, architecture (Instruction Set Architecture – ISA)
  - Determine number of **machine instructions** executed **per operation** (clock Cycles Per Instruction – CPI)
- Processor and memory system
  - Determine **how fast instructions** are **executed** (cycle time)
- I/O system (and OS)
  - Determines **how fast I/O operations** are **executed**

Program Execution



32 bit MIPS R3000 processor (115000 transistors) early 1990s

- We will examine two MIPS hardware implementations (aka “datatpath”) with identical ISAs:
  - A simplified version
  - A more realistic pipelined version (**I**nstruction-**L**evel **P**arallellism)
- We will subsequently introduce “exception” handling and what this requires in the datapath
- We will use a simple (but sufficient) subset of the instruction set, focusing on essential instructions.

  Different types of instructions (Instruction Set):

  - Memory access: `lw, sw`
  - Arithmetic/logical: `add, sub, and, or, slt`
  - Control transfer: `beq, j`

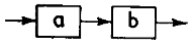
Böhm – Jacopini theorem

The “structured program” theorem (from programming language theory):

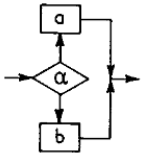
Böhm, Corrado and Jacopini, Giuseppe (1966).
”Flow Diagrams, Turing Machines and Languages with only Two Formation Rules”. Communications of the ACM 9(5):366-371.
http://www.cs.unibo.it/~martini/PP/bohm-jac.pdf

A class of control flow graphs can compute any computable function (algorithm) if it combines subprograms in only three specific ways (*i.e.,* by means of only three control structures):

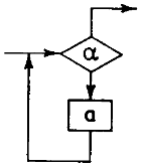
1) Executing one subprogram, and then another subprogram (sequence)



2) Executing one of two subprograms according to the value of a Boolean expression (selection)

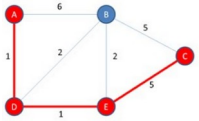


3) Repeatedly executing a subprogram as long as a Boolean expression is true (iteration)



Note: assembly/machine code is not “structured” HLL (as it uses **Go To**).

Edsger Dijkstra (1968).
"**Go To** Statement Considered Harmful". Communications of the ACM. 11 (3): 147–148.
https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf



Frank Rubin (1987).
""GOTO Considered Harmful" Considered Harmful". Communications of the ACM. 30 (3): 195–196.
http://www.ecn.purdue.edu/ParaMount/papers/rubin87goto.pdf

""GOTO Considered Harmful" Considered Harmful' Considered Harmful?" ...

Instruction Set Architecture (ISA)

Special Architectures:

- (Super) vector computers
- GPU (matrix operations … and AI)
- Special purpose (signal processing, ECU, ...)

Instruction Set Architecture (ISA)

Design Principles (HW/SW):

1. Regularity
2. Smaller is Faster
3. Make the Common Case Fast
4. Good Design demands Good Compromises

Instruction Set Architecture (ISA)

Different instruction **types**:

Memory access: `lw, sw`
Arithmetic/logical: `add, sub, and, or, slt`
Control transfer: `beq, j`

Different instruction **instances**:

```
add    $s1, $s2, $s3
add    $s1, $s1, $s2
add    $s1, $s1, $s1
```

Different instruction (encoding) **formats**:

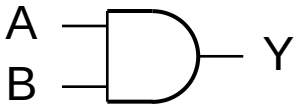
Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	`shamt`	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, `imm`. format
J-format	op	target address					Jump instruction format

Logic Design Basics (recap)

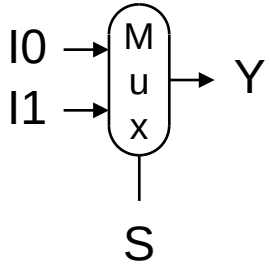
- Information encoded in **bi**nary digi**ts**
  - Low voltage = 0, High voltage = 1 (or reverse)
  - One wire per bit
  - Multi-bit data encoded on multi-wire **buses**
- **Combinational** element
  - **Operate** on **data**
  - Output is a **function** of input
- **State** (**sequential**) elements
  - **Store/Hold/Retrieve** information

Combinational Elements

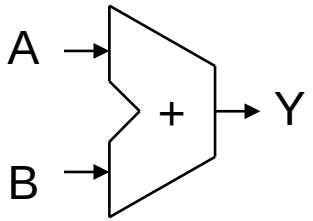
- AND-gate
  - Y = A & B



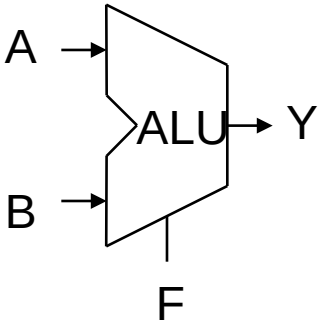
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B



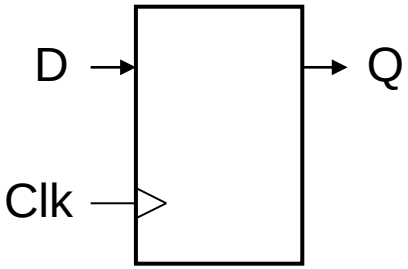
- Arithmetic/Logic Unit
  - Y = F(A, B)

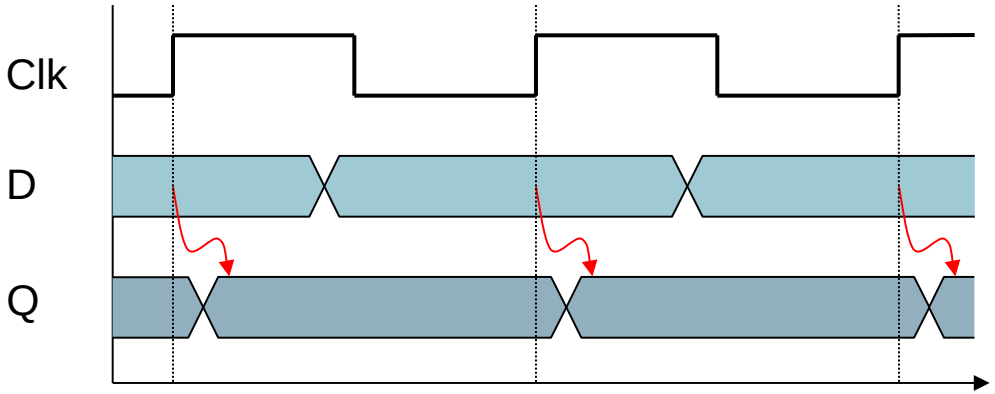


Sequential Elements

Register: stores data in a memory circuit

- Uses a clock signal Clk to determine **when** to **update** the **stored** value Q with D
- (rising/falling) **Edge-triggered**: **update** data in memory **when** Clk changes (from 0 to 1/1 to 0)

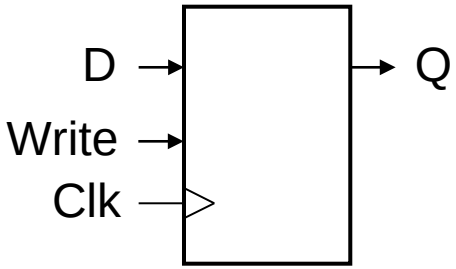


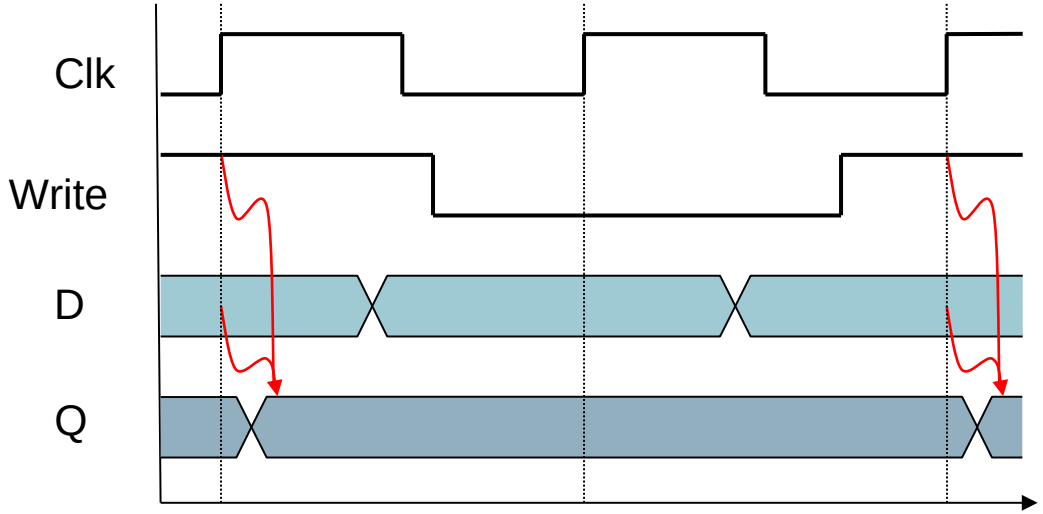


Sequential Elements

Register with **write control**

- Only updates on **clock edge**
  only when **write control input is 1**
- Used when stored value is to be kept over **multiple** clock cycles

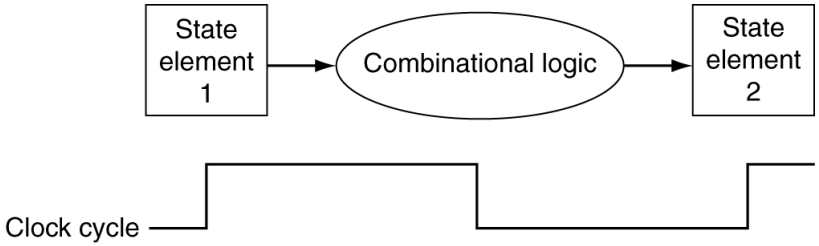


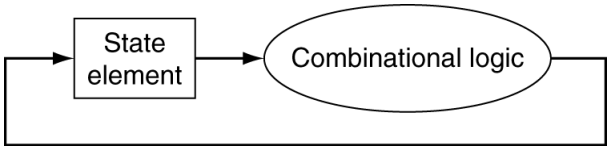


Clocking Methodology

Combinational logic
transforms data **during** clock cycles

- Between clock edges
- Input from state elements,
  Output to state element
- **Longest delay** due to combinational logic (implementing ISA “instructions”) determines **minimum** required **clock period**





Executing Machine Instructions

```
1          .data
2
3  values: .word   10
4          .word   12
5  result: .word   9
6
7          .text
8  start:
9
10 # ALU operations
11         li   $t1, 1
12         li   $t2, 2
13         add  $t3, $t1, $t2
14
15 # memory operations
16         la   $t0, values
17         lw   $t1, 0($t0)
18         lw   $t2, 4($t0)
19         add  $t3, $t1, $t2
20         la   $t0, result
21         sw   $t3, 0($t0)
22
23 # control flow
24         beq  $t3, $t3, start
25         addi $t3, $t3, 2
26
27         j    start
```

Executing Machine Instructions

Registers | Coproc 1 | Coproc 0

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		0x00400000
hi		0x00000000
lo		0x00000000

Executing Machine Instructions

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24090001	addiu $9,$0,0x00000001	11: li $t1, 1
	0x00400004	0x240a0002	addiu $10,$0,0x0000...	12: li $t2, 2
	0x00400008	0x012a5820	add $11,$9,$10	13: add $t3, $t1, $t2
	0x0040000c	0x3c011001	lui $1,0x00001001	16: la $t0, values
	0x00400010	0x34280000	ori $8,$1,0x00000000	
	0x00400014	0x8d090000	lw $9,0x00000000($8)	17: lw $t1, 0($t0)
	0x00400018	0x8d0a0004	lw $10,0x00000004($8)	18: lw $t2, 4($t0)
	0x0040001c	0x012a5820	add $11,$9,$10	19: add $t3, $t1, $t2
	0x00400020	0x3c011001	lui $1,0x00001001	20: la $t0, result
	0x00400024	0x34280008	ori $8,$1,0x00000008	
	0x00400028	0xad0b0000	sw $11,0x00000000($8)	21: sw $t3, 0($t0)
	0x0040002c	0x116bfff4	beq $11,$11,0xfffffff4	24: beq $t3, $t3, start
	0x00400030	0x216b0002	addi $11,$11,0x0000...	25: addi $t3, $t3, 2
	0x00400034	0x08100000	j 0x00400000	27: j start

Executing Machine Instructions

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24090001	addiu $9,$0,0x00000001	11: li $t1, 1
	0x00400004	0x240a0002	addiu $10,$0,0x0000...	12: li $t2, 2
	0x00400008	0x012a5820	add $11,$9,$10	13: add $t3, $t1, $t2
	0x0040000c	0x3c011001	lui $1,0x00001001	16: la $t0, values
	0x00400010	0x34280000	ori $8,$1,0x00000000	
	0x00400014	0x8d090000	lw $9,0x00000000($8)	17: lw $t1, 0($t0)
	0x00400018	0x8d0a0004	lw $10,0x00000004($8)	18: lw $t2, 4($t0)
	0x0040001c	0x012a5820	add $11,$9,$10	19: add $t3, $t1, $t2
	0x00400020	0x3c011001	lui $1,0x00001001	20: la $t0, result
	0x00400024	0x34280008	ori $8,$1,0x00000008	
	0x00400028	0xad0b0000	sw $11,0x00000000($8)	21: sw $t3, 0($t0)
	0x0040002c	0x116bfff4	beq $11,$11,0xfffffff4	24: beq $t3, $t3, start
	0x00400030	0x216b0002	addi $11,$11,0x0000...	25: addi $t3, $t3, 2
	0x00400034	0x08100000	j 0x00400000	27: j start

