CS3350B Computer Organization
Chapter 3: CPU Control & Datapath
Part 1: Introduction to MIPS

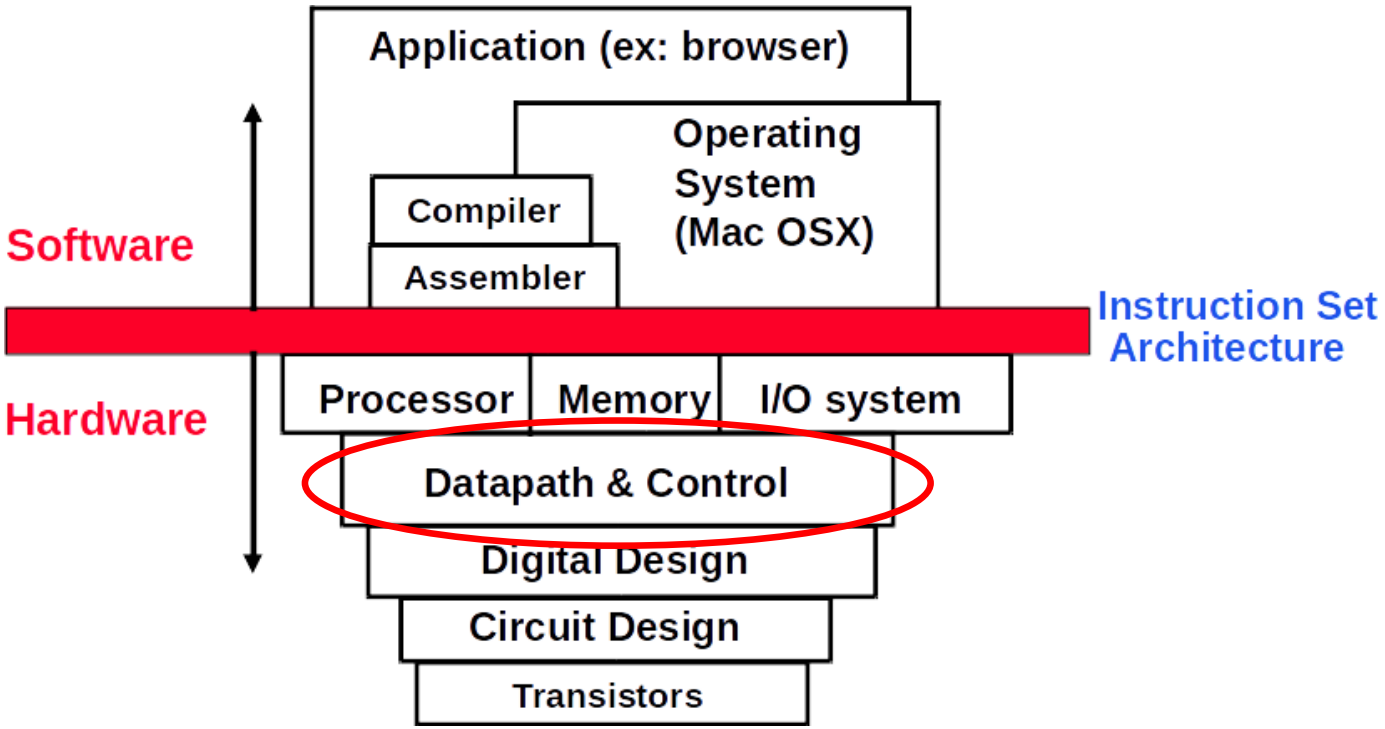
Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Thursday February 14, 2019

Outline

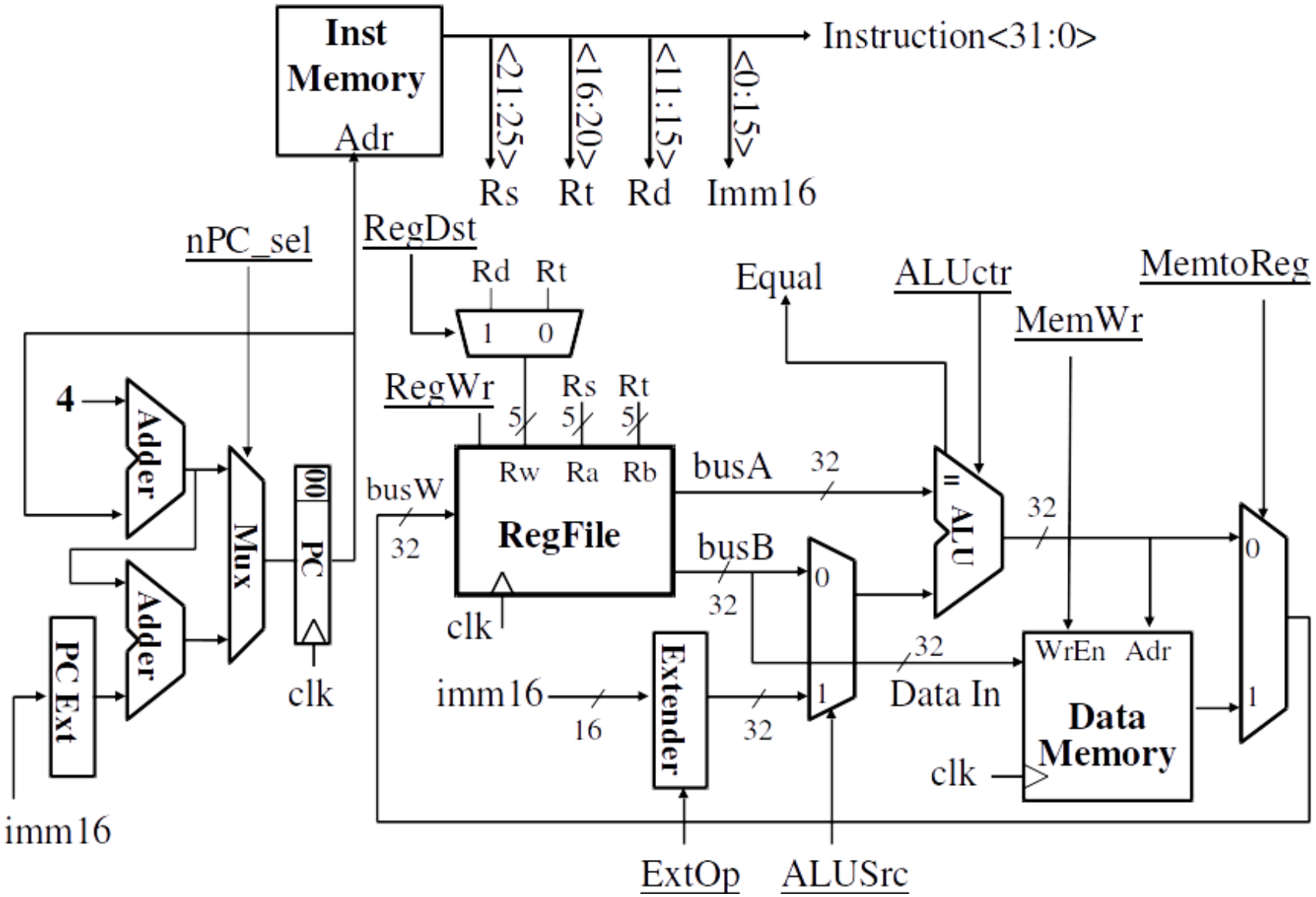
Layers of Abstraction



We will put together the combinational circuits and state circuits to see how a CPU actually works.

- How does data flow through the CPU?
- How are the path and circuits (MUX, ALU, registers) controlled?

Preview: MIPS Datapath



MIPS ISA

MIPS ISA: Microprocessor without Interlocked Pipelined Stages.

- ISA: The language of the computer.
- Microprocessor: a CPU.
- **Interlocking**: To come in chapter 4.
- **Pipelined Stages:** The data path is broken into a ubiquitous 5-stage pipeline (also chapter 4).

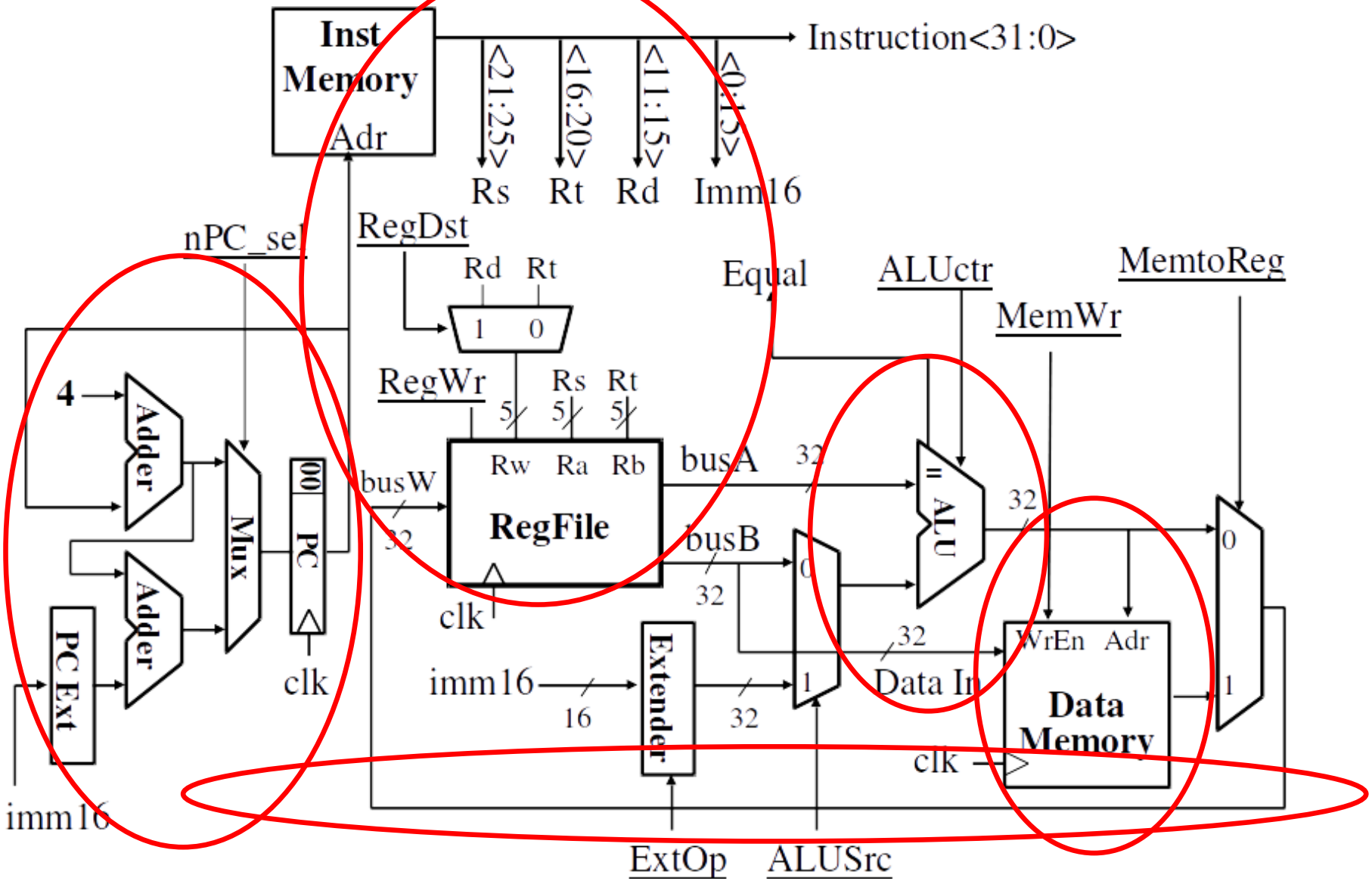
MIPS is a RISC ISA.