Labels

Label	Address ▲
tst.asm	
start	0x00400000
values	0x10010000
result	0x10010008

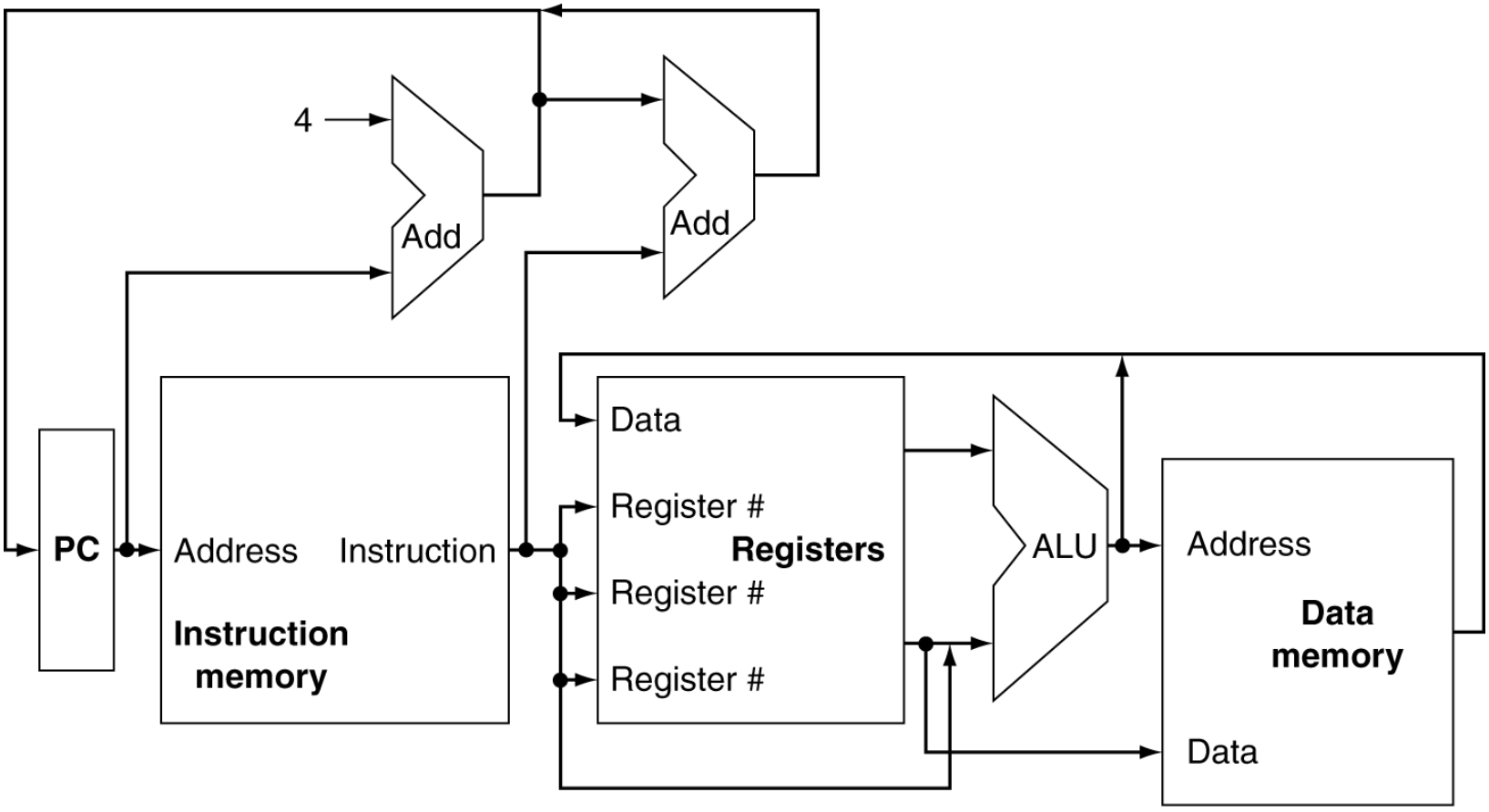
Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	
0x10010000	0x0000000a	0x0000000c	0x00000009	0x00000000	0x00000000	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
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

Instruction Execution

- **PC** → instruction memory, **fetch** instruction and **instruction decode**
- Register numbers → **register file**, **read** registers
- Depending on **instruction type (class)**
  - Use **ALU** to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data **memory** for load/store
  - **PC** ← **PC + 4** (“next sequential instruction”) or **target address**

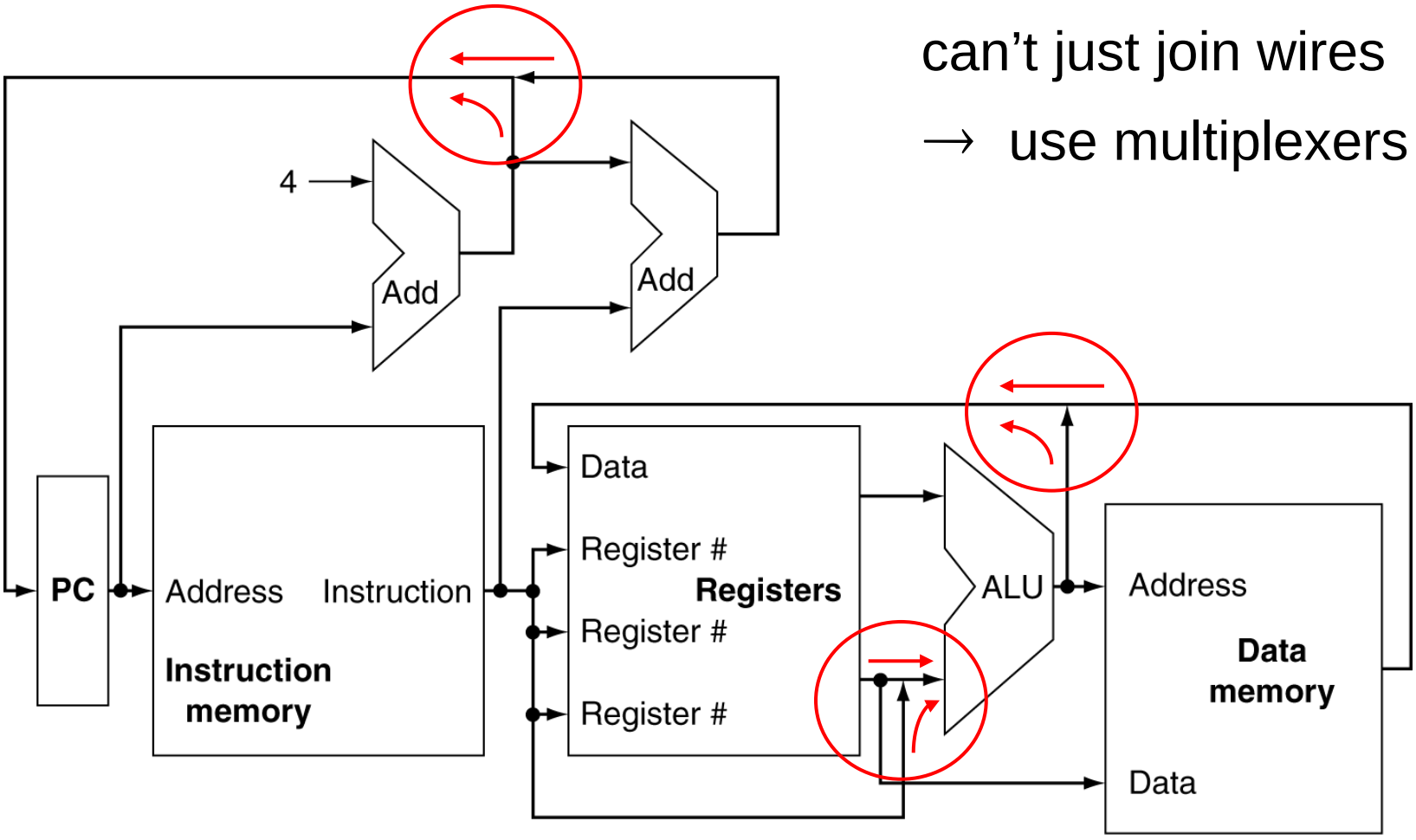
CPU Overview



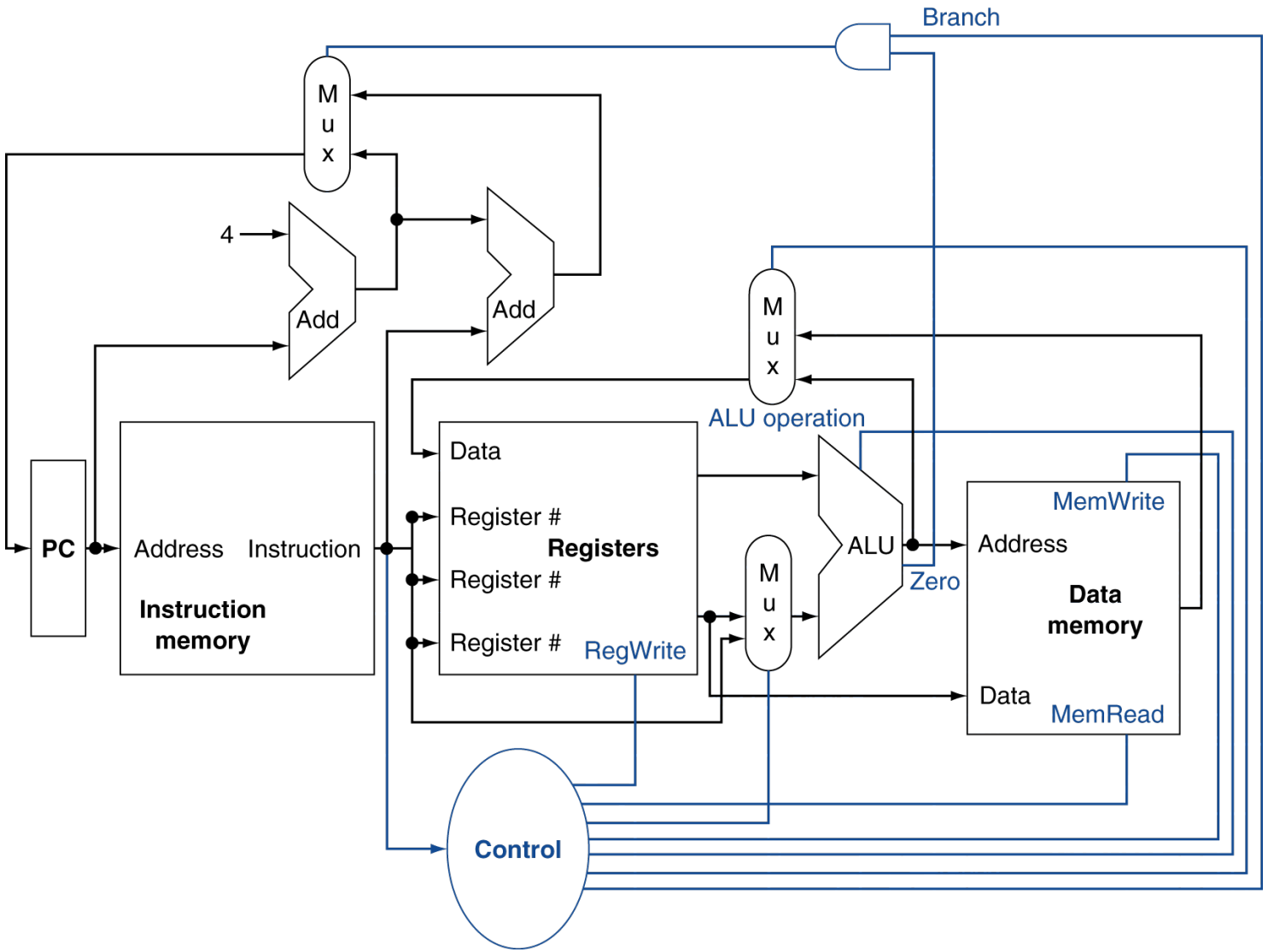
Multiplexers

can’t just join wires

→ use multiplexers



Control (“orchestrate”)



Building a Datapath

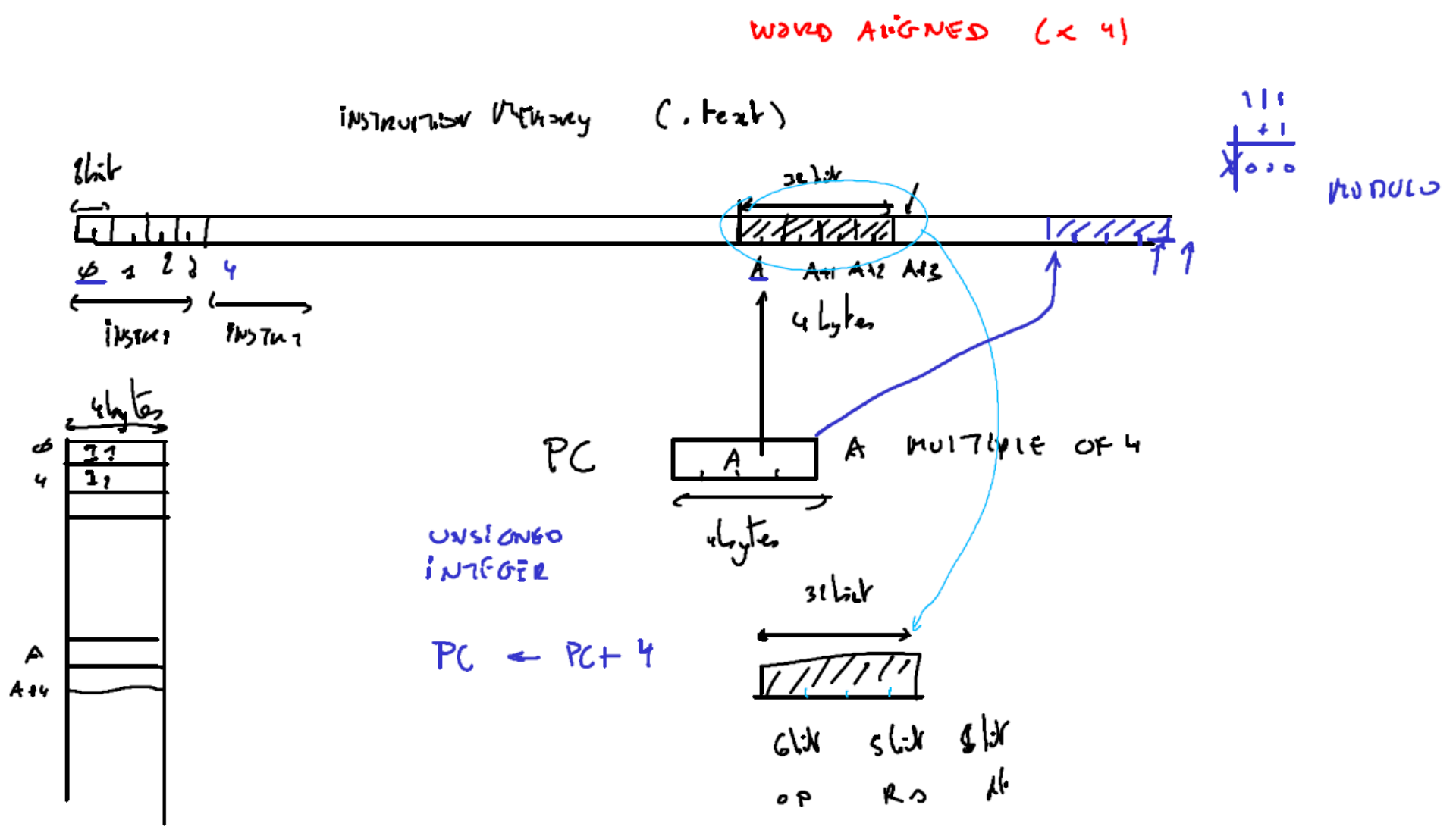
- Datapath =

  CPU **hardware architecture**
  that processes **instructions** and **data**

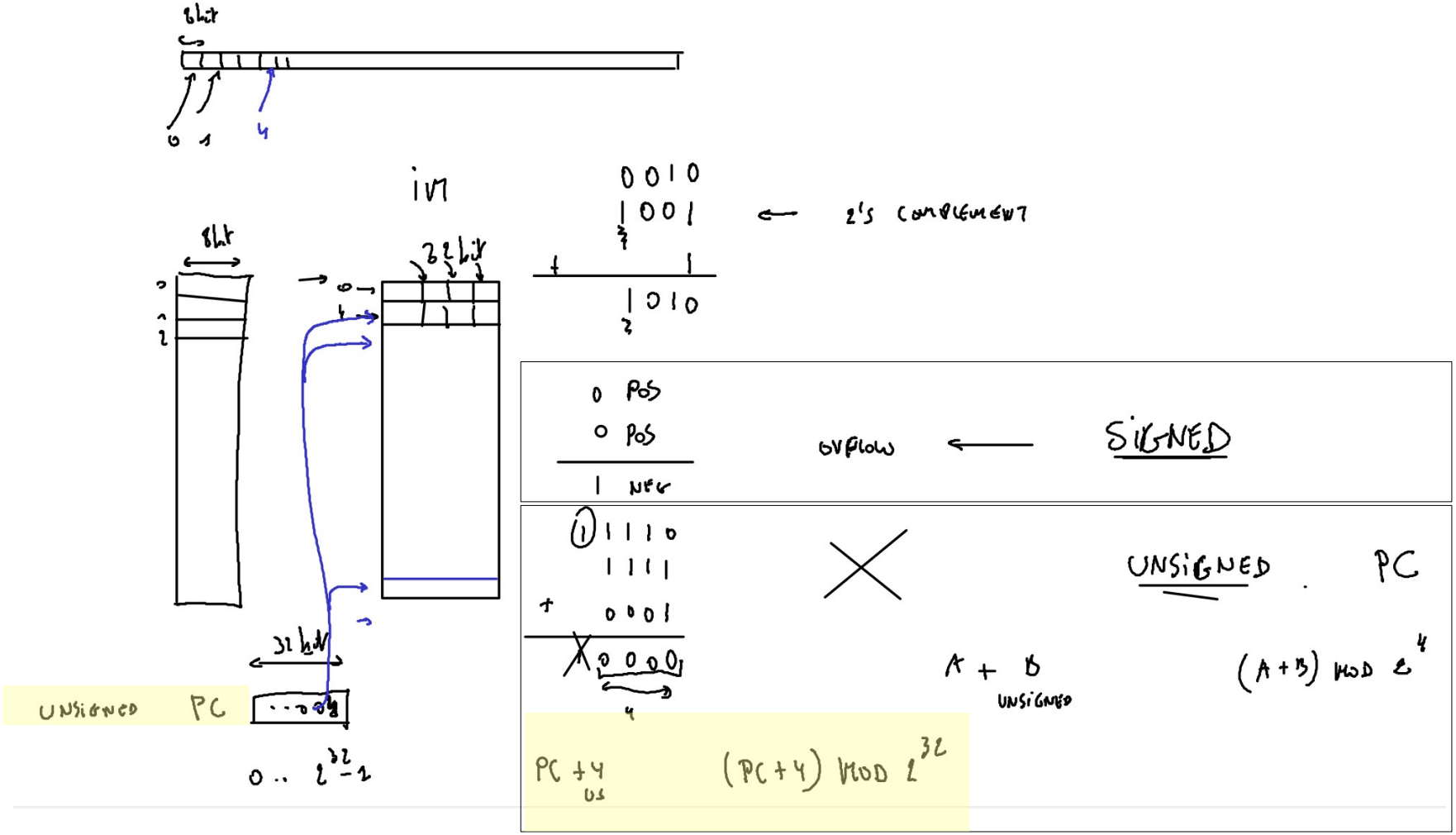
  - registers, ALUs, multiplexers, memories

- We will build a simplified MIPS datapath incrementally, refining the overview design

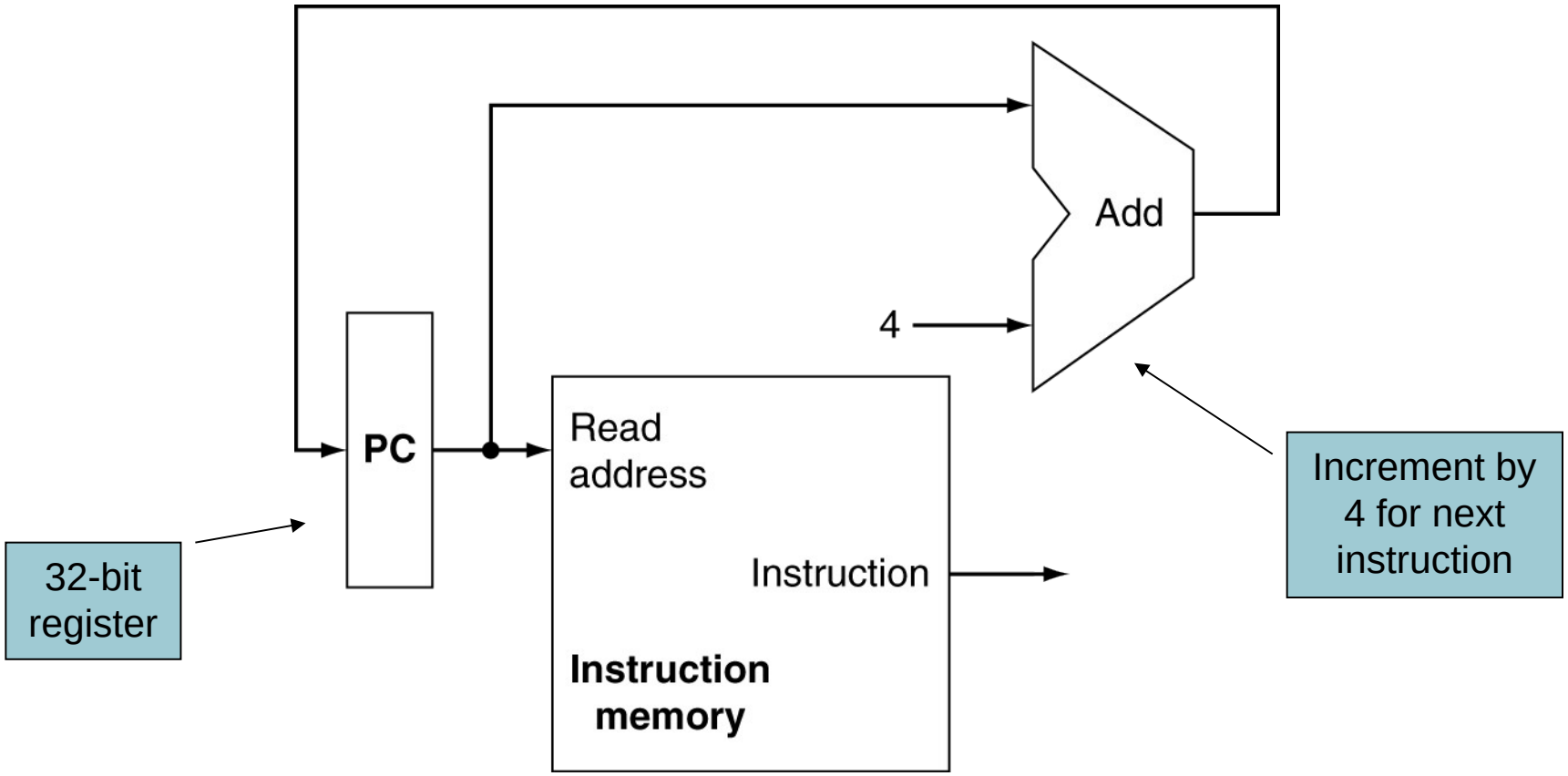
Instruction Fetch



Program Counter (PC) is unsigned



Instruction Fetch

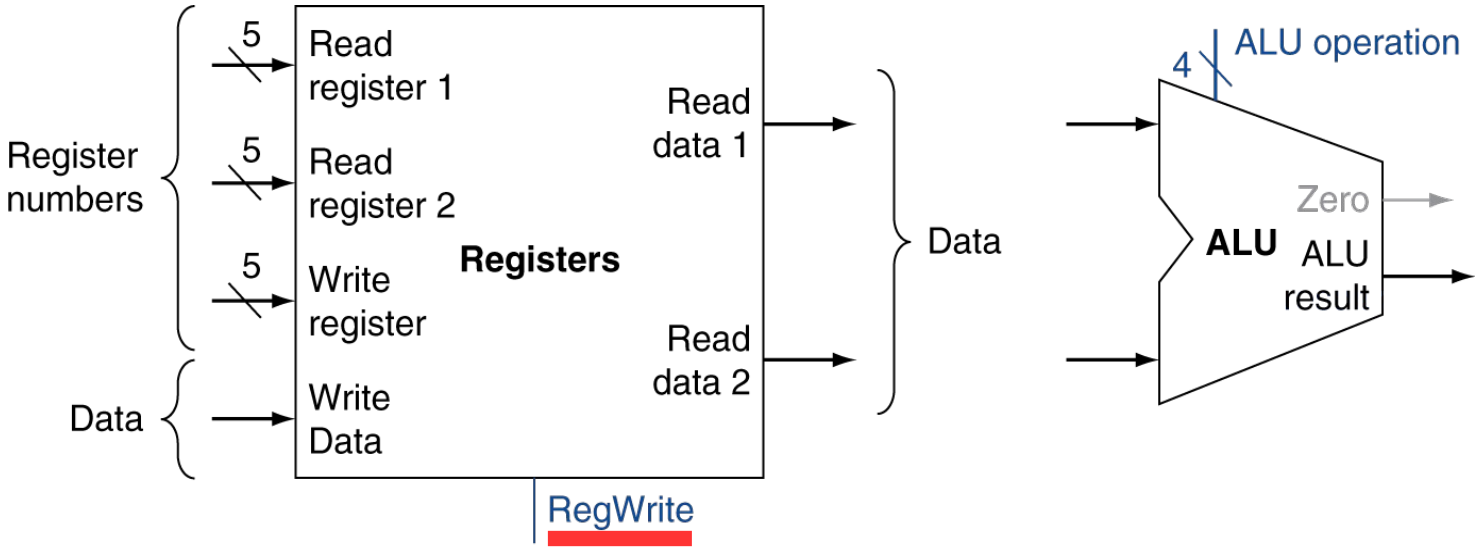


R-Format Instructions

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

why here?



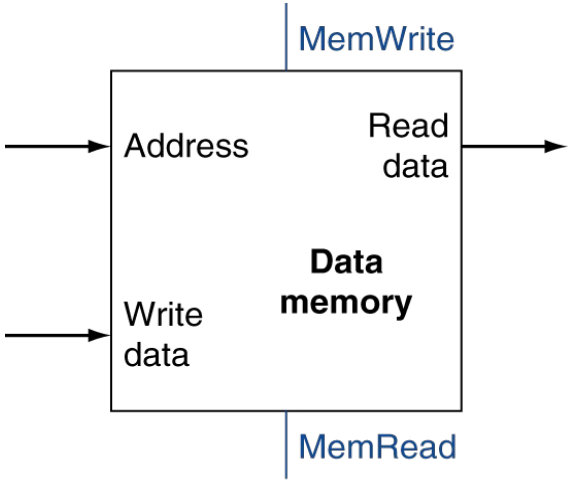
a. Registers

b. ALU

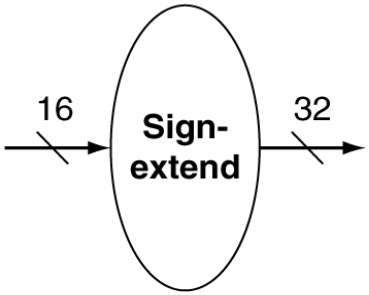
Load/Store Instructions

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

- Read register operands
- Calculate address using 16-bit offset
  - use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