- **RISC:** Reduced Instruction Set Computer.
  - Provided instructions are simple, datapath is simple.
- Contrast with **CISC:** Complex Instruction Set Computer.
  - Instructions can be very "meta", each performing many lower-level instructions.

The 5-Stages of the Datapath

1. **IF**: Instruction Fetch
2. **ID**: Instruction Decode
3. **EX/ALU**: Execute/Arithmetic
4. **MEM**: Access Memory
5. **WB**: Write-back result

MIPS Datapath, Spot The Stages



Coupling of ISA and Datapath

A datapath must be built to satisfy the requirements of the ISA.

- Instructions in the ISA determine what is needed internally.
- Circuitry limit possible instructions.
- Built from combinational blocks composed together.
- Very hard to decouple the two.

We begin by looking at the MIPS ISA before looking at the datapath components.

↳ We need a common language to discuss the datapath and give concrete examples.

Layers of an ISA

Start at high-level and work down.

- MIPS assembly
- MIPS instruction formats
- MIPS instruction binary
  - ↳ *Everything* on a computer is a number
  - ↳ Recall instruction memory cache (banked L1 cache)

MIPS assembly is a type of **RTL:** Register transfer language.

- Everything in MIPS is specified by registers and movement between them or between registers and memory.
- Most often, we abstract away the concept of caches here and assume CPU talks directly with memory.
  - ↳ In reality, the circuity of the cache automatically abstracts away the memory hierarchy and handles cache hits/misses.

Outline

MIPS Assembly: The Basics

Registers:

- 32 general purpose 32-bit integer registers, denoted $0–$31.
- $0 always holds the value 0.
- $31 is reserved as the link register: stores the the point in instruction memory to return to after a function call.
- $PC holds the **program counter** – address of current instruction
- $HI & $LO store results of multiplication/division

Memory:

- 32-bit words and 32-bit memory addresses.
- Byte-addressable memory.
- Indexed like a big array of bytes: Mem[0], Mem[1024], Mem[32768].

MIPS Assembly: RTL Examples

3-Operand Arithmetic:

- add \$8 \$9 \$10 $\equiv$ R[8] $\leftarrow$ R[9] + R[10];
- sub \$8 \$9 \$10 $\equiv$ R[8] $\leftarrow$ R[9] - R[10];

2-Operand Arithmetic (Immediate Arithmetic):

- addi \$8 \$9 127 $\equiv$ R[8] $\leftarrow$ R[9] + 127;
- addi \$8 \$9 913 $\equiv$ R[8] $\leftarrow$ R[9] + 913;
- subi \$8 \$9 6 $\equiv$ R[8] $\leftarrow$ R[9] - 6;

Data Transfer (Memory Accesses):

- lw \$13 32(\$10) $\equiv$ R[13] $\leftarrow$ Mem[R[10] + 32];
- sw \$13 8(\$10) $\equiv$ Mem[R[10] + 8] $\leftarrow$ R[13];

MIPS Assembly: 3 Operand Arithmetic

`op $rd $rs $rt` $\equiv$ $\$rd = \$rs \text{ op } \$rt$

- $rd is the destination register.
- $rs is the (first) source register.
- $rt is the second source register.
- op is some arithmetic operation:
  - ↳ `add`, `addu`, `sub`, `subu`, `and`, `or`, `xor`, ...

These instructions *assume* an interpretation of the bits stored in the register.

- Programmer/compiler must choose proper instruction for data.
- add vs addu

MIPS Assembly: 2 Operand Arithmetic

$$\texttt{op \$rd \$rs imm} \quad \equiv \quad \$rd = \$rs \text{ op imma}$$

- \$rd is the destination register.
- \$rs is the (first) source register.
- `imm` is an **immediate**—a number whose value is hard-coded into the instruction.
  - ↳ C Example: int i = j + 12;
- op is some arithmetic operations:
  - ↳ `addi`, `addiu`, `subi`, `subiu`, `andi`, `ori`, `xori`, ...
  - ↳ `sll`, `srl` (logical shifts); `sla`, `sra` (arithmetic shifts)

These instructions *assume* an interpretation of the bits stored in the register.

- Programmer/compiler must choose proper instruction for data.
- Signed vs unsigned arithmetic. Logical vs arithmetic shifts.

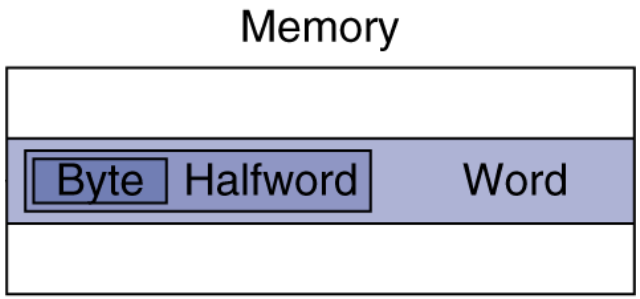
MIPS Assembly: Data Transfer

$$\texttt{lw \$rt, offset(\$rs)} \equiv \$rt = Mem[\$rs + offset]$$
$$\texttt{sw \$rt, offset(\$rs)} \equiv Mem[\$rs + offset] = \$rt$$

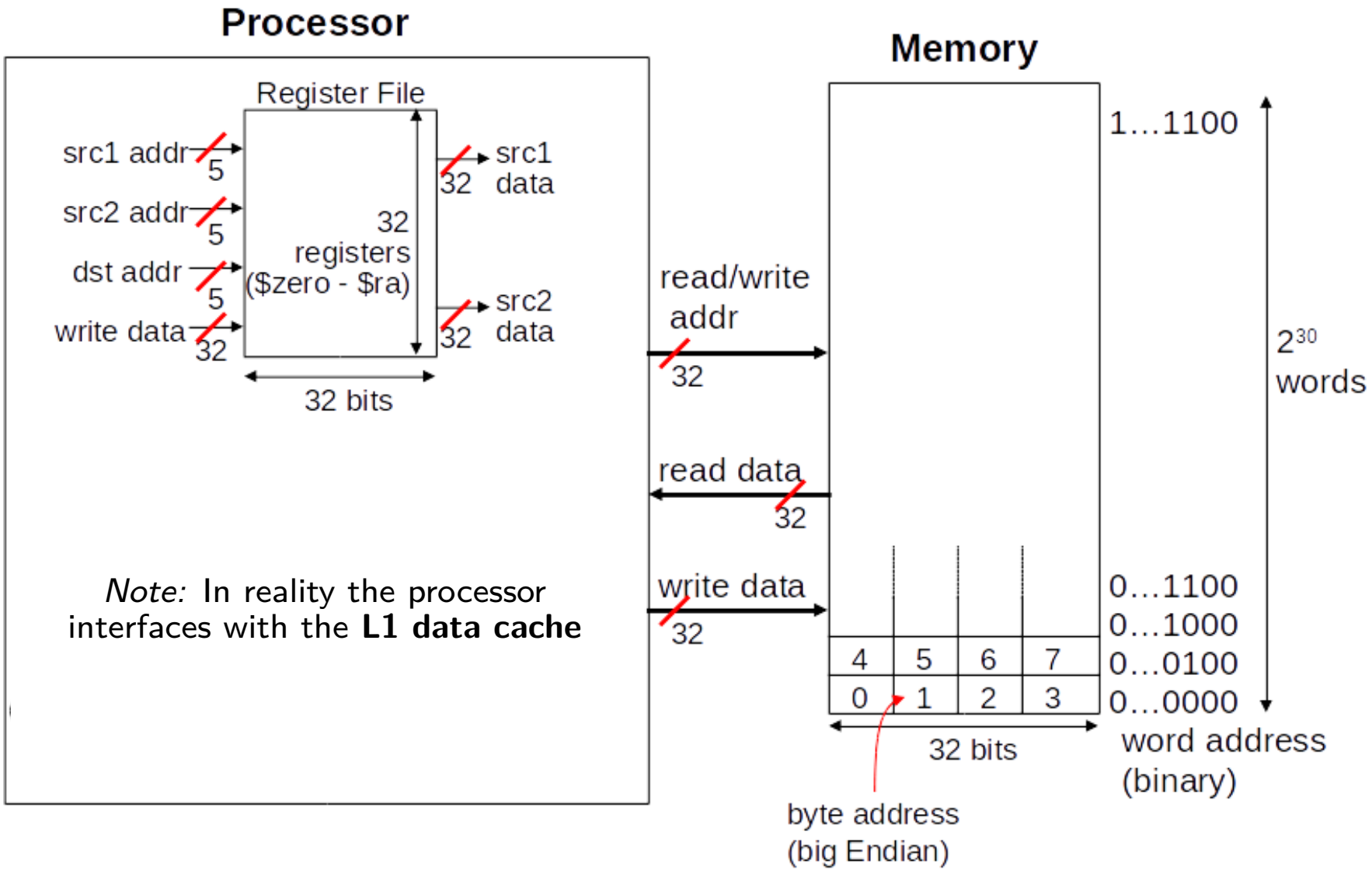
- \$rt is the "value" register.
- \$rs is the "address" register.
- `offset` is an **immediate**.

It is also possible to load and store bytes, **halfwords**:

- lb, sb (byte);
- lwr, swr (least-signficiant halfword);
- lwl, swl (most-significant halfword).



MIPS: A View of Memory



Note: In reality the processor interfaces with the **L1 data cache**

Aside: Endianness Defined

Endianness: The ordering of multiple bytes which are intended to be interpreted together as a single number.

- Important in memory layout, digital signals, networks, etc.

Consider the number: `0xAABBCCDD`

Little-Endian: The least-significant byte is stored/sent first.