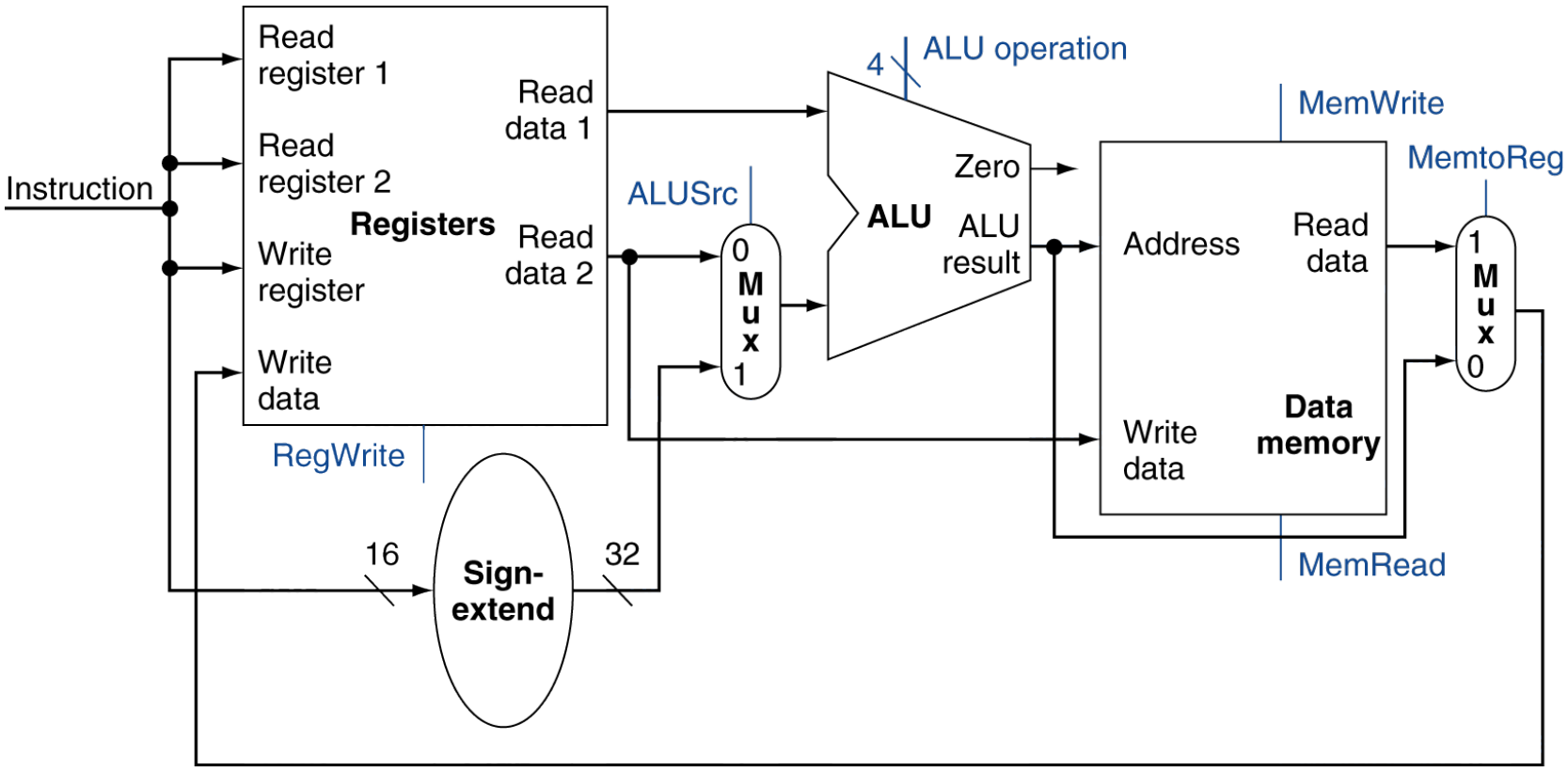


a. Data memory unit



b. Sign extension unit

R-Type/Load/Store Datapath

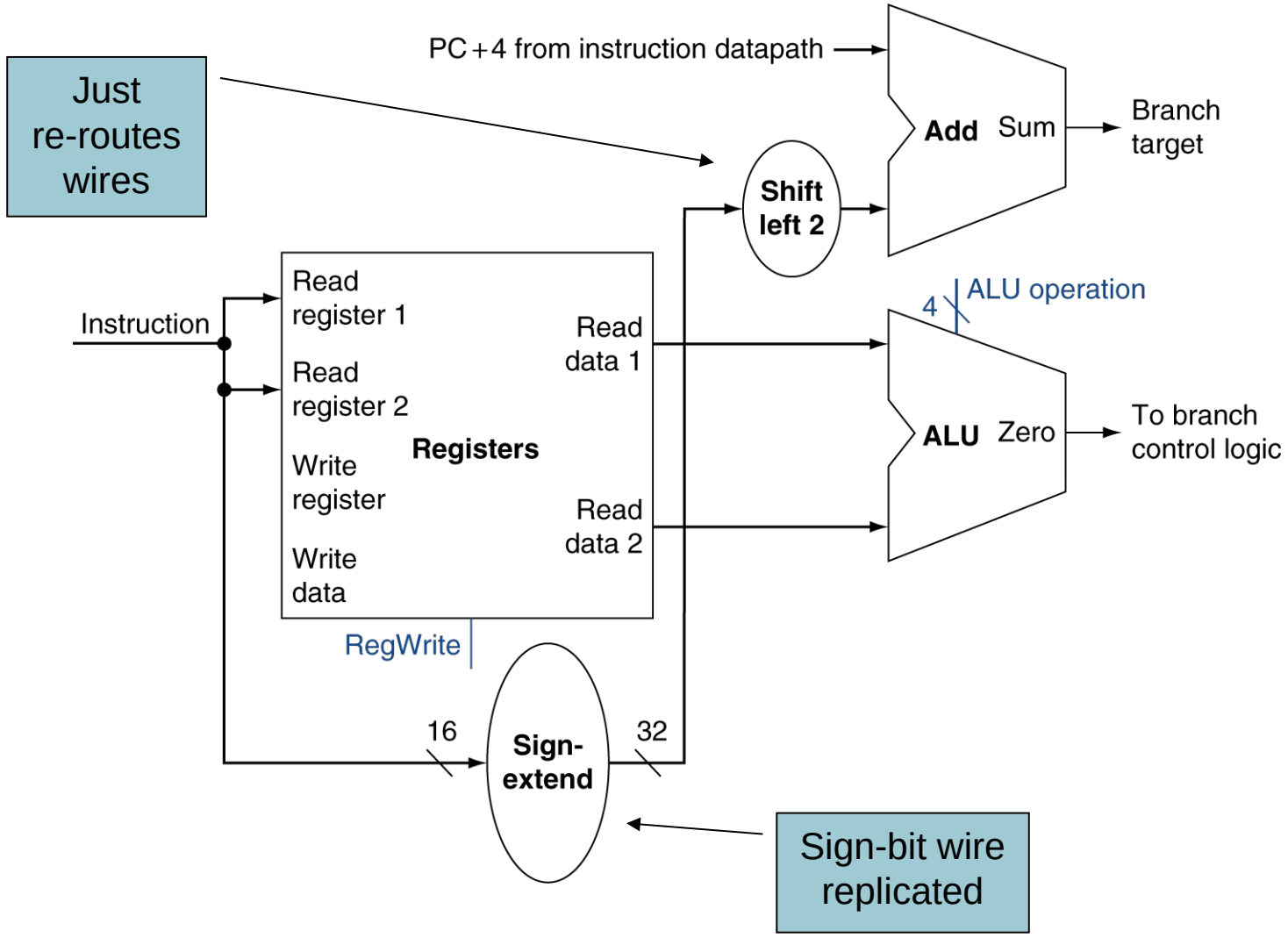


Branch Instructions

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places
    (instructions are word-aligned)
  - Add to PC **+ 4**
    - Already calculated by instruction fetch

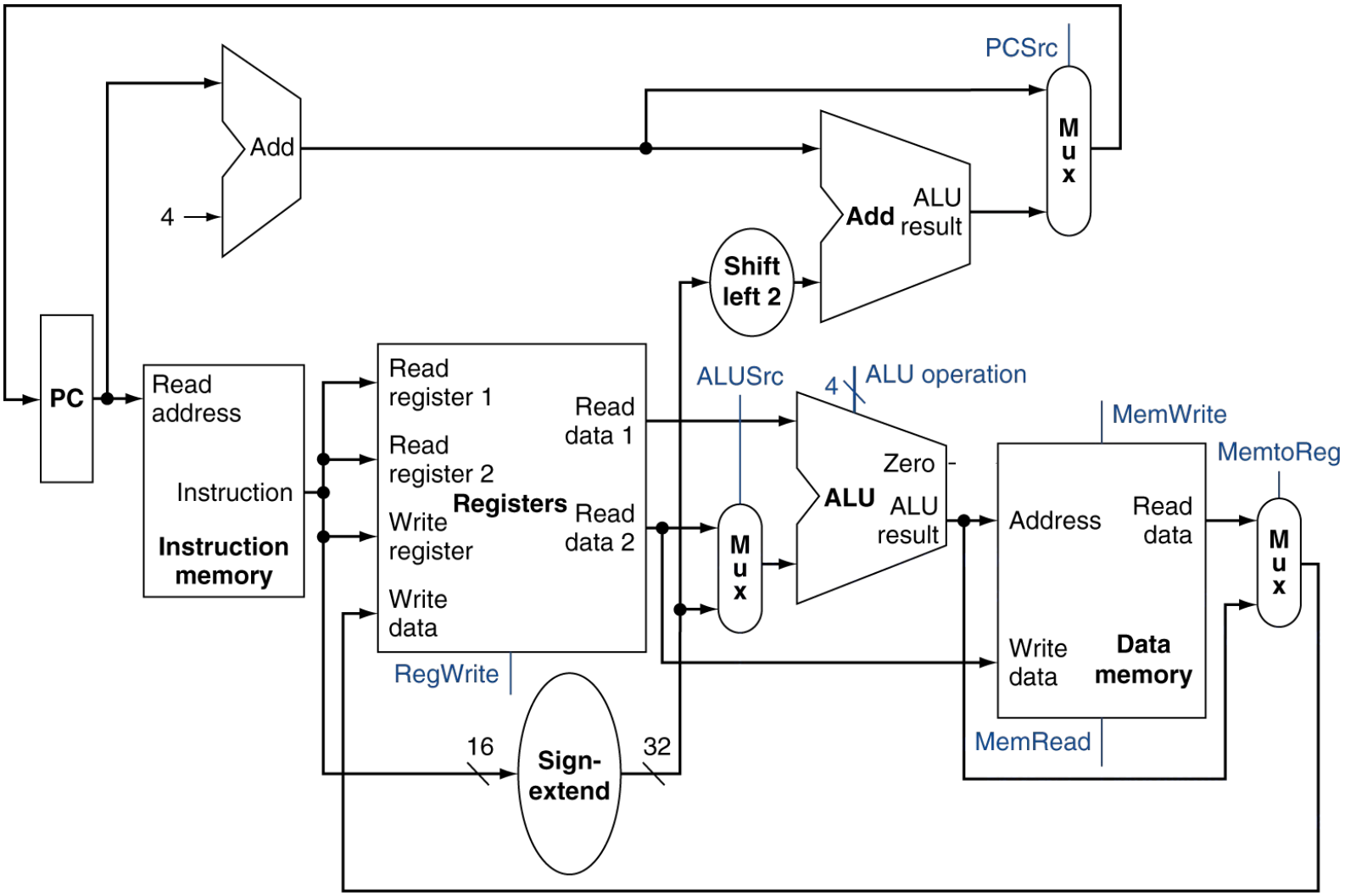
Branch Instructions



Composing the Elements

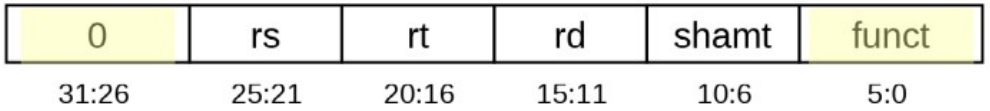
- First attempt at a datapath that processes one **instruction** in **one clock cycle**
  - Each datapath element can only do one function at a time (*i.e.,* in one clock cycle)
  - Hence, we need **separate** instruction and data memories!
- Use **multiplexers** where **alternate data sources** (*e.g.,* from ALU or from memory) are used for different instructions

Full Datapath (incomplete)



ALU Control

- ALU used for
  - R-type: Function depends on `funct` field



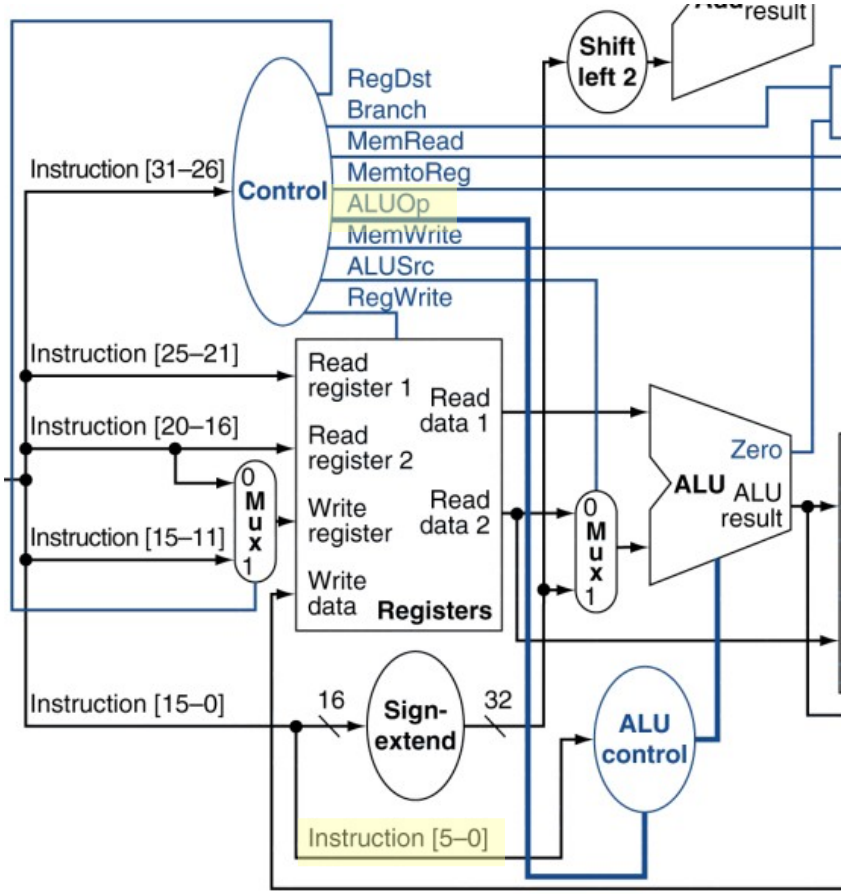
  - Load/Store: Function = **add**
  - Branch `beq`: Function = **subtract**

ALU control	Function
0000	and
0001	or
0010	add
0110	subtract
0111	Set-**on**-Less-Than
1100	nor

ALU Control

- 2-bit ALU Operation derived from opcode

R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0
Load/ Store beq	35 or 43	rs	rt	address		
	31:26	25:21	20:16	15:0		
Branch	4	rs	rt	address		
	31:26	25:21	20:16	15:0		