- Ordering: `0xDD`, `0xCC`, `0xBB`, `0xAA`

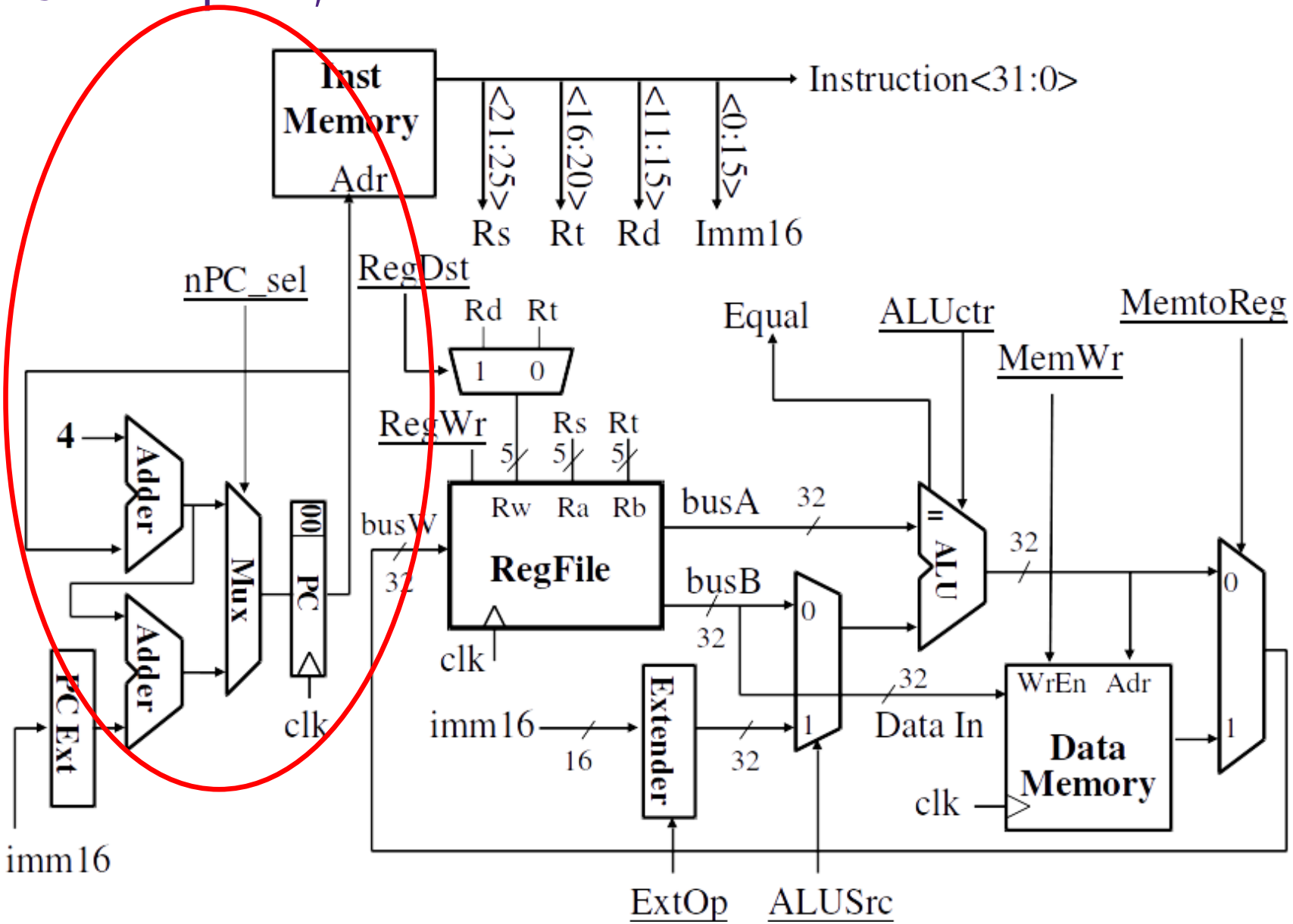
Big-Endian: The most-significant byte is stored/sent first.

- Ordering: `0xAA`, `0xBB`, `0xCC`, `0xDD`
- MIPS is big-endian.

Big-endian conceptually easier but little-endian has performance benefits. In reality, hardware handles all conversions to and from, so we *rarely* care.

Outline

MIPS Datapath, Instruction Fetch



Instruction Fetch

IF — Instruction Fetch

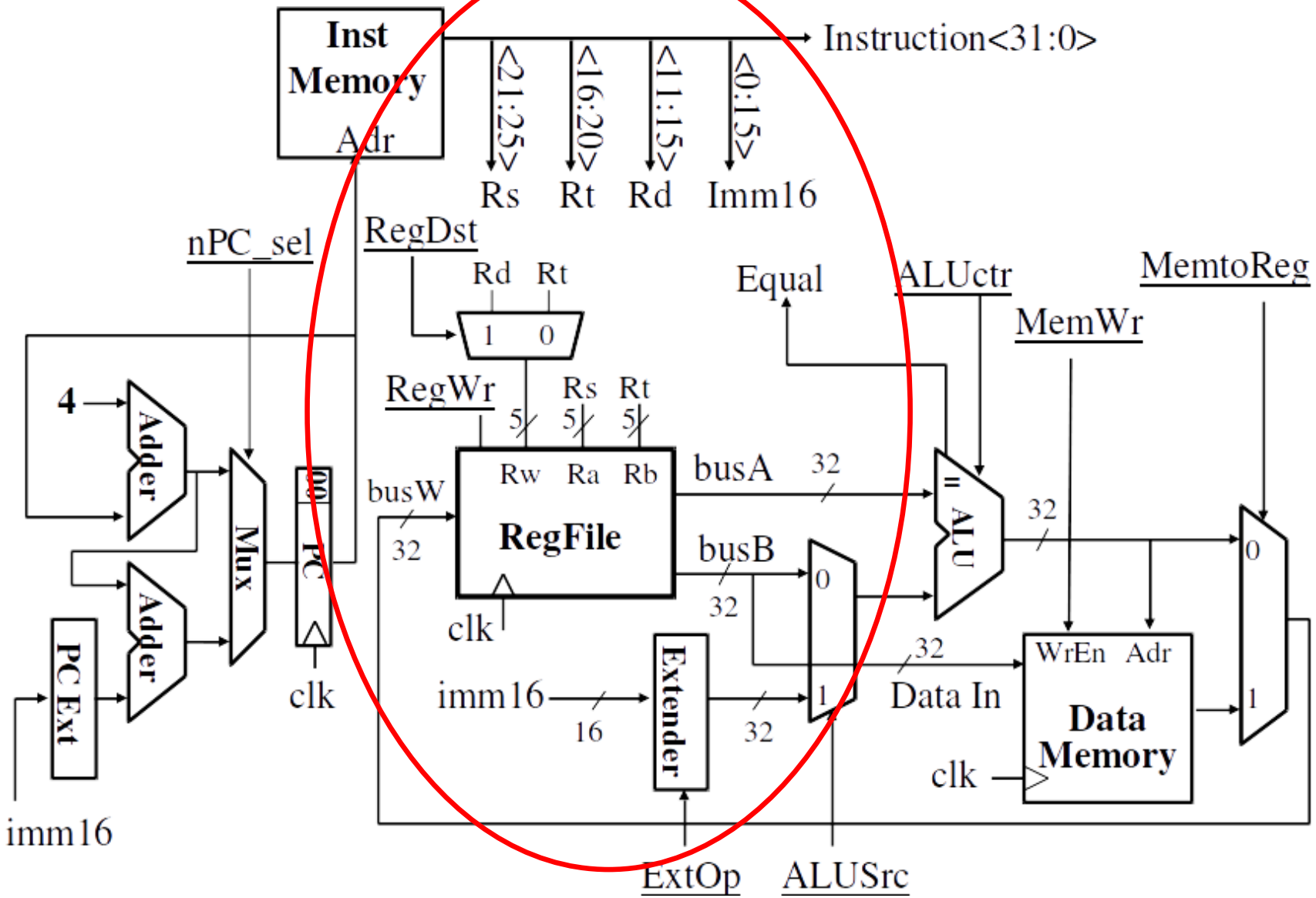
- The simplest part of the datapath.
- One job: fetch the next instruction to execute from memory.
- *Banked L1 cache* $\Longrightarrow$ separate cache just for instructions

1. On the clock's rising edge, update the value of **PC**—program counter.
   ↳ Essentially the index into instruction memory.

2. *Fetch* the instruction from memory and pass it to next stage: instruction decode.

3. Prepare for next instruction: calculate PC + 4 (4 bytes since instructions are word-aligned).

MIPS Datapath, Instruction Decode



Instruction Decode

ID — Instruction Decode

- Break the *binary value* of an instruction into its parts and decide what to do.
  - ↳ Recall: instructions eventually get compiled down to *bytecode* (i.e. binary).
- Get the values ready for arithmetic: registers, immediates.

1. Break the instruction into individual bit segments.
2. Access operand values from registers.
3. Extend immediate to 32 bits (if using).

But how to break the instruction?

Aside: MIPS Special Register Names

- $zero: the zero-valued register ($0)
- $at: reserved for compiler ($1)
- $v0, $v1: result values ($2, $3)
- $a0 - $a3: arguments ($4–$7)
- $t0 - $t9: temporaries ($8–$15, $24, $25)
  - ↳ Can be overwritten by callee
- $s0 - $s7: saved ($16–$23)
  - ↳ Must be saved/restored by callee
- $gp: global pointer for static data ($28)
- $sp: stack pointer ($29)
- $fp: frame pointer ($30)
- $ra: return address ($31)

Outline

MIPS Instruction Formats

Every instruction in MIPS is 32-bits.

- A memory word is 32 bits, after all.

All instructions belong to 3 pre-defined formats:

- **R-Type**: "Register"
- **I-Type**: "Immediate"
- **J-Type**: "Jump"

Each format defines how those 32 bits of instruction data are broken up into individual "bit-fields" and how they are interpreted during ID stage.

- The first 6 bits always encode the **opcode**.
- The opcode determines the type of instruction and format of the remaining bits.

R-Type Instructions

R-Type instructions usually have 3 registers as its operands.

- ↳ "Register type".
- ↳ General arithmetic operations.

- op — the opcode.
- rs — first source register.
- rt — second source register.
- rd — destination register.
- shamt — shift amount; used for shift instructions, 0 otherwise.
- funct — the arithmetic *function* the ALU should perform.

R-Type Examples 1

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

```
add $t0, $s1, $s2
```

op	$s1	$s2	$t0	shamt	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

```
sub $t0, $s1, $s2
```

op	$s1	$s2	$t0	shamt	sub
0	17	18	8	0	34
000000	10001	10010	01000	00000	100010

- For R-Type instructions the opcode and funct *together* determine the operations to perform.

R-Type Examples 2

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

```
sll $s0, $t0, 4
```

op	rs	$t0	$s0	4	shift left
0	0	8	16	4	0
000000	00000	01000	10000	00100	000000

Note: shift instruction have two registers and an immediate, but are *not* immediate instructions.

- Here, the allowed value of the shift amount is only 5 bits, not 16 bits as in an immediate-type instruction.

R-Type Examples 3: Bit-Wise Logical Operations

- Useful to mask (remove) 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

- 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

I-Type Instructions

I-Type instructions always have 2 registers and an **immediate**.

- "Immediate type".
- Immediate arithmetic, data transfer, branch.

- op — the opcode.
- rs — first source register.
- rt — second source (or destination) register.
- imm — the immediate/constant.

I-Type Examples 1

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

```
addi $t1, $t0, 10
```

op	rs	rt	immediate
8	$t0	$t1	10
001000	01000	01001	0000000000001010

```
addiu $t1, $t0, 10
```

op	rs	rt	immediate
9	$t0	$t1	10
001001	01000	01001	0000000000001010

Note: unsigned instructions will *not* signal exception on overflow.

I-Type Examples 2

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

```
lw $t1, 12($t0)
```

op	rs	rt	immediate
35	$t0	$t1	12
100011	01000	01001	0000000000001100

```
sw $t1, 32($t0)
```

op	rs	rt	immediate
43	$t0	$t1	32
101011	01000	01001	0000000000100000

I-Type Examples 3

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

```
bne $t0, $t1, 24
```

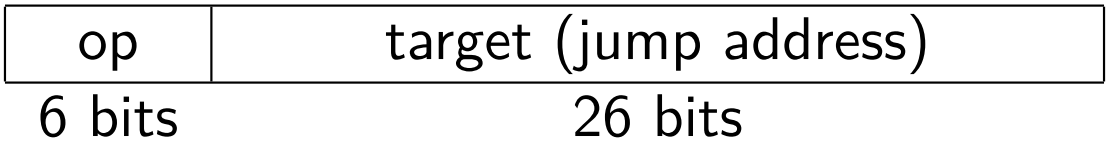
```
if ($t0 != $t1)
    PC = PC + 4 + (24 << 2);
else
    PC = PC + 4;
```

op	rs	rt	immediate
5	$t0	$t1	24
000101	01000	01001	0000000000110000

J-Type Instructions

J-Type instructions have just one big immediate, called a **target**.

- "Jump type".
- Only two instructions: `j` (jump) and `jal` (jump and link).



- op — the opcode.
- target — the target memory address to jump to.

Note: target is always multiplied by 4 before being applied to program counter...

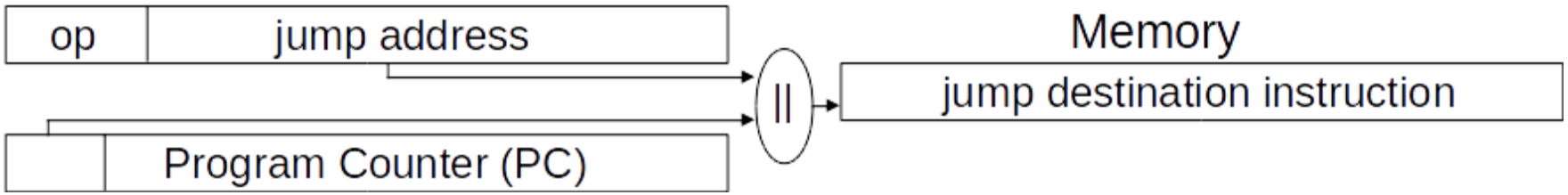
J-Type Instructions and Pseudo-Direct Addressing

Pseudo-Direct Addressing: Almost a direct addressing of instruction memory.

- Compiler usually handles the calculation of the exact jump target.

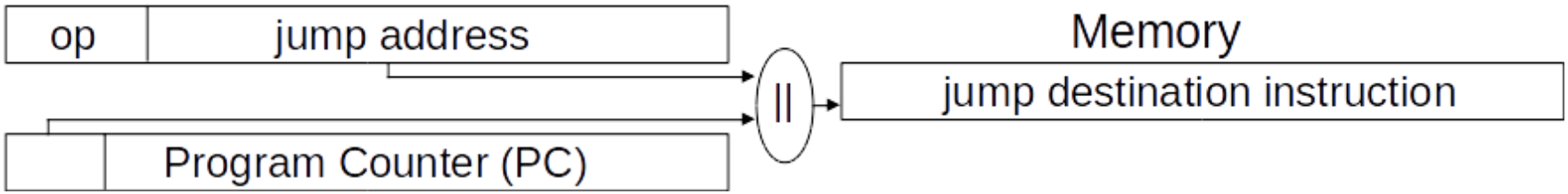
Next value of PC is target $\times$ 4 combined with upper 4 bits of current PC.

```
nPC = (PC & 0xf0000000) | (target << 2);
```

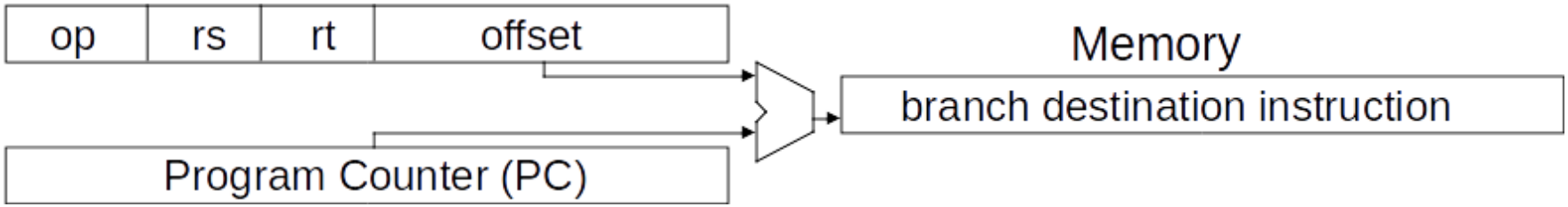


Addressing Instruction Memory in MIPS

- **Pseudo-Direct:** J-Type instructions



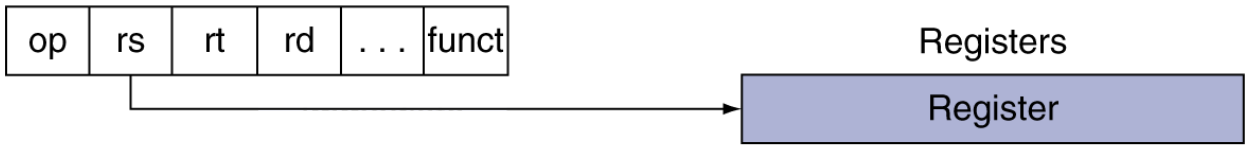
- **PC-Relative:** Branch instructions



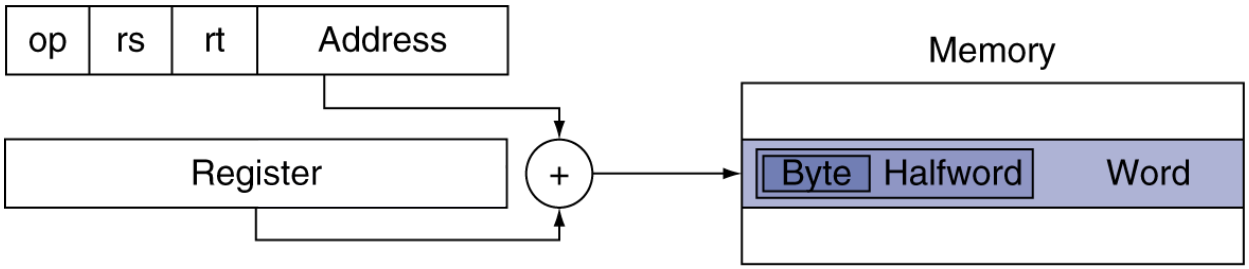
Addressing Operands in MIPS

- **Immediate Addressing**, I-Type instruction.

- **Register Addressing**, Almost all instructions.



- **Base Addressing**, Data transfer instructions.



MIPS ISA: Some Important Instructions

Category	Instruction		OP/ funct	Example	Meaning
Logic & Arith.	add	R	0/32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	R	0/34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	I	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	and/or	R	0/(36/37)	(and/or) $s1, $s2, $s3	$s1 = $s2 ($\wedge$/$\vee$) $s3
	(and/or) immediate	I	12/13	(andi/ori) $s1, $s2, 6	$s1 = $s2 ($\wedge$/$\vee$) 6
	shift right logical	R	0/2	srl $rt, $rd, 4	$rd = $rt >> 4
	shift right arithmetic	R	0/3	sra $rt, $rd, 4	$rd = $rt >> 4
Data Transfer	load word	I	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	I	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	I	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	I	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
Cond. Branch	br on equal	I	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	I	5	bne $s1, $s2, L	if ($s1 != $s2) go to L
	set less than	R	0/42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set less than immediate	I	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump	jump	J	2	j 250	go to 1000
	jump register	R	0/8	jr $t1	go to $t1
	jump and link	J	3	jal 250	go to 1000; $ra=PC+4

Note: knowing the binary values of each bit-field is not neccesary, but understanding the semantic meaning of each instruction *is* important.

Full Method Example: C to MIPS

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

```
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
```

Note: words and `int`-type are both 32-bits here.

Outline

Revisiting Program Basics

- **Frame:** The encapsulation of one method call; arguments, local variables.
  - ↳ "Enclosing subroutine context".
- (Call) **Stack** (of frames): The stack of method invocations.
  - ↳ Base of stack is the main method, each method call adds a frame to the stack.
- **Heap**: globally allocated data that lives beyond the scope of the frame in which it was allocated.
- **Static Data:** Global data which is stored in a *static* memory address throughout life of program.