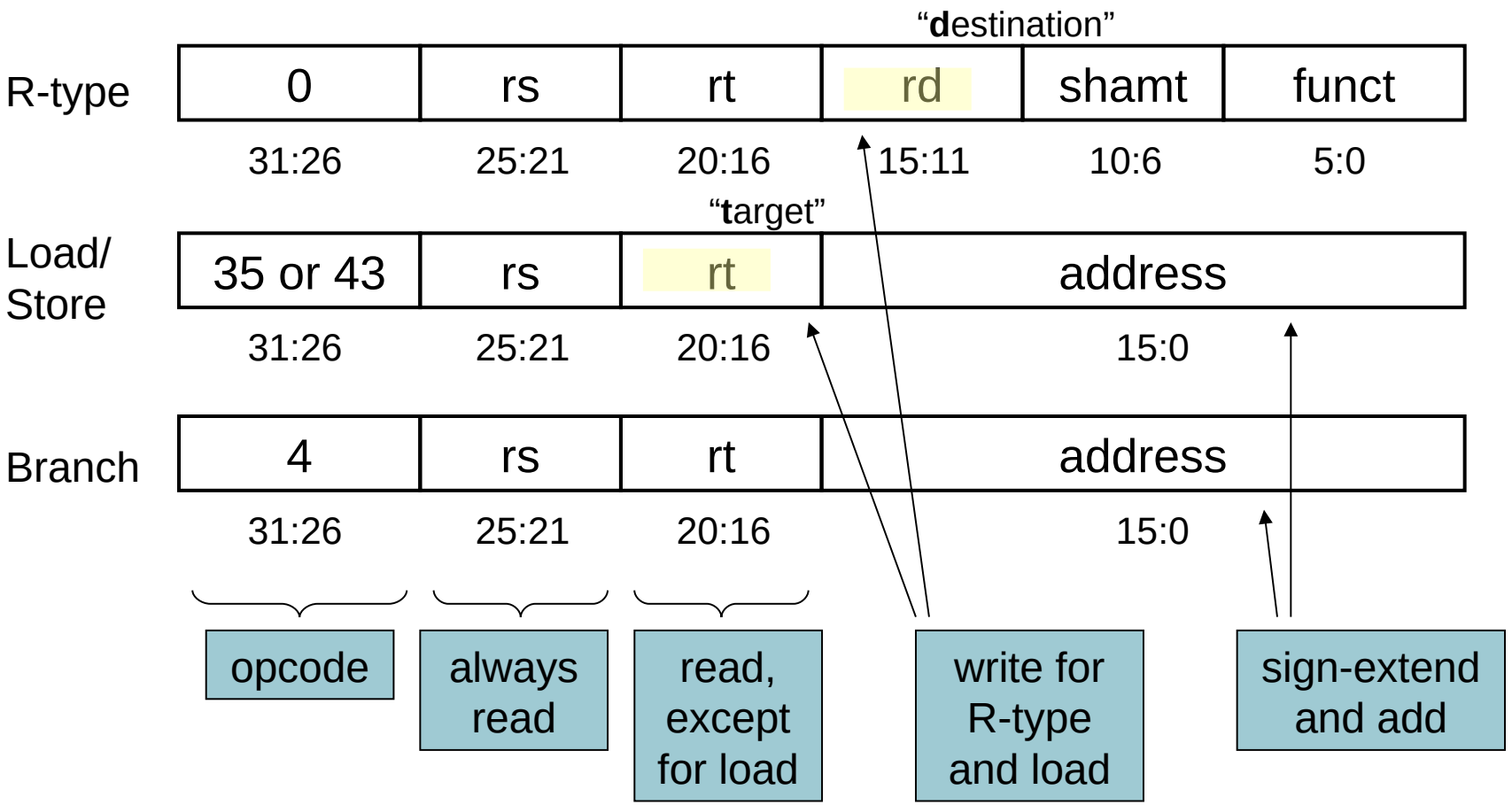
ALU Control

- 2-bit ALUOp derived from opcode
- Combinational logic for ALU control

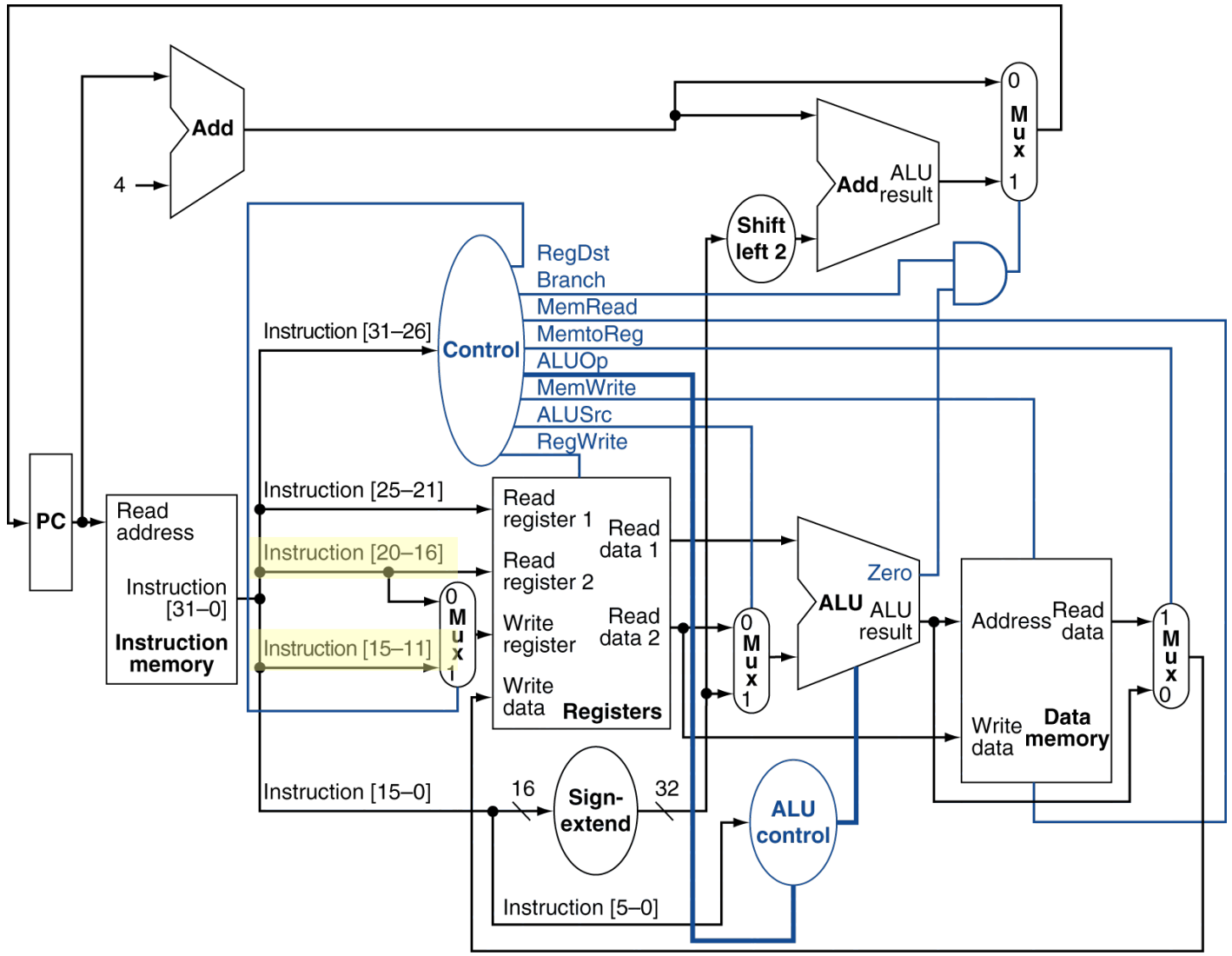
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		and	100100	and	0000
		or	100101	or	0001
		set-on-less-than	101010	set-**on**-less-than	0111

The Main Control Unit

- information extracted from instruction



Datapath With Control



Example program

```
1           .data
2
3   values: .word   10
4           .word   12
5   result: .word   9
6
7           .text
8   start:
9
10  # ALU operations
11          li   $t1, 1
12          li   $t2, 2
13          add  $t3, $t1, $t2
14
15  # memory operations
16          la   $t0, values
17          lw   $t1, 0($t0)
18          lw   $t2, 4($t0)
19          add  $t3, $t1, $t2
20          la   $t0, result
21          sw   $t3, 0($t0)
22
23  # control flow
24          beq  $t3, $t3, start
25          addi $t3, $t3, 2
26
27          j    start
```

Example program, assembled, executing

Text Segment

Bkpt	Address	Code	Basic	Source			
	0x00400000	0x24090001	addiu $9,$0,0x00000001	11:	li	$t1, 1	
	0x00400004	0x240a0002	addiu $10,$0,0x0000...	12:	li	$t2, 2	
	0x00400008	0x012a5820	add $11,$9,$10	13:	add	$t3, $t1, $t2	
	0x0040000c	0x3c011001	lui $1,0x00001001	16:	la	$t0, values	
	0x00400010	0x34280000	ori $8,$1,0x00000000				
	0x00400014	0x8d090000	lw $9,0x00000000($8)	17:	lw	$t1, 0($t0)	
	0x00400018	0x8d0a0004	lw $10,0x00000004($8)	18:	lw	$t2, 4($t0)	
	0x0040001c	0x012a5820	add $11,$9,$10	19:	add	$t3, $t1, $t2	
	0x00400020	0x3c011001	lui $1,0x00001001	20:	la	$t0, result	
	0x00400024	0x34280008	ori $8,$1,0x00000008				
	0x00400028	0xad0b0000	sw $11,0x00000000($8)	21:	sw	$t3, 0($t0)	
	0x0040002c	0x116bfff4	beq $11,$11,0xfffffff4	24:	beq	$t3, $t3, start	
	0x00400030	0x216b0002	addi $11,$11,0x0000...	25:	addi	$t3, $t3, 2	
	0x00400034	0x08100000	j 0x00400000	27:	j	start	

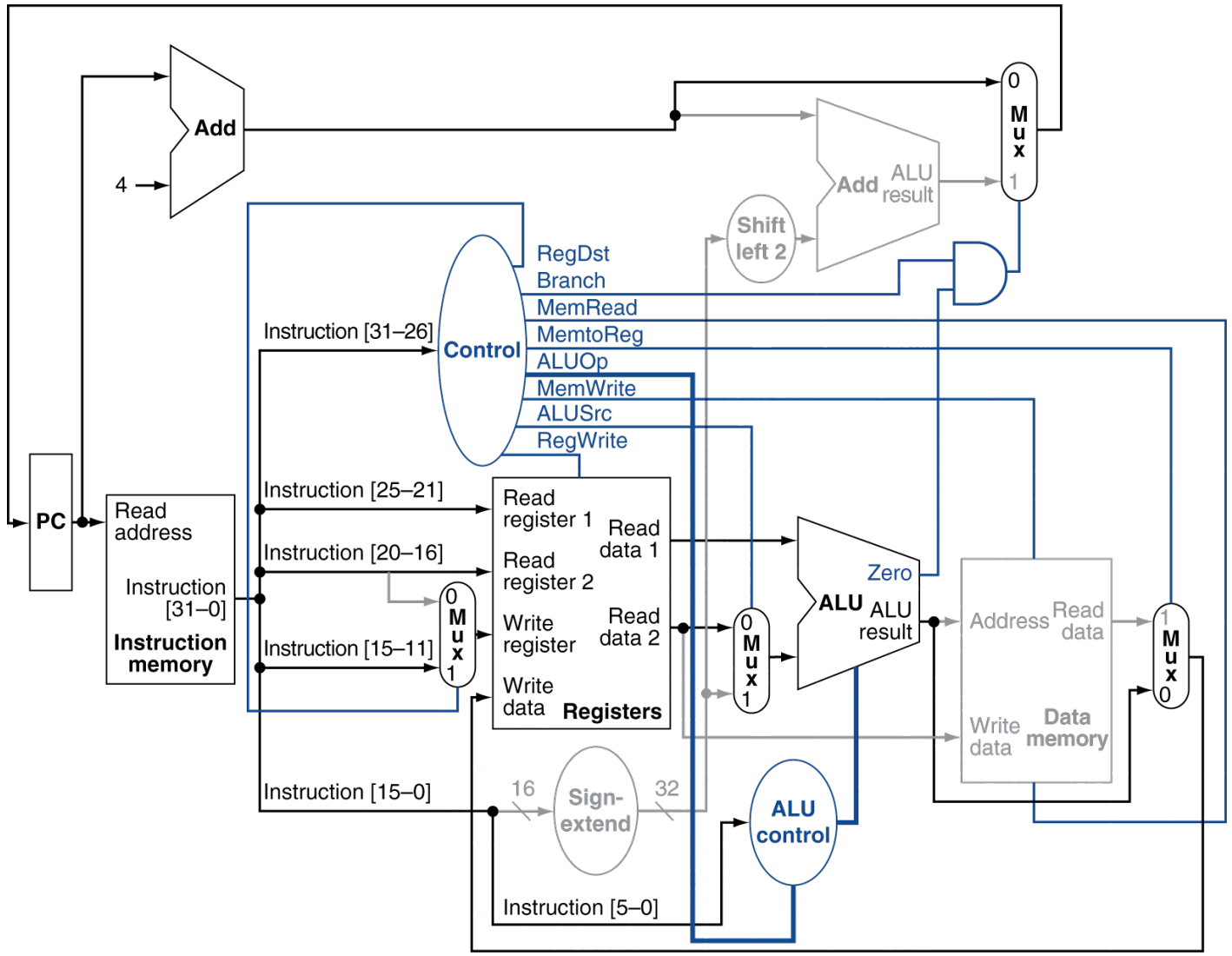
Labels

Label	Address ▲
tst.asm	
start	0x00400000
values	0x10010000
result	0x10010008

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	
0x10010000	0x0000000a	0x0000000c	0x00000009	0x00000000	0x00000000	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
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

R-Type Instruction




```
0x0040001c  0x012a5820  add $11,$9,$10        19:          add  $t3, $t1, $t2
```

R-Type Instruction encoding

0x0040001c	0x012a5820	add $11,$9,$10	19:	add $t3, $t1, $t2

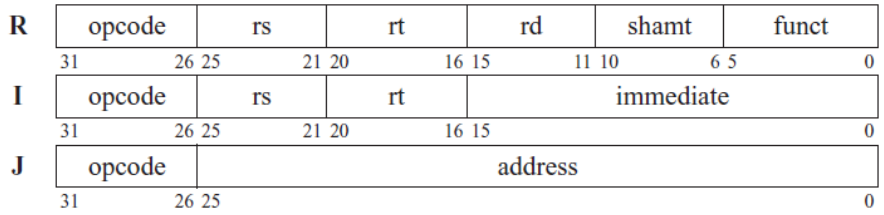
MIPS Reference Data

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}

(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



add opcode (in 6bits) = 00_{16} = 000000_2
add func (in 6 bits)= 20_{16} = 100000_2
add shamt (in 5 bits)= 00_{16} = 00000_2

$t1 = $9 = 01001_2
$t2 = $10 = 01010_2
$t3 = $11 = 01011_2

encoding of add $t3, $t1, $t2

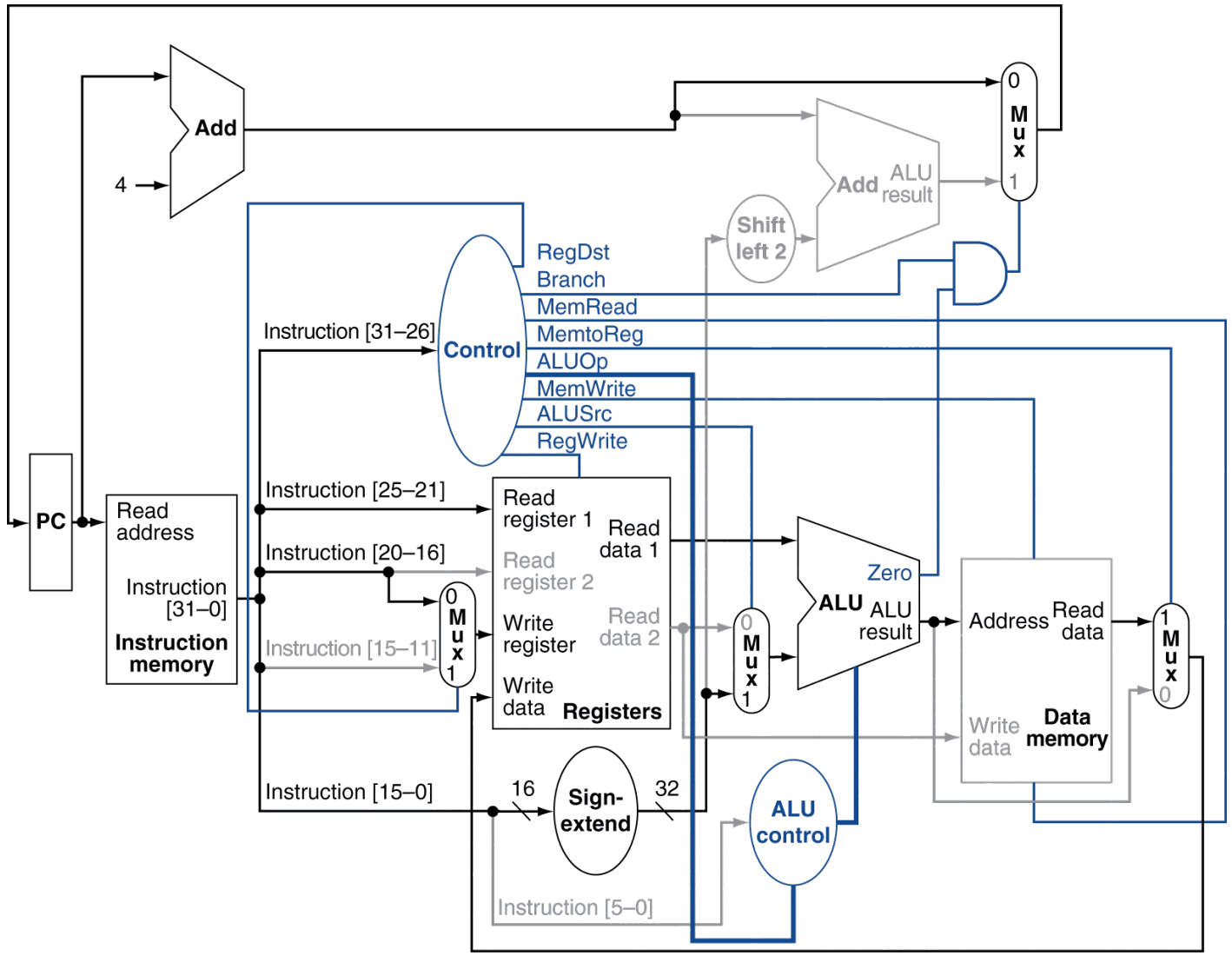
= 000000 01001 01010 01011 00000 100000_2

= 0000 0001 0010 1010 0101 1000 0010 0000_2

= 0 1 2 a 5 8 2 0_{16}

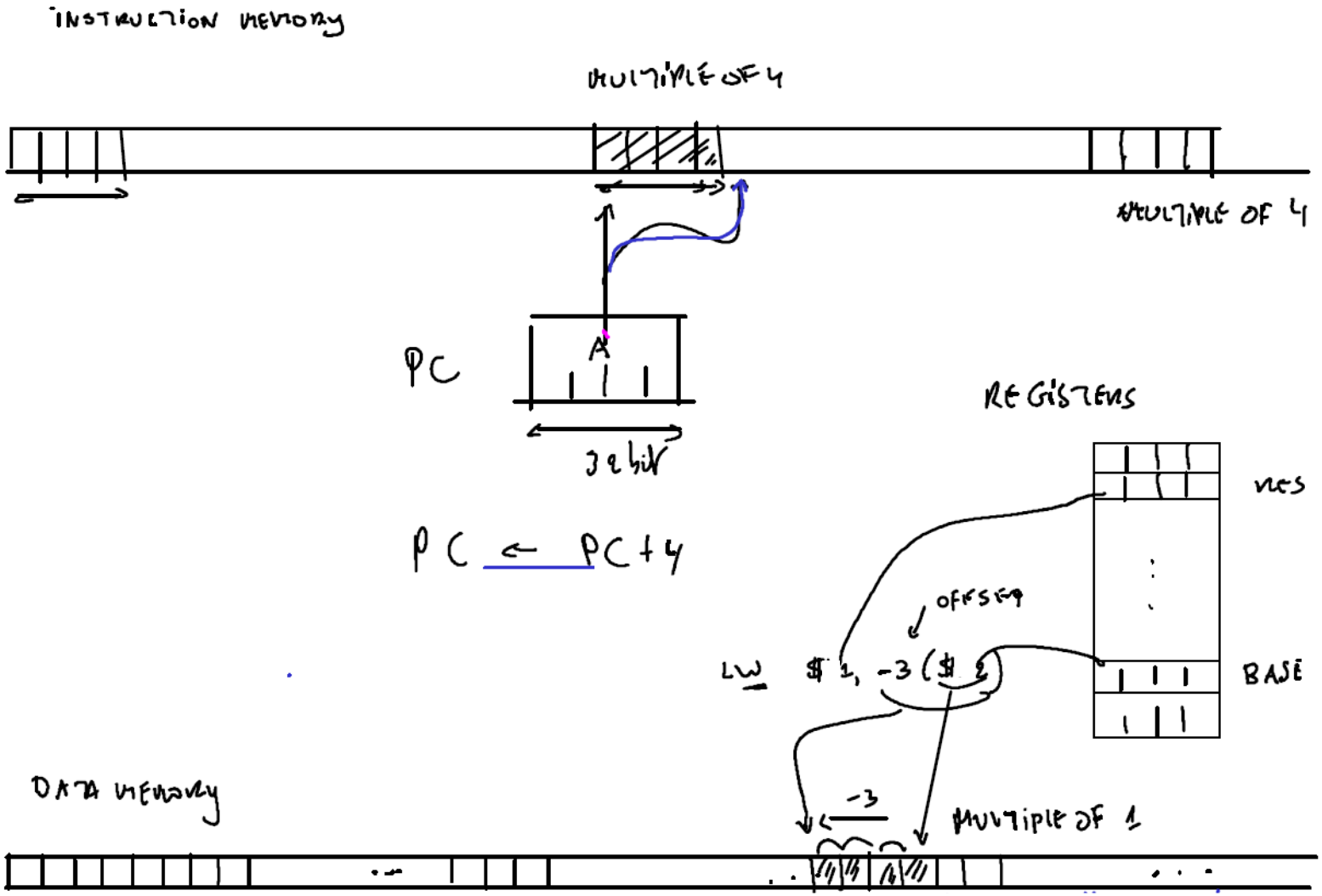
decoding (disassembling) of $012a5820_{16}$ ...