MIPS Special Registers

- $v0, $v1: result values ($2, $3)
- $a0 - $a3: arguments ($4–$7)
- $t0 - $t9: temporaries ($8–$15, $24, $25)
  - ↳ Can be overwritten by callee
- $s0 - $s7: saved ($16–$23)
  - ↳ Must be saved/restored by callee
- $gp: global pointer for static data ($28)
- $sp: stack pointer ($29)
- $fp: frame pointer ($30)
  - ↳ The stack pointer *before* the current frame's invocation.
- $ra: return address ($31)

Memory Layout in MIPS (and most languages)

Text: program code

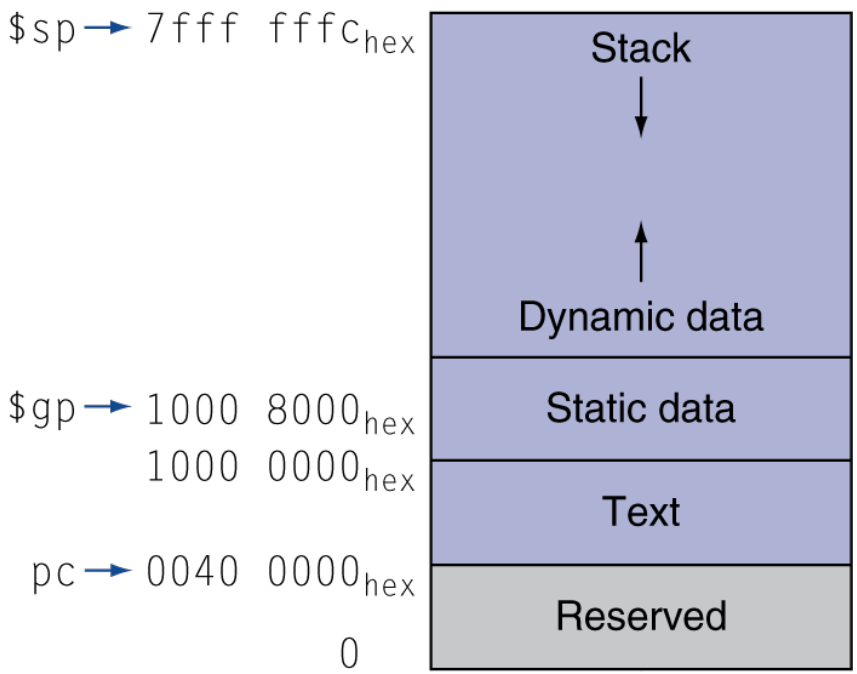
Static data: global variables

- ↳ static/global variables, constant arrays, etc.
- ↳ $gp initialized to address allowing ±offsets into this segment

Dynamic data: heap

- ↳ e.g., malloc in C, new in Java

Stack: "automatic" storage



Note: In this diagram the higher memory addresses are at top.

Handling the Stack in MIPS

```
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
        ...                  # call swap a bunch to do bubble sort
-------------------------------------------------------------
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
```

CS3350B Computer Organization
Chapter 3: CPU Control & Datapath
Part 2: Single Cycle Datapath

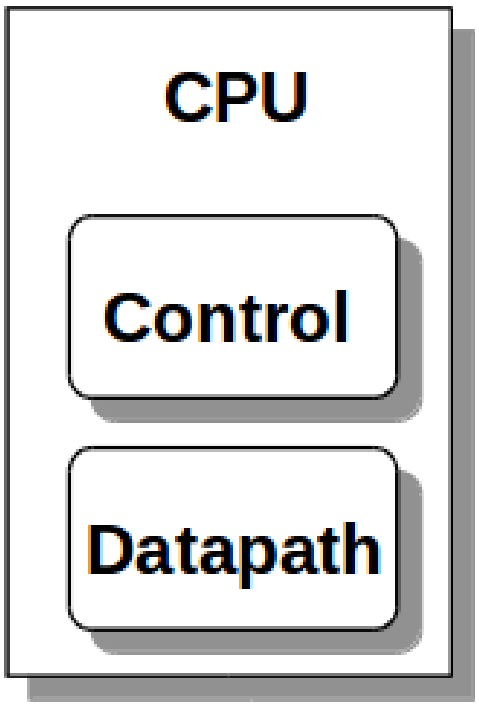
Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Tuesday February 26, 2019

Outline

Defining Parts of the Processor

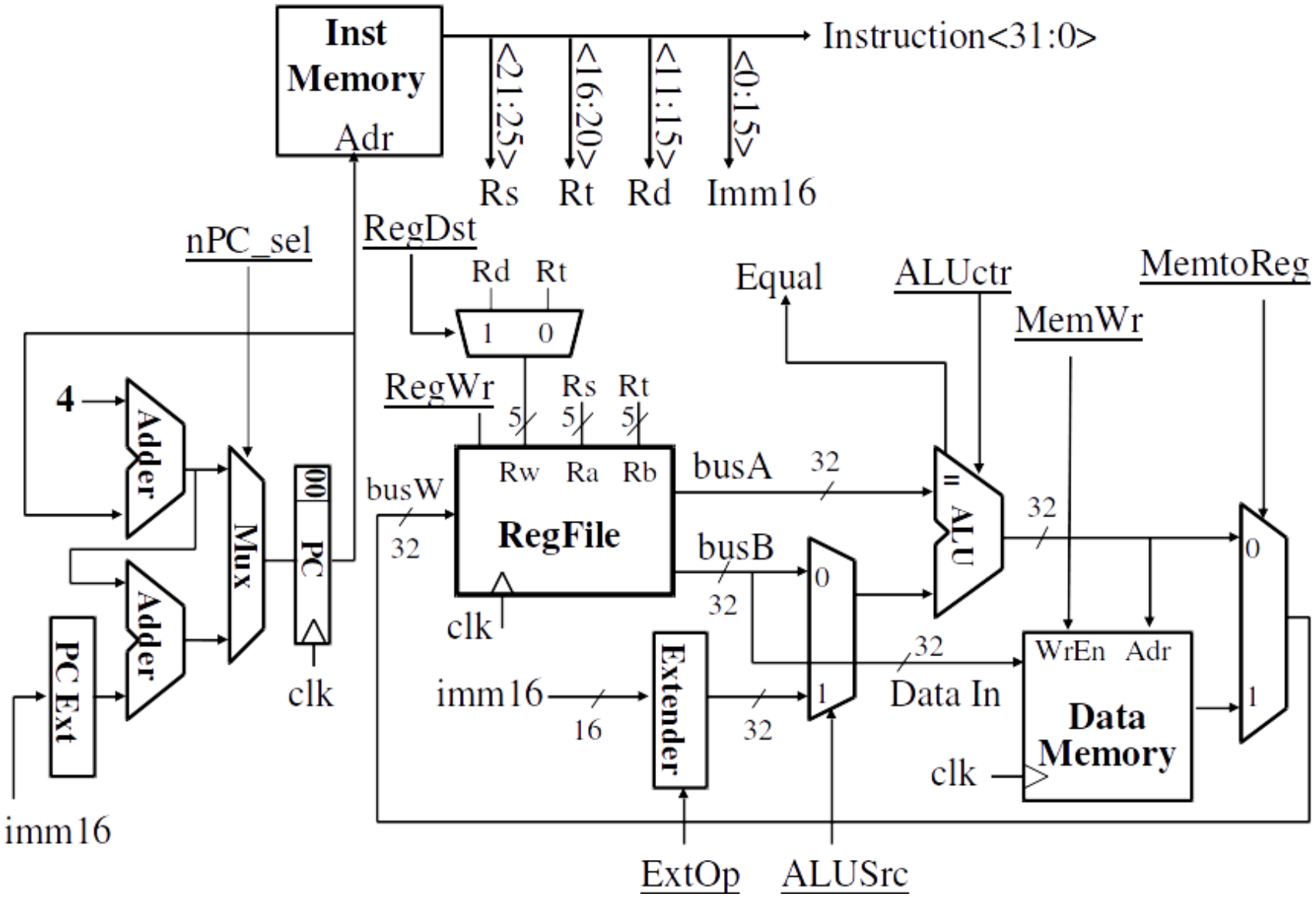


CPU/Processor: The encapsulation of the "working" part of the computer. Performs all the math, arithmetic, thinking, etc.

Datapath: The flow of data through the processor. Contains circuits and logic, arithmetic, etc. What does the actual work.

Control: Controls the flow of data through the datapath. Controls the circuits' operations (e.g. what operation the ALU will perform).

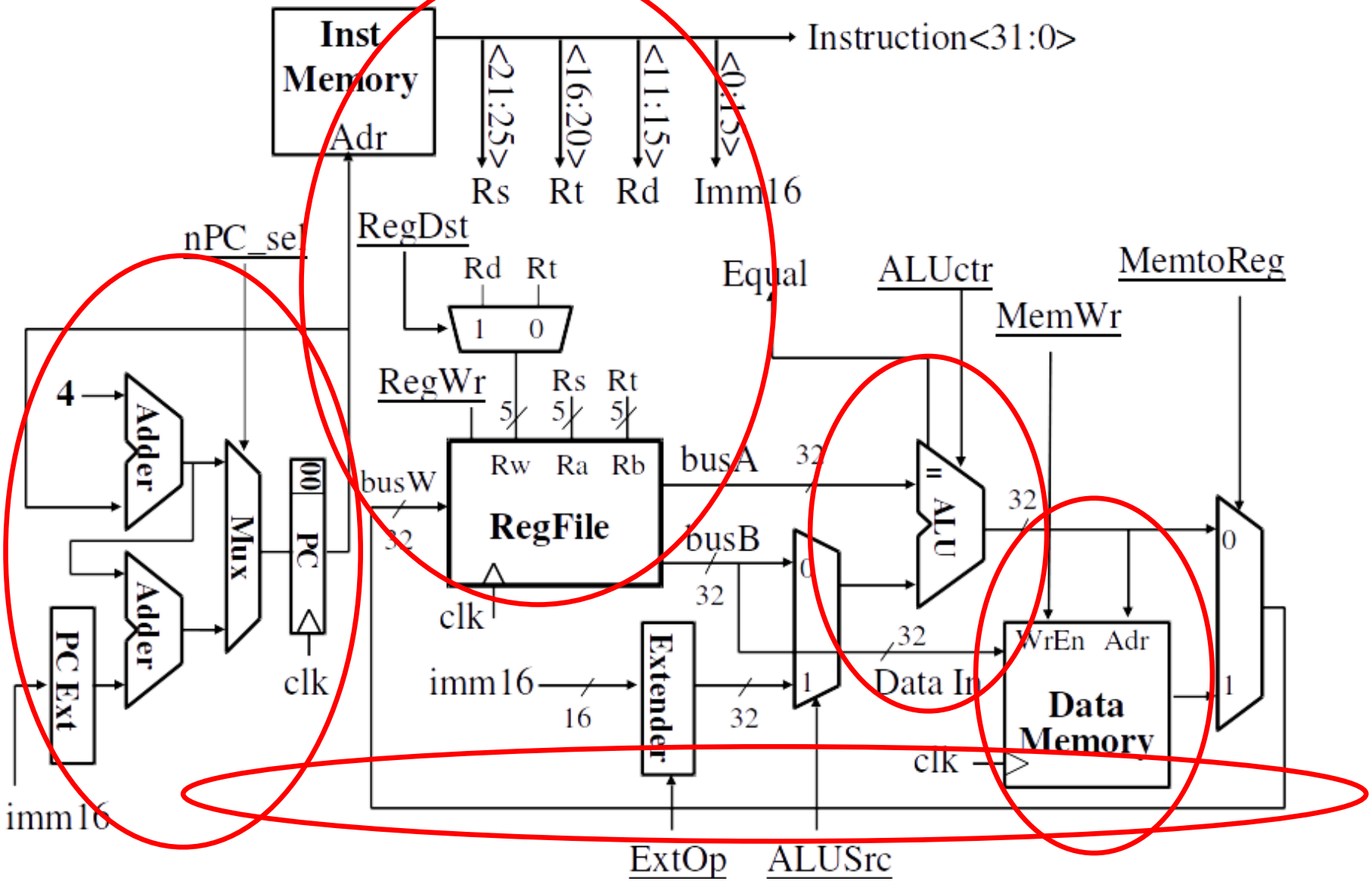
Preview: MIPS Datapath



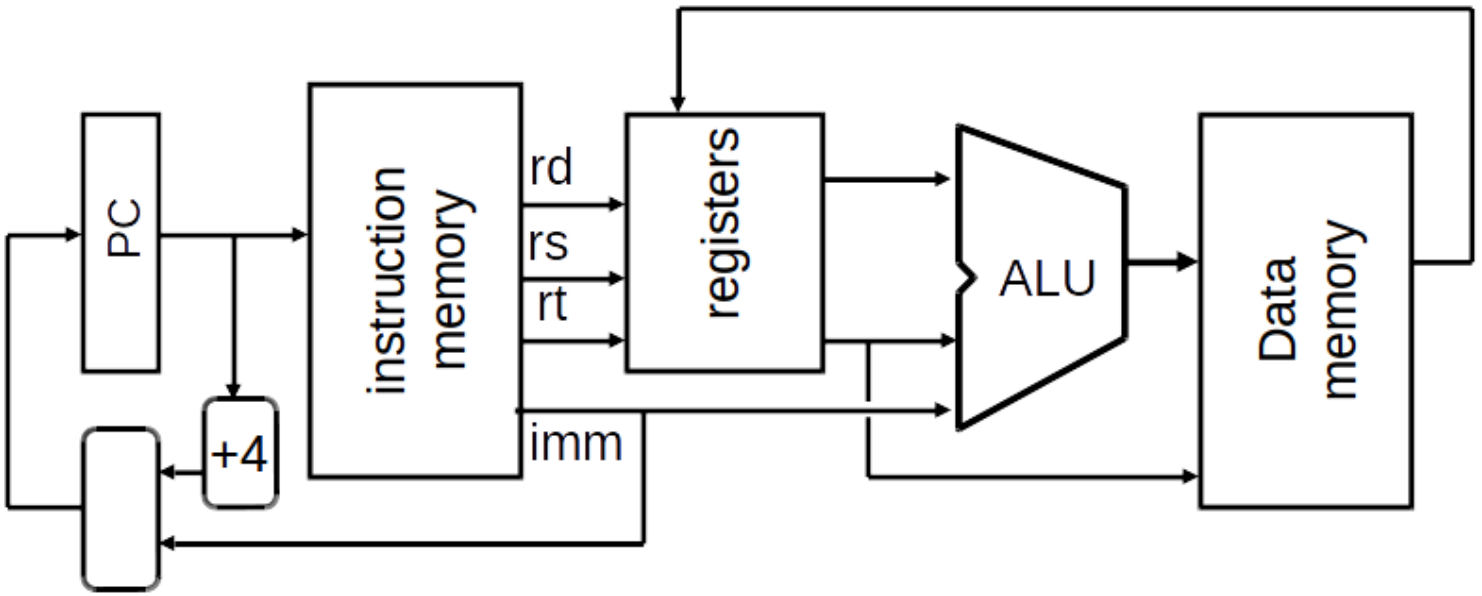
The 5-Stages of the Datapath

1. **IF**: Instruction Fetch
2. **ID**: Instruction Decode
3. **EX/ALU**: Execute/Arithmetic
4. **MEM**: Access Memory
5. **WB**: Write-back result

MIPS Datapath, Spot The Stages



A Simplified Datapath



5 Stages in the Path

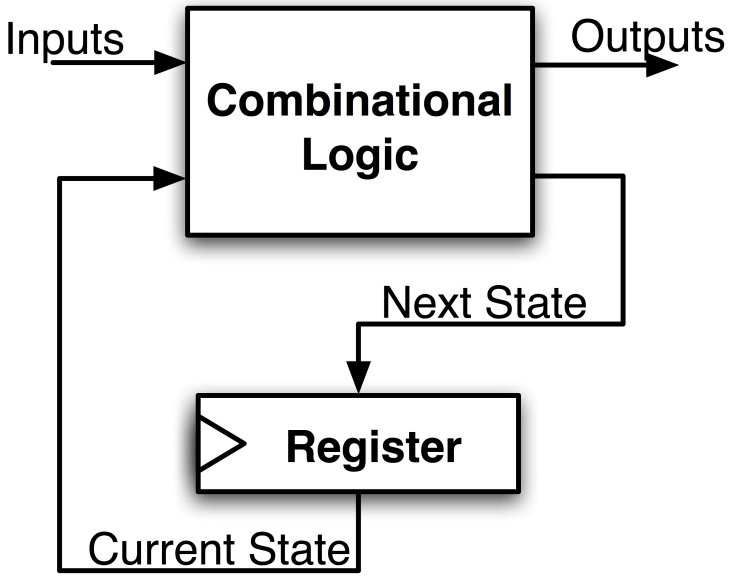
Why is there 5 stages?

- That's just what the designers of MIPS came up with.
  - ↳ Also, SPARC and Motorola.
  - ↳ Has been deemed the "Classic RISC Pipeline".

- Many other architectures use a different number of stages.
  - ↳ Intel has used 7, 10, 20, and 31 stages.
  - ↳ More stages $\Longrightarrow$ More complexity in circuits and control.

- Roughly speaking, each stage takes the same amount of time.
  - ↳ Prelude to Chapter 3: Part 4: The multi-cycle datapath

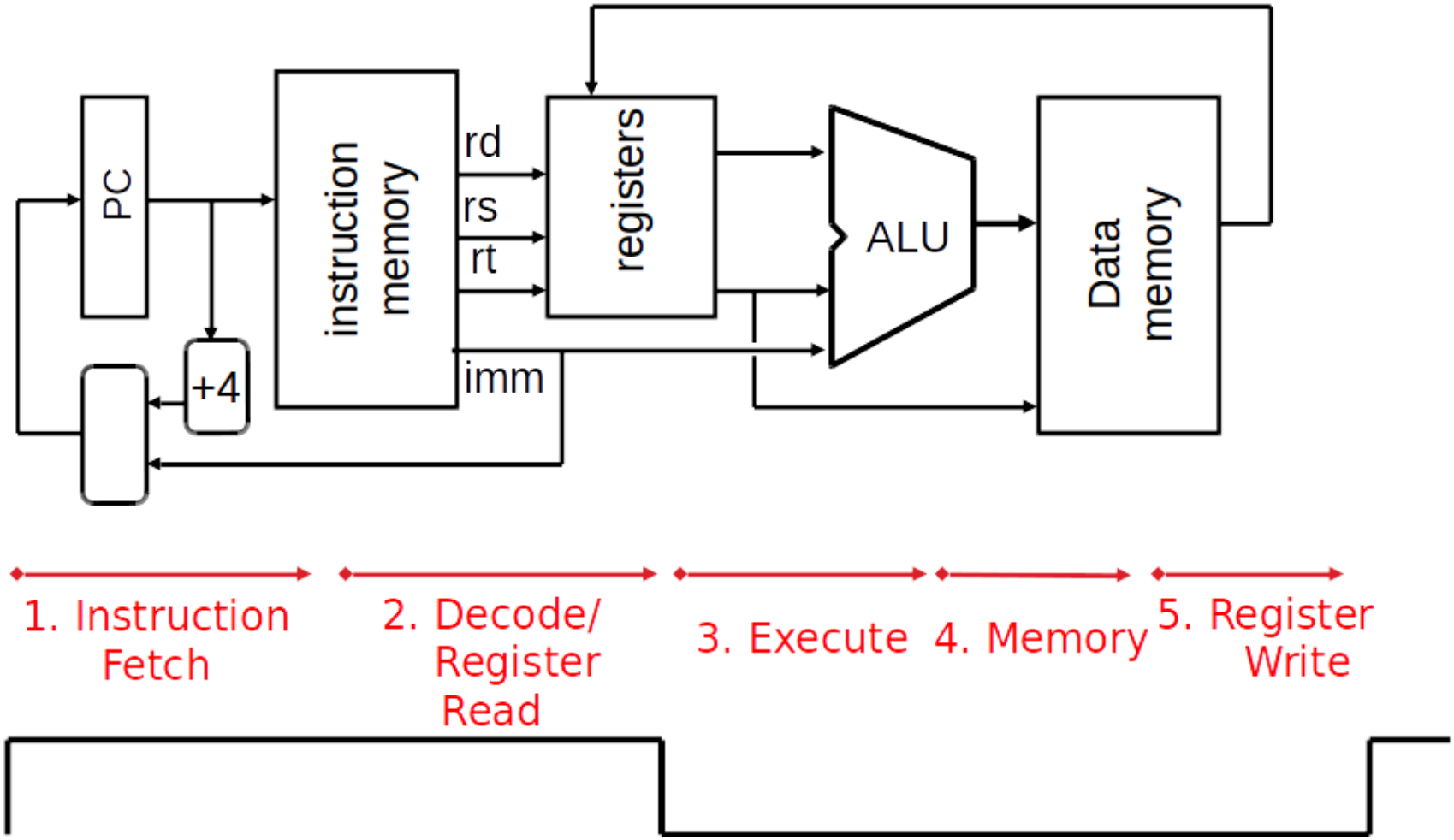
Single Cycle Datapath



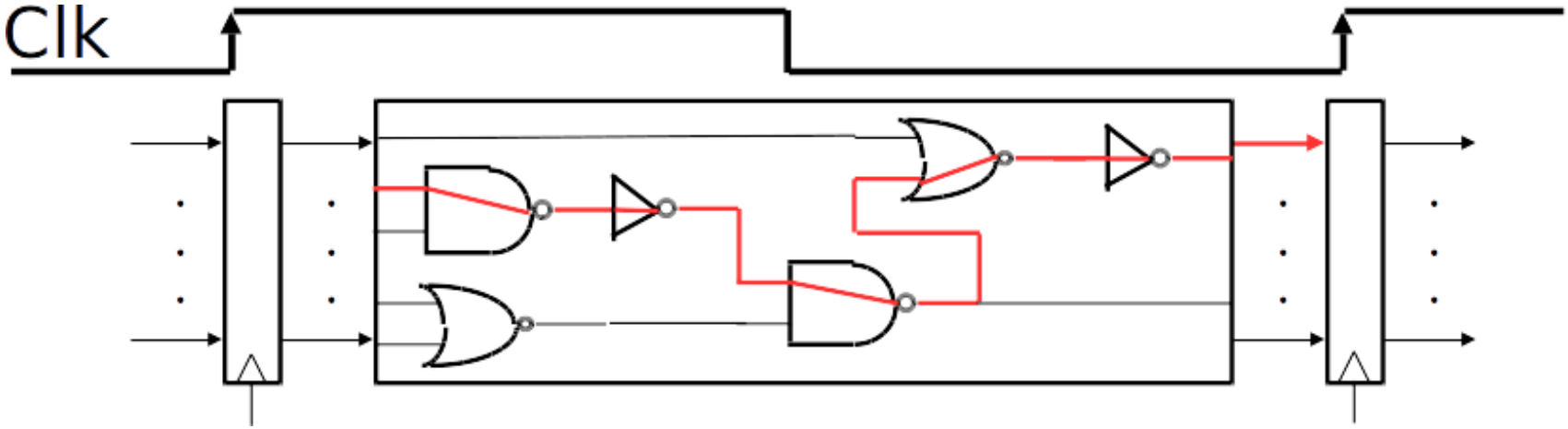
What makes a datapath **single cycle**?