Load Instruction



0x00400018 | 0x8d0a0004 | lw $10,0x00000004($8) | 18: lw $t2, 4($t0)

Load Instruction



Load Instruction

```
lw dest_reg#, offset (base_reg#)
```

offset can be both positive and negative:
16 bit signed → range $[-2^{15}, \ldots, +2^{15}-1]$

base reg contains base address
→ data to be loaded
from data memory into dest reg
from address **R[**base_reg#**]** + offset

32 bit addresses, so sign extend offset

Load Instruction encoding

```
0x00400018  0x8d0a0004  lw $10,0x00000004($8)  18:          lw   $t2, 4($t0)
```

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	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}

(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

lw opcode (in 6 bits) = 23_{16} = 100011_2
$t0 (in 5 bits) = $8 = 01000_2
$t2 (in 5 bits) = $10 = 01010_2
4 (in 16 bits signed) = $0000\ 0000\ 0000\ 0100_2$

encoding of lw $t2, 4($t0)

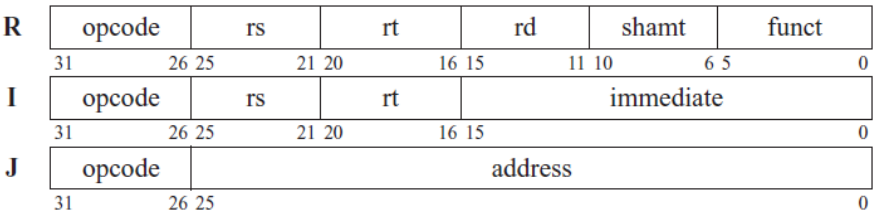
= $100011\ 01000\ 01010\ 0000\ 0000\ 0000\ 0100_2$

= $1000\ 1101\ 0000\ 1010\ 0000\ 0000\ 0000\ 0100_2$

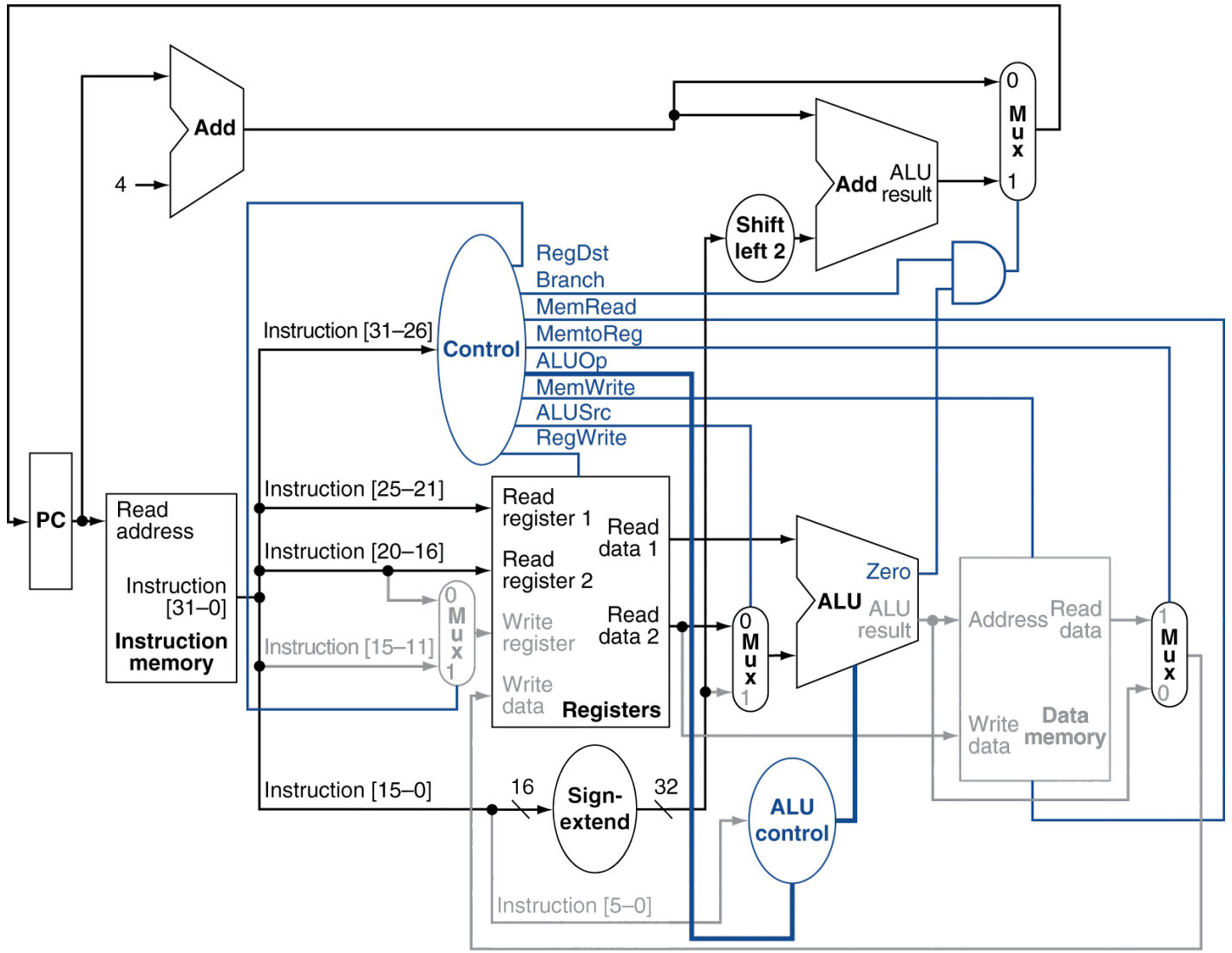
= $8\ d\ 0\ a\ 0\ 0\ 0\ 4_{16}$

decoding (disassembling) of $8d0a0004_{16}$ ...

BASIC INSTRUCTION FORMATS

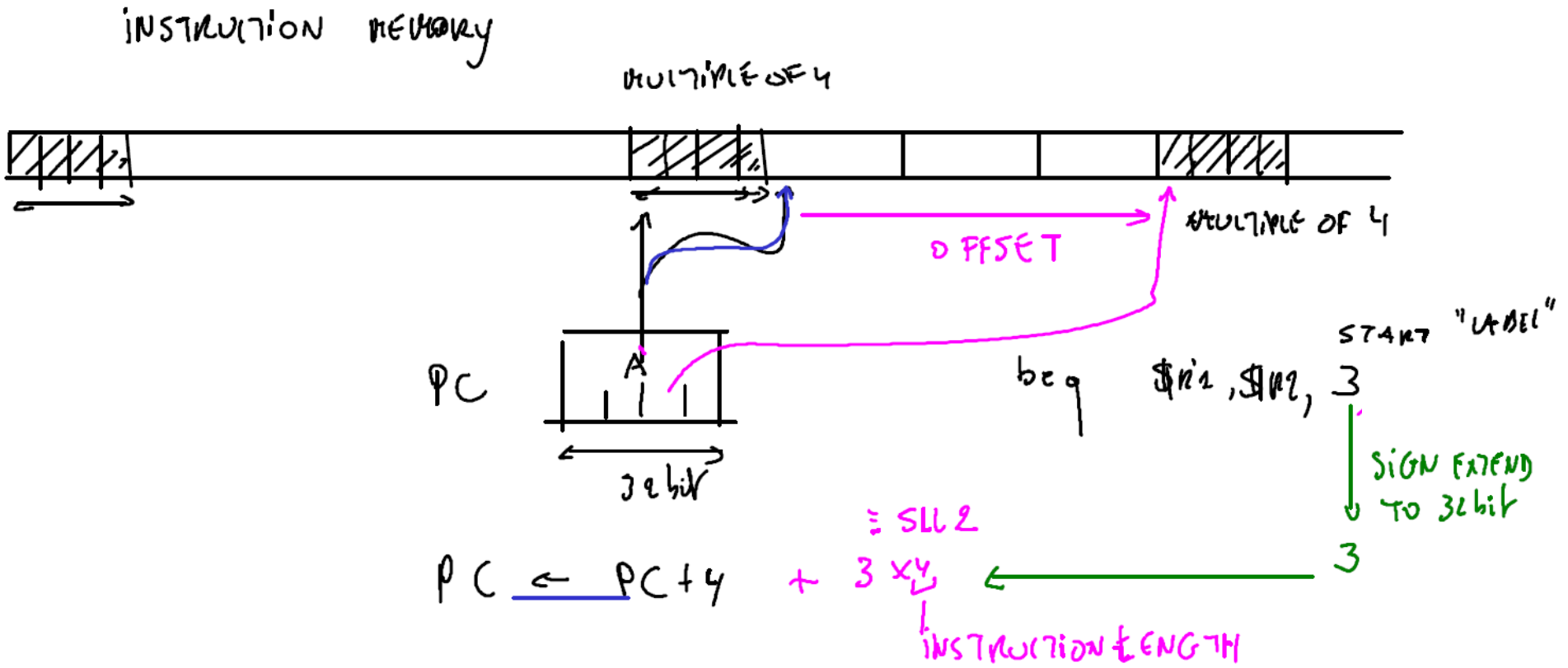


Branch-on-Equal Instruction




```
0x0040002c  0x116bfff4  beq $11,$11,0xfffffff4  24:          beq  $t3, $t3, start
```

Branch-on-Equal Instruction



4	rs	rt	address
31:26	25:21	20:16	15:0

beq Instruction encoding

0x0040002c	0x116bfff4	beq $11,$11,0xfffffff4	24:	beq $t3, $t3, start

Text Segment

Bkpt	Address	Code	Basic	Source		
	0x00400000	0x24090001	addiu $9,$0,0x00000001	11:	li	$t1, 1
	0x00400004	0x240a0002	addiu $10,$0,0x0000...	12:	li	$t2, 2
	0x00400008	0x012a5820	add $11,$9,$10	13:	add	$t3, $t1, $t2
	0x0040000c	0x3c011001	lui $1,0x00001001	16:	la	$t0, values
	0x00400010	0x34280000	ori $8,$1,0x00000000			
	0x00400014	0x8d090000	lw $9,0x00000000($8)	17:	lw	$t1, 0($t0)
	0x00400018	0x8d0a0004	lw $10,0x00000004($8)	18:	lw	$t2, 4($t0)
	0x0040001c	0x012a5820	add $11,$9,$10	19:	add	$t3, $t1, $t2
	0x00400020	0x3c011001	lui $1,0x00001001	20:	la	$t0, result
	0x00400024	0x34280008	ori $8,$1,0x00000008			
	0x00400028	0xad0b0000	sw $11,0x00000000($8)	21:	sw	$t3, 0($t0)
→	0x0040002c	0x116bfff4	beq $11,$11,0xfffffff4	24:	beq	$t3, $t3, start
	0x00400030	0x216b0002	addi $11,$11,0x0000...	25:	addi	$t3, $t3, 2
	0x00400034	0x08100000	j 0x00400000	27:	j	start

```
1          .data
2
3   values: .word   10
4           .word   12
5   result: .word   9
6
7           .text
8   start:
9
10  # ALU operations
11          li    $t1, 1
12          li    $t2, 2
13          add   $t3, $t1, $t2
14
15  # memory operations
16          la    $t0, values
17          lw    $t1, 0($t0)
18          lw    $t2, 4($t0)
19          add   $t3, $t1, $t2
20          la    $t0, result
21          sw    $t3, 0($t0)
22
23  # control flow
24          beq   $t3, $t3, start
25          addi  $t3, $t3, 2
26
27          j     start
```

start address in beq = PC + 4 + offset (**in words**: 4 bytes)

→ offset = -12_{10}

$$
\begin{aligned}
12_{10} &= \quad 0000\ 0000\ 0000\ 1100_2 \\
-12_{10} &= \quad 1111\ 1111\ 1111\ 0011_2 \\
&\quad +\ 0000\ 0000\ 0000\ 0001_2 \\
&= \quad 1111\ 1111\ 1111\ 0100_2 \\
&= \quad FFF4_{16}
\end{aligned}
$$

beq Instruction encoding

```
0x0040002c  0x116bfff4  beq $11,$11,0xfffffff4  24:        beq  $t3, $t3, start
```

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}

(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

beq opcode = 4_{16} = 000100_2
$t3 = $11 = 01011_2
offset = $FFF4_{16}$ = $1111\ 1111\ 1111\ 0100_2$

encoding of beq $t3, $t3, start

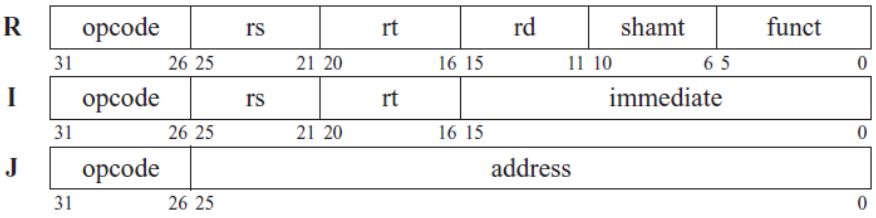
$= 000100\ 01011\ 01011\ 1111\ 1111\ 1111\ 0100_2$

$= 0001\ 0001\ 0110\ 1011\ 1111\ 1111\ 1111\ 0100_2$

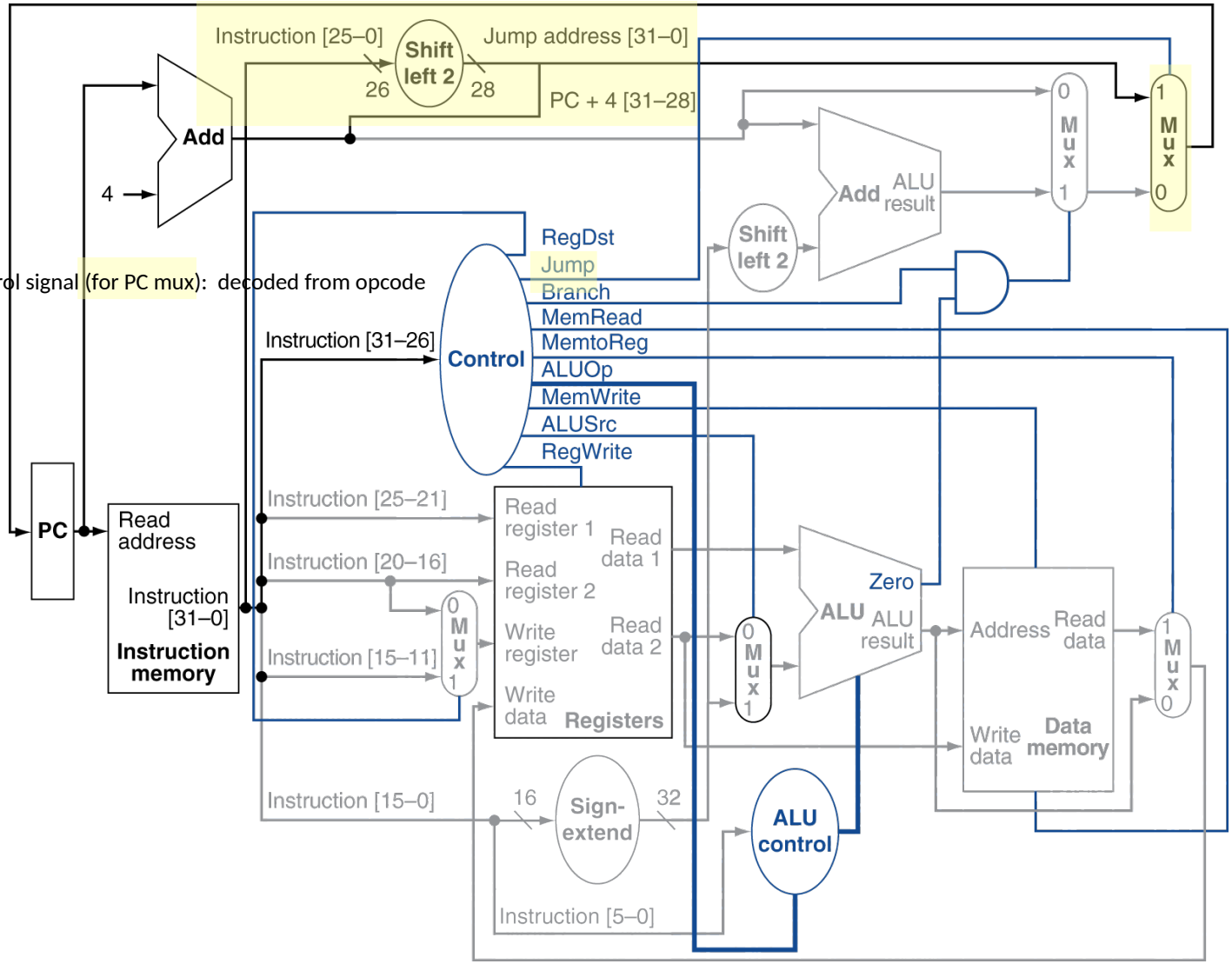
$= 1\ 1\ 6\ b\ f\ f\ f\ 4_{16}$

decoding (disassembling) of $116bfff4_{16}$ ...

BASIC INSTRUCTION FORMATS



Datapath With Jumps Added

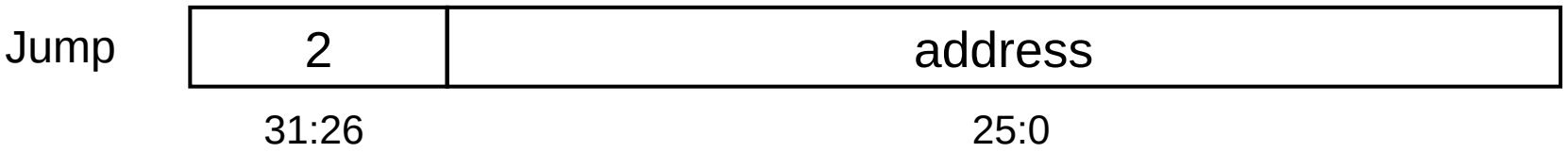


Need extra control signal (for PC mux): decoded from opcode

```
0x00400034  0x08100000  j 0x00400000          27:          j    start
```

Implementing Jumps

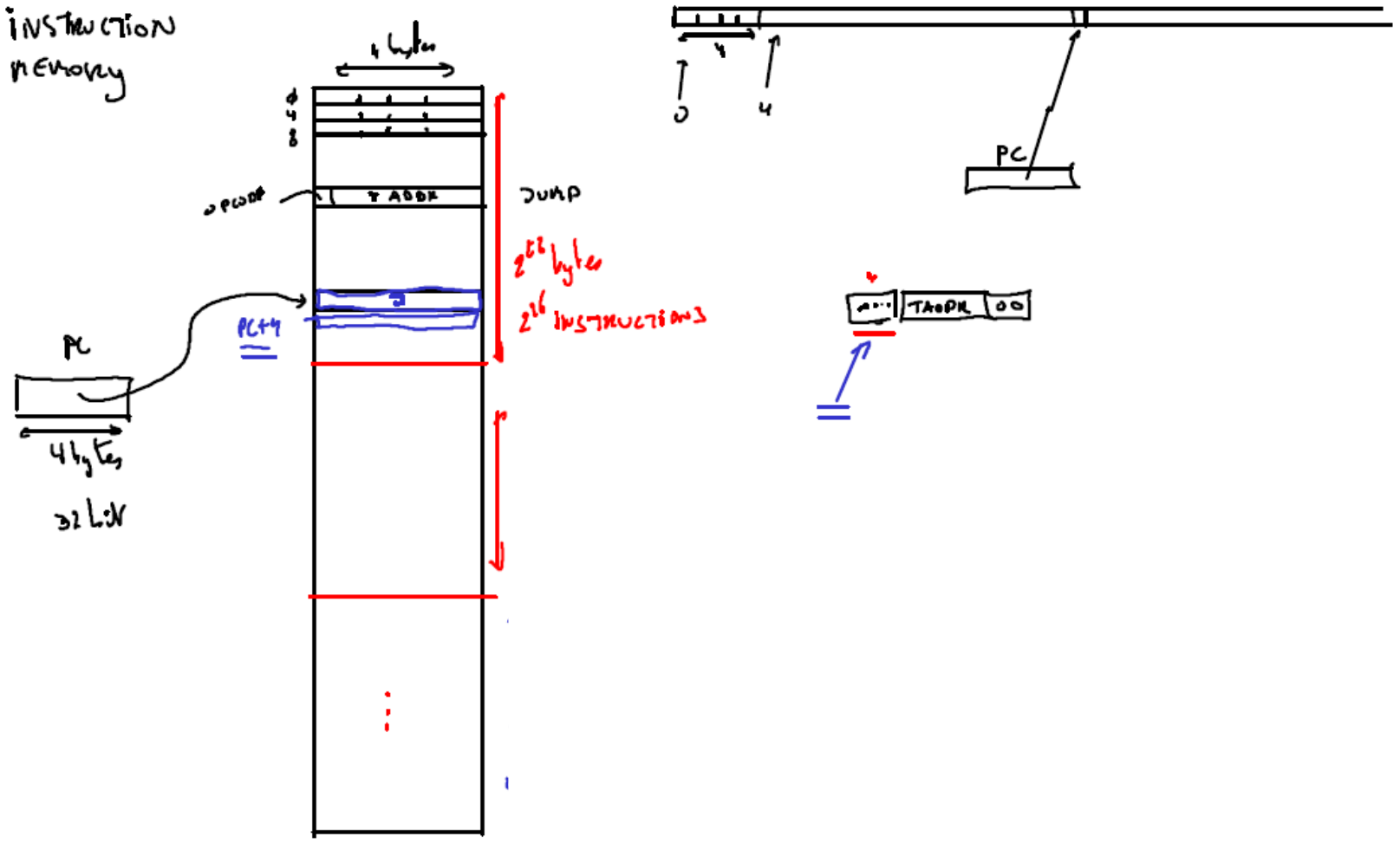
Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format



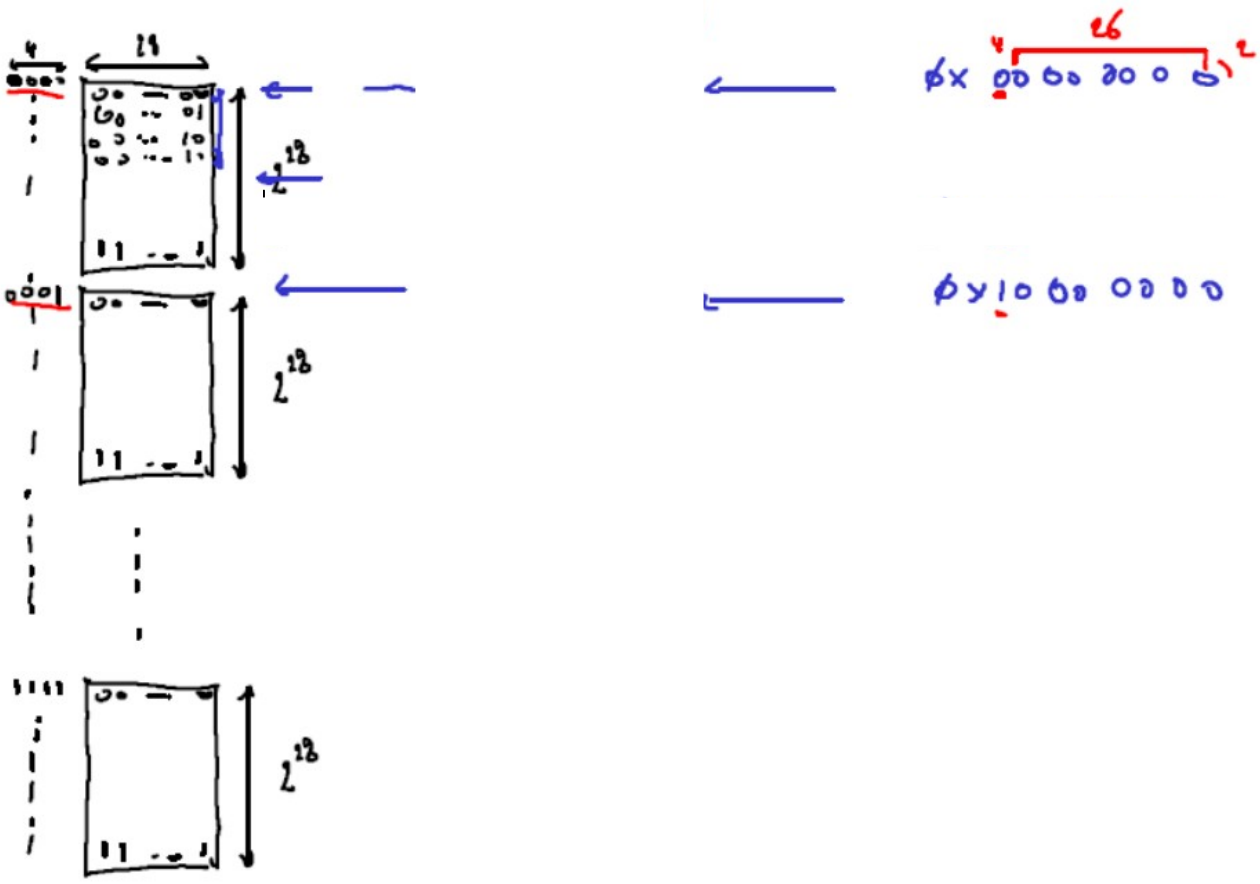
- Jump uses **word** address (not byte address)
- Update (32 bit) PC with concatenation of
  - **top 4 bits** of (**old PC + 4**)
  - 26-bit jump address
  - 00
- Need extra control signal (for PC mux): decoded from opcode

0x00400034	0x08100000	j 0x00400000	27:	j start

Implementing Jumps



Implementing Jumps



jump Instruction encoding

0x00400034	0x08100000	j 0x00400000	27:	j	start

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}

(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

j opcode = 2_{16} = 000010_2

PC = 004000034_{16}

PC + 4 = 004000038_{16}

top 4 bits of (PC + 4) = 0_{16} = 0000_2

target address
= 00400000_{16}
= $0000\ \mathbf{0000\ 0100\ 0000\ 0000\ 0000\ 0000\ 00}00_2$

26 bit address

encoding of j start

= 000010 0000 0100 0000 0000 0000 0000 00_2

= 0000 1000 0001 0000 0000 0000 0000 0000_2

= 0 8 1 0 0 0 0 0_{16}

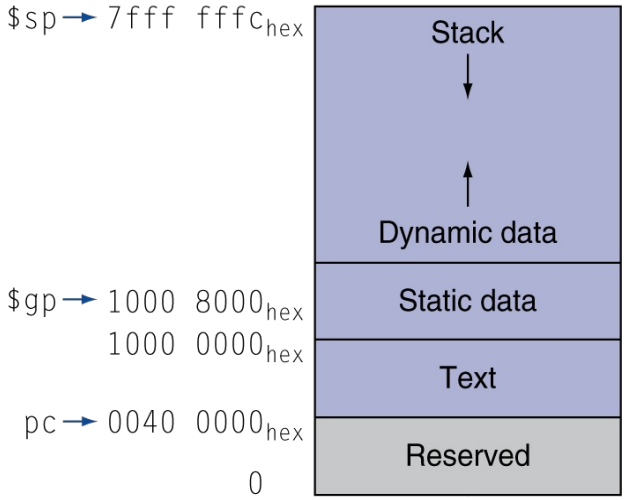
start:

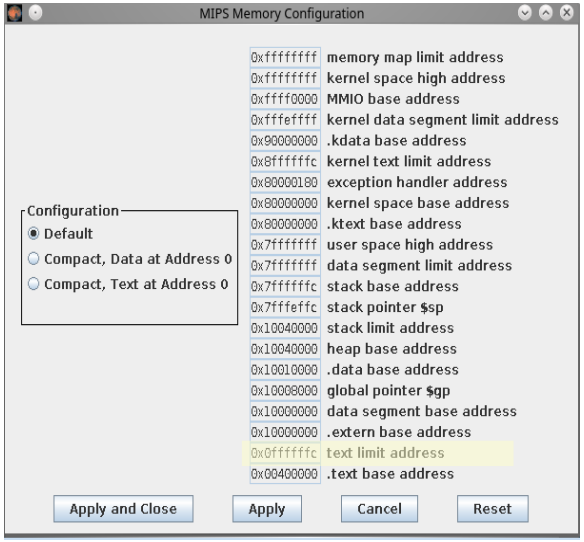
Text Segment

Bkpt	Address	Code	Basic		Source	
	0x00400000	0x24090001	addiu $9,$0,0x00000001	11:	li	$t1, 1
	0x00400004	0x240a0002	addiu $10,$0,0x0000...	12:	li	$t2, 2
	0x00400008	0x012a5820	add $11,$9,$10	13:	add	$t3, $t1, $t2
	0x0040000c	0x3c011001	lui $1,0x00001001	16:	la	$t0, values
	0x00400010	0x34280000	ori $8,$1,0x00000000			
	0x00400014	0x8d090000	lw $9,0x00000000($8)	17:	lw	$t1, 0($t0)
	0x00400018	0x8d0a0004	lw $10,0x00000004($8)	18:	lw	$t2, 4($t0)
	0x0040001c	0x012a5820	add $11,$9,$10	19:	add	$t3, $t1, $t2
	0x00400020	0x3c011001	lui $1,0x00001001	20:	la	$t0, result
	0x00400024	0x34280008	ori $8,$1,0x00000008			
	0x00400028	0xad0b0000	sw $11,0x00000000($8)	21:	sw	$t3, 0($t0)
	0x0040002c	0x116bfff4	beq $11,$11,0xfffffff4	24:	beq	$t3, $t3, start
	0x00400030	0x216b0002	addi $11,$11,0x0000...	25:	addi	$t3, $t3, 2
	0x00400034	0x08100000	j 0x00400000	27:	j	start