- Flow of data through all stages of the datapath must occur within one clock cycle.
- The tic of the clock corresponds to the start of a new instruction starting to execute.
- One instruction is fetched, decoded, executed per clock cycle.
- Clock cycle must be long enough account for propagation delay of entire data path.

Clock Cycle for Single Cycle Datapath

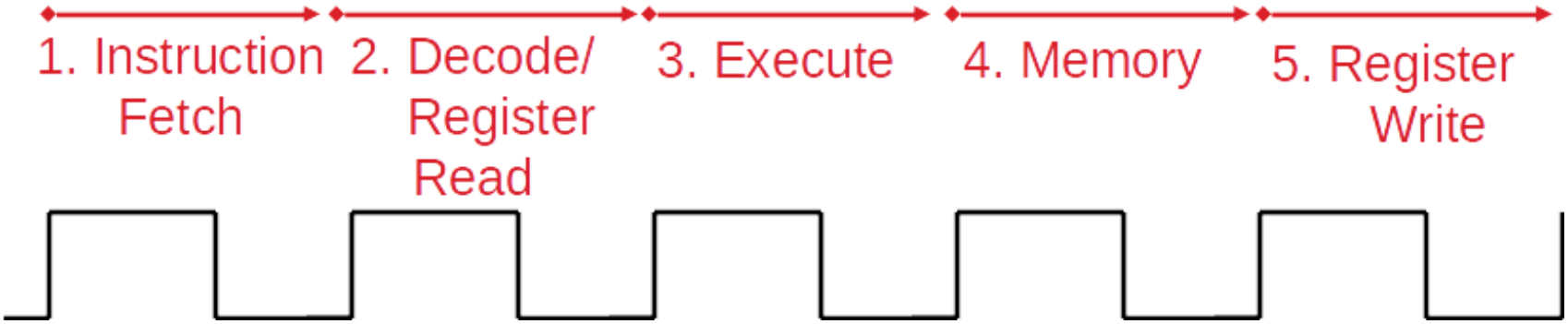


Critical Path Clocking



- The **critical path** determines length of clock cycle.
- Clock cycle must be long enough to accommodate the propagation delay of the longest path through the combination logic/datapath.
- *Recall:* all registers synchronized by the same rising edge of clock.

Multi-Cycle Datapath



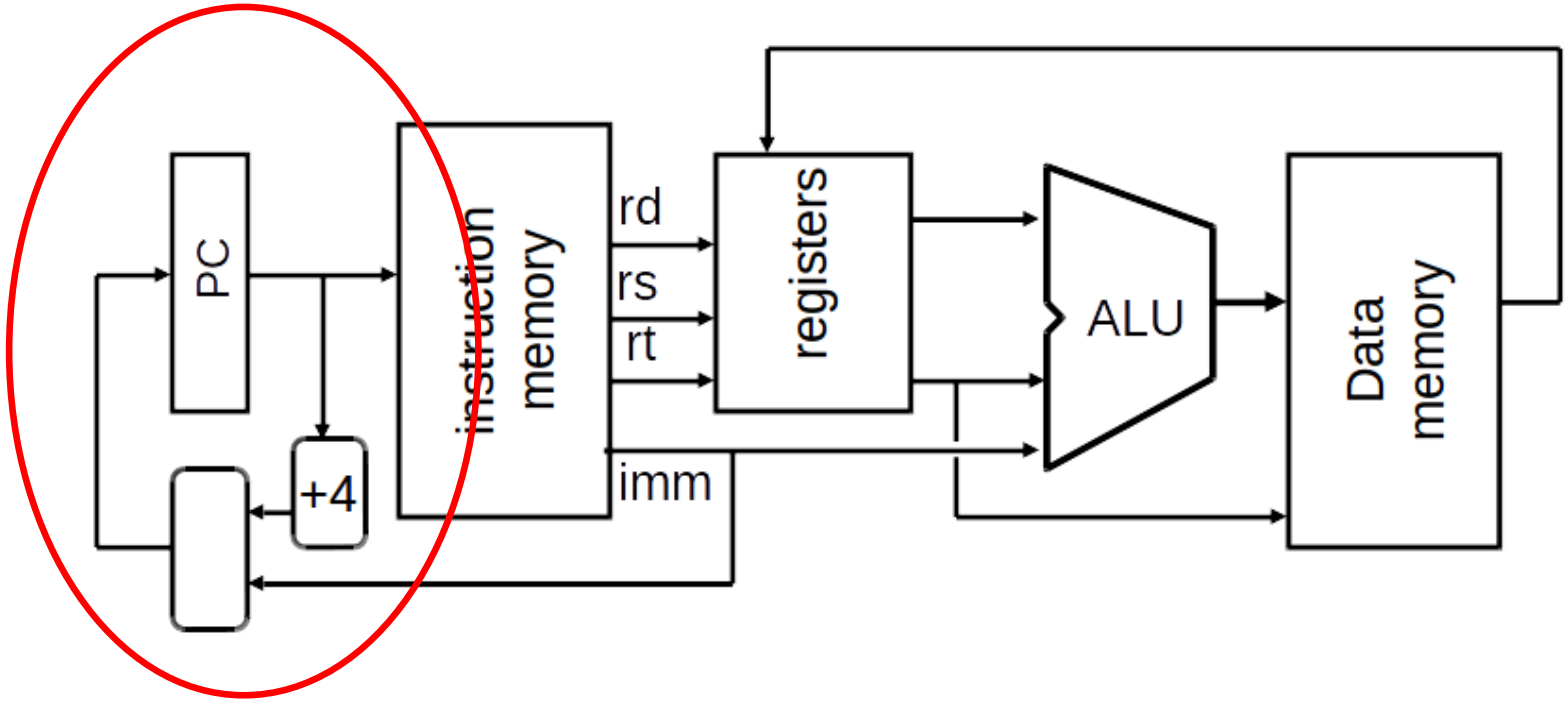
- One clock cycle per *stage* within datapath.
- Clock cycle must be long enough to accommodate slowest stage.
- Allows for optimizations:
  - ↳ Skipping unused stages.
  - ↳ *Pipelining*.
  - ↳ We ignore these optimizations until the next chapter.

Outline

The Five Stages

- The components of the datapath represent the **union** all circuitry needed by every instruction.
- Not every instruction will use every stage.
- Not every instruction will use every component within a stage.
- Nonetheless, all components are necessary to fulfill all instructions specified in the Instruction Set Architecture.

Instruction Fetch (1/2)

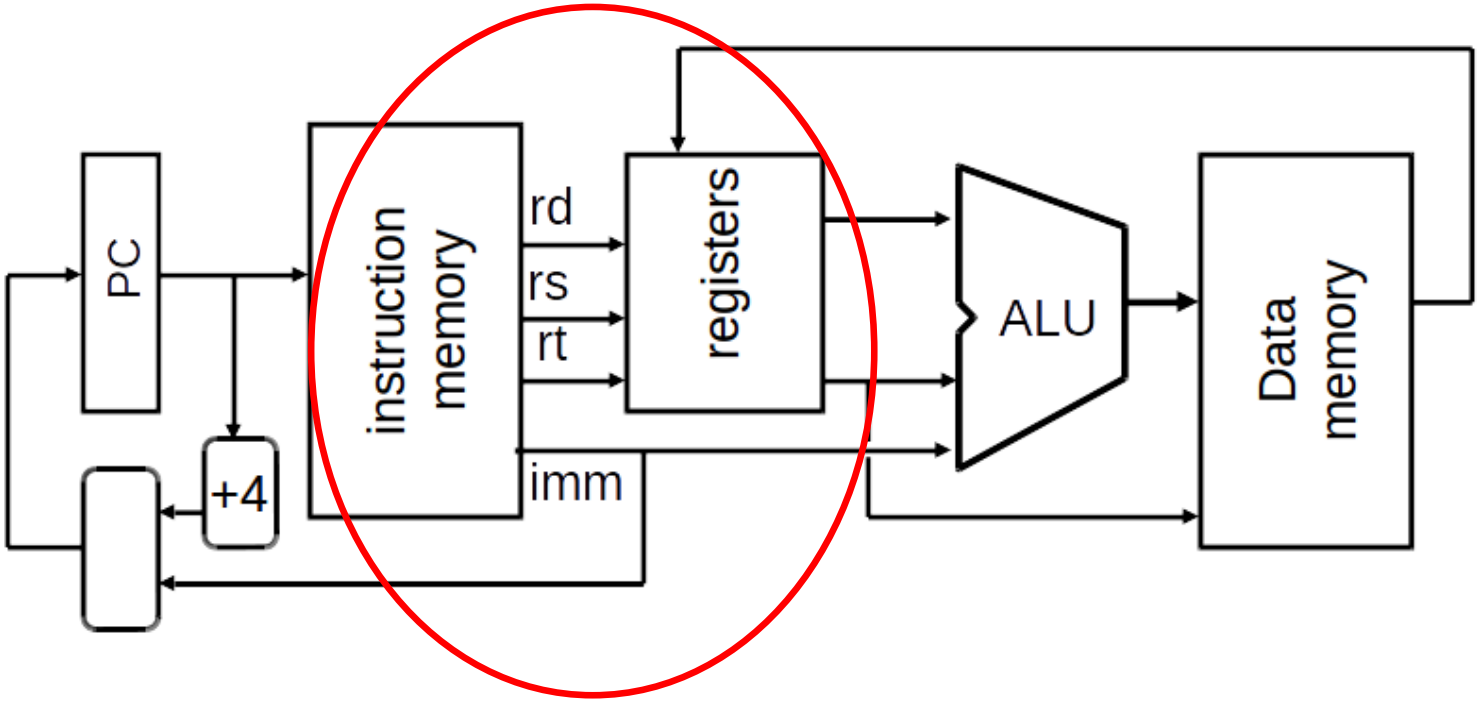


Instruction Fetch (2/2)

Instruction Fetch

- The instruction must be fetched from the instruction memory (banked L1 cache).
- Instructions are themselves encoded as a binary number.
- Instructions are stored in a memory word.
  - ↳ 32 bits in the case of MIPS.
- *Increment PC*: update the program counter for the next fetch.