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
```

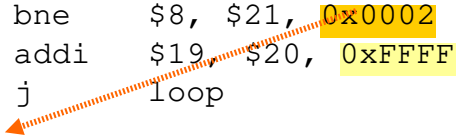
Address not allowed in MARS!

This is a synthetic example to demonstrate use of top 4 bits in address calculation in the `jump` 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  ???????
```

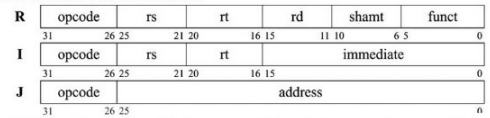
MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	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

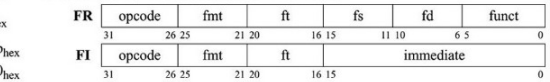
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	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



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 ???????
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
0x1C500854  …       26 bit
```

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)

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

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

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

disAssembling Example (j)

Decoding machine code (“disassembling”)

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

Opcode of Jump (**J**)

$1C500854_{hex}$ = **0001** 1100 0101 0000 0000 1000 0101 0100_{bin}

(26 bit) $314020F_{hex}$ = 11 0001 0100 0000 0010 0000 1111_{bin}

Jump target = **0001** 11 0001 0100 0000 0010 0000 1111 **00**$_{bin}$

= 0001 1100 0101 0000 0000 1000 0011 1100_{bin}

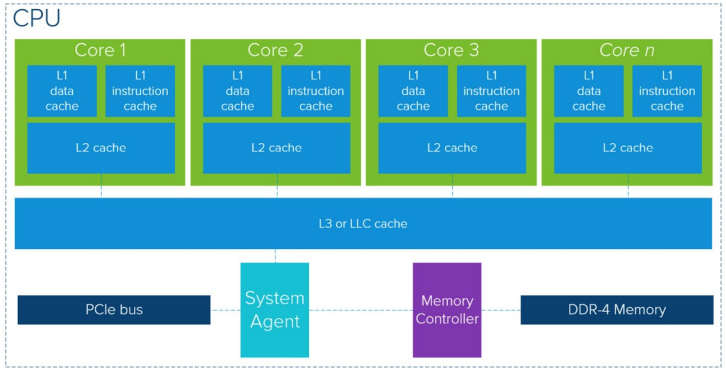
= 1 C 5 0 0 8 3 C_{hex}

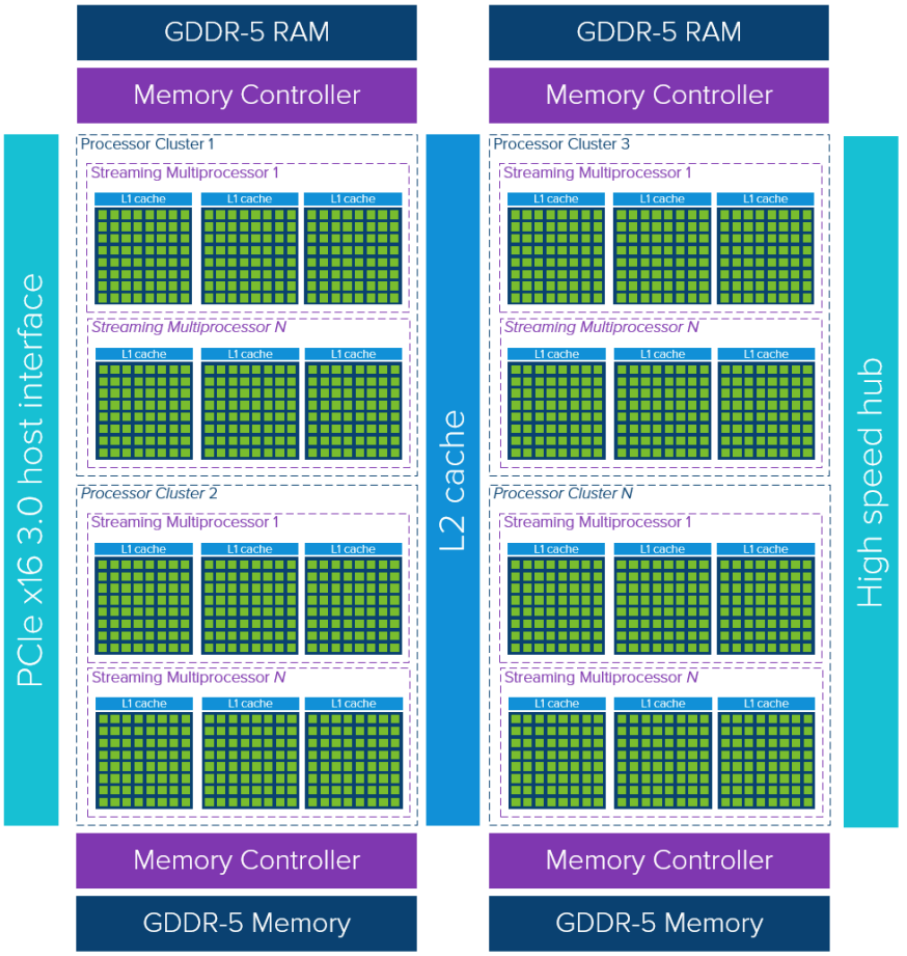
Performance Issues

- **Longest delay** determines **clock period**
  - **Critical** path: **load** instruction
  - Instruction memory → register file → ALU → data memory → register file
- Varying clock period for different instructions violates design principles:
  - **regularity**
  - **make the common case fast**
- Will improve performance by Instruction-Level Parallellism (ILP) aka “pipelining”
  (note that a constant clock period is needed for ILP)

FYI: Special Architectures

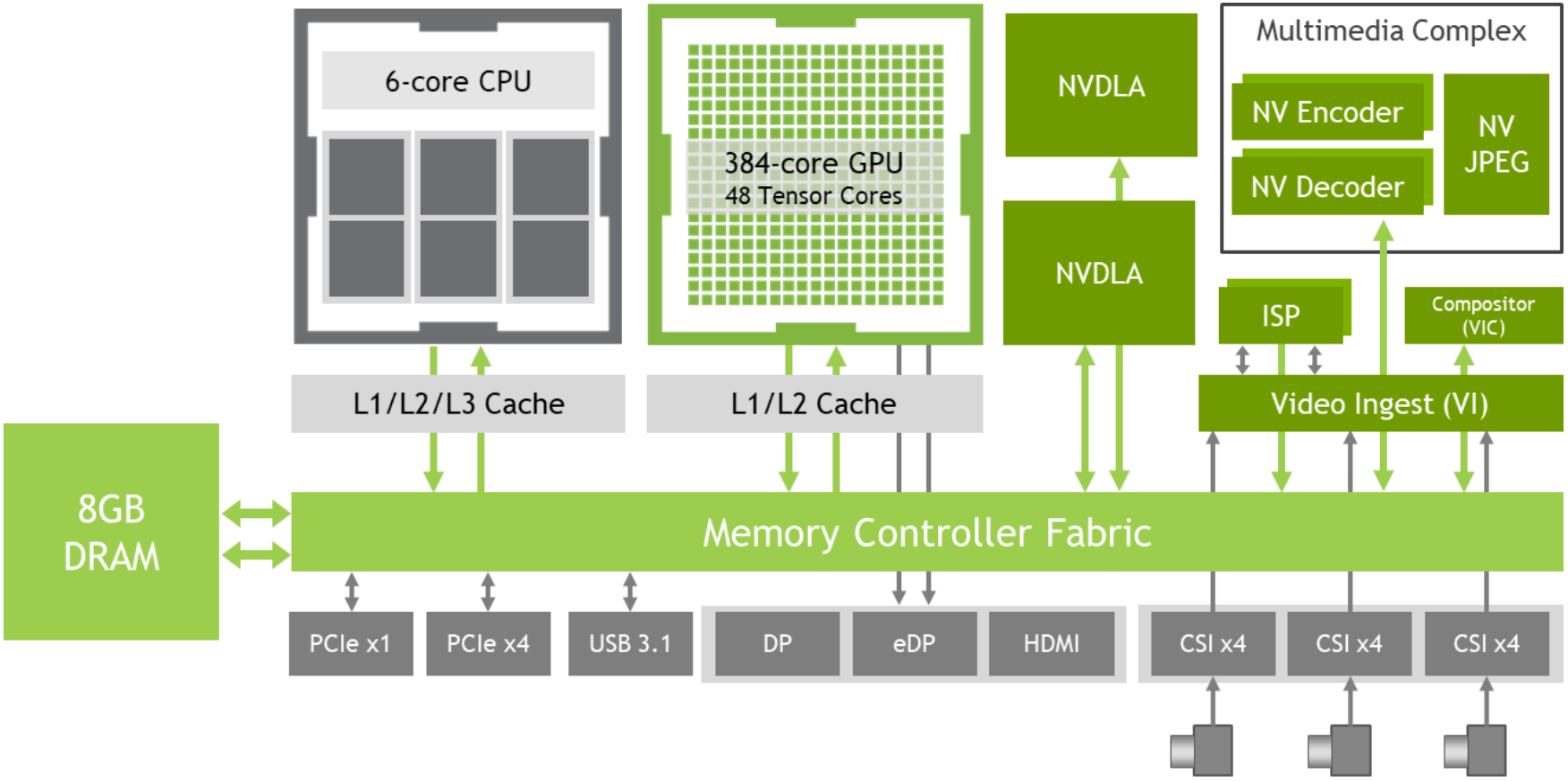
GPU (matrix operations … for graphics)
program using CUDA (Compute Unified Device Architecture) API





FYI: Special Architectures

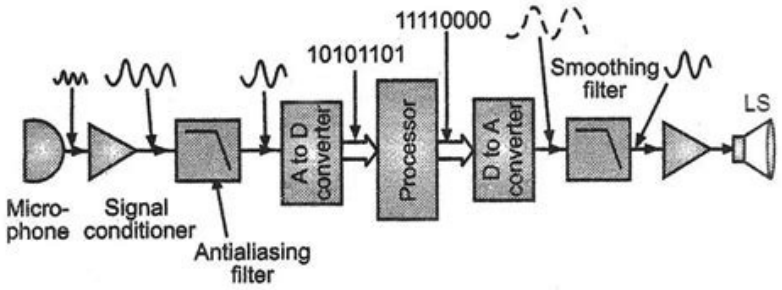
GPU (matrix operations … and AI)



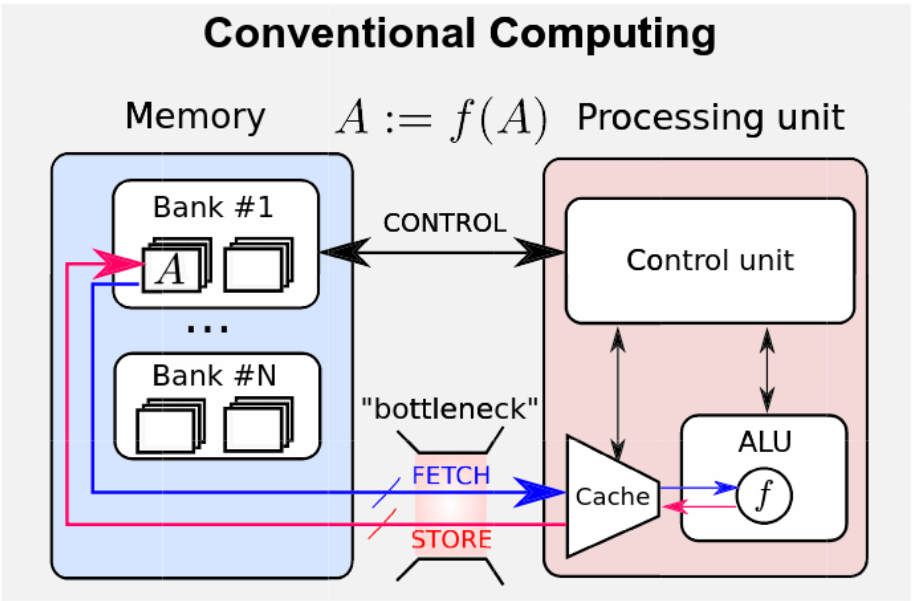
https://developer.nvidia.com/blog/jetson-xavier-nx-the-worlds-smallest-ai-supercomputer/

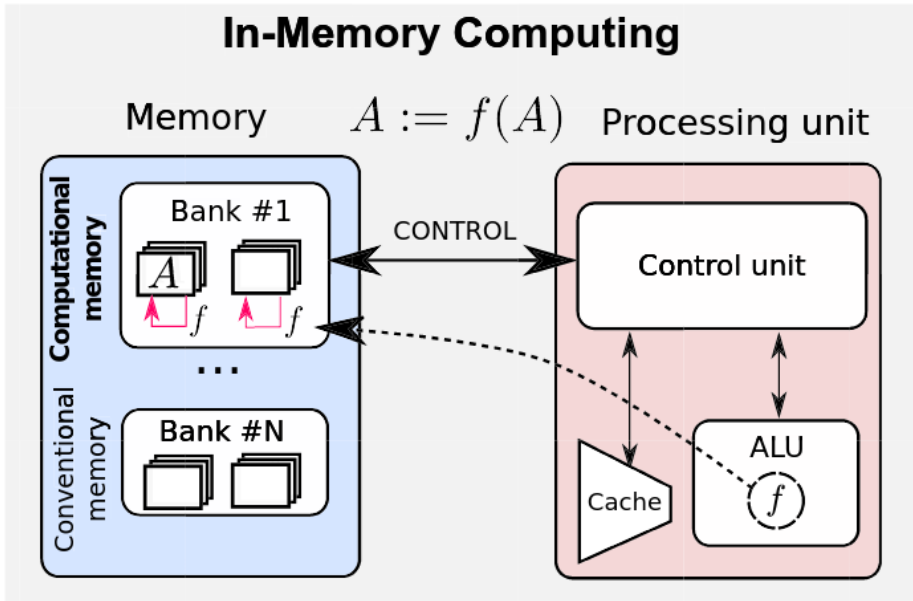
FYI: Special Architectures

Special purpose (signal processing, ECU, …)



In-Memory Computing





Manuel Le Gallo, Abu Sebastian, Evangelos Eleftheriou. In-Memory Computing: Towards Energy-Efficient Artificial Intelligence. ERCIM News 115, pp. 44 – 45, 2018.

Unconventional Computing:
Quantum Computing (and Reversible Computing), Analog Computing, ...