Instruction Decode (1/2)

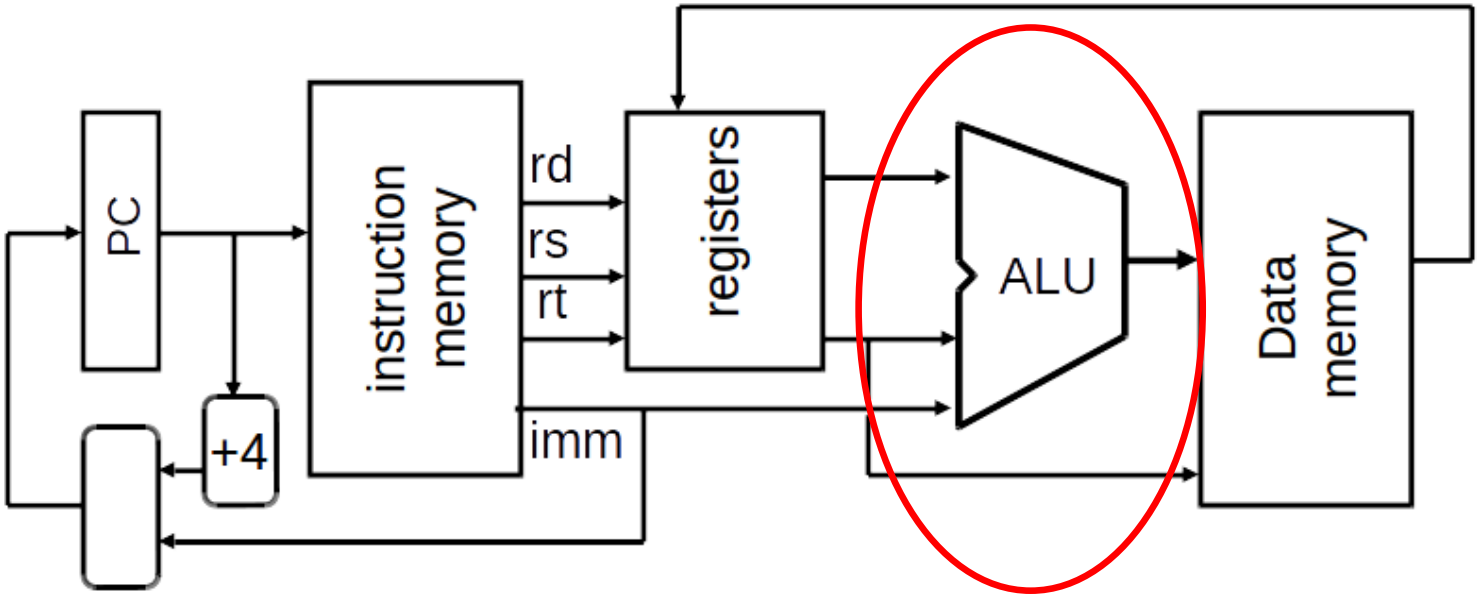


Instruction Decode (2/2)

Instruction Decode

- Determine the type of instruction to execute.
  - ↳ Read the *opcode*; it's always the first 6 bits in MIPS, regardless of the eventual type of instruction.
- Knowing the type of instruction, break up the instruction into the proper chunks; determine the instruction operands.
- Once operands are known, read the actual data (from registers) or extend the data to 32 bits (immediates).

Execute (a.k.a. ALU) (1/2)

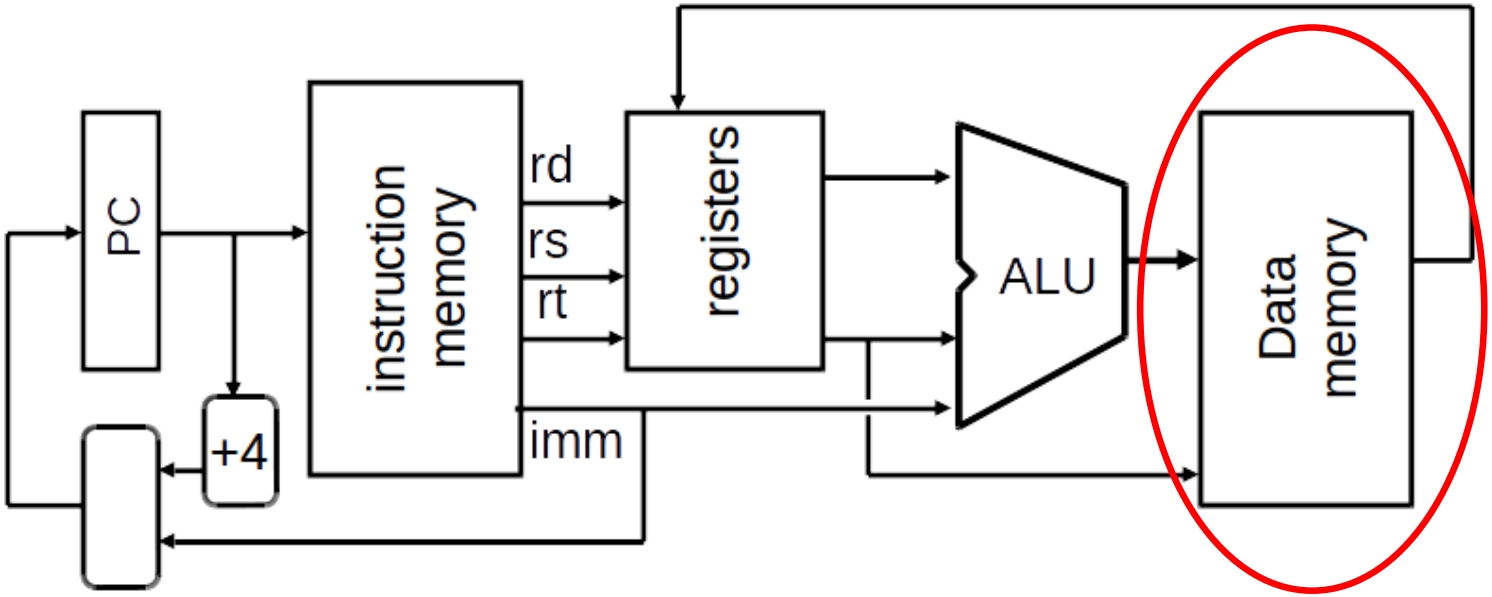


Execute (a.k.a. ALU) (2/2)

Execute

- Do the actual work of the instruction.
  - ↳ Add, subtract, multiply, shifts, logical operations, comparisons.
- For data transfer instructions, calculate the actual address to access.
  - ↳ Recall data transfer instructions have an offset and a base address.
  - ↳ `lw $t1, 12($t0)`
  - ↳ Calculates memory address `$t0 + 12`.

Memory Access (1/2)

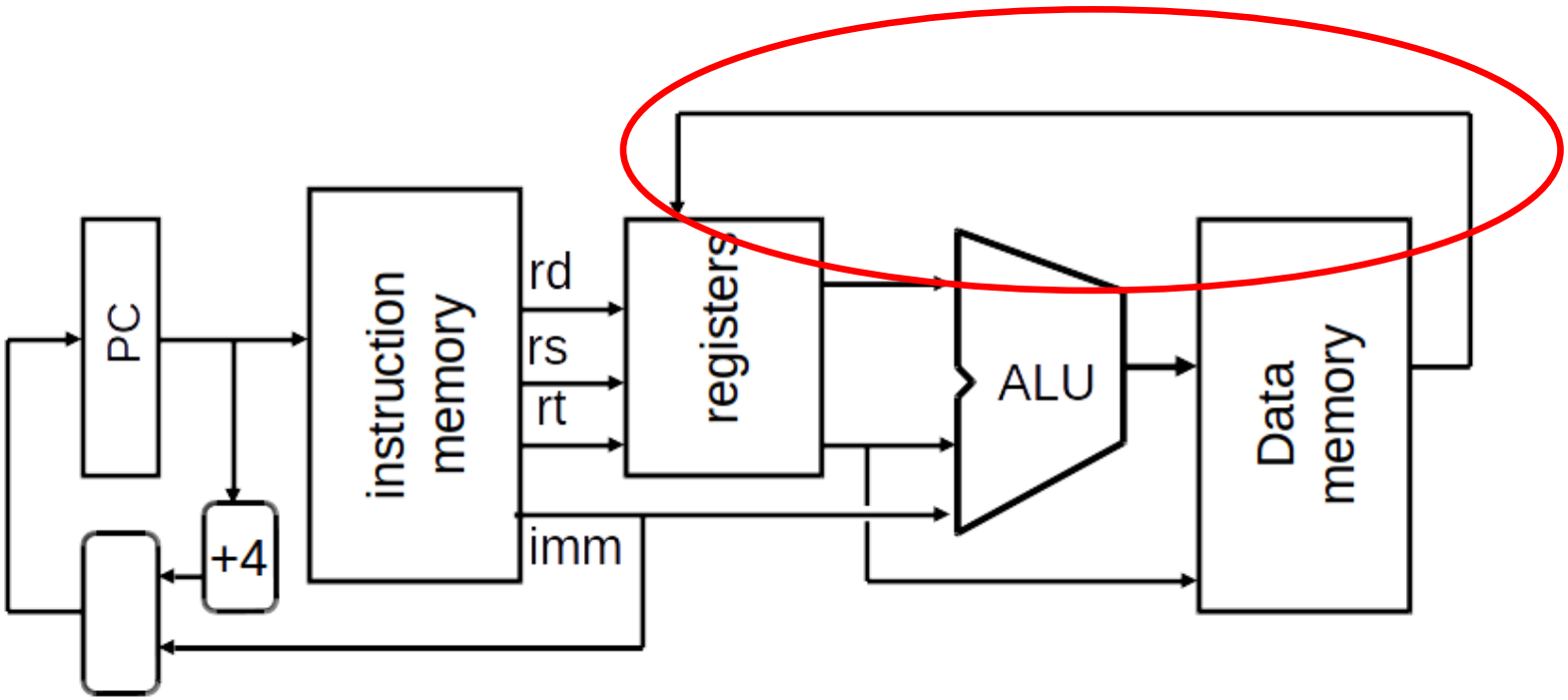


Memory Access (2/2)

Memory Access

- Access the memory using the address calculated in EX stage.
- Can be a read or a write.
- If the particular instruction is not a memory-accessing instruction, just *do nothing*.
- Since memory is relatively slow, just reading (or writing) data from it takes as much time as doing a full arithmetic operation.
  - ↳ **But** still quite fast due to caching and the memory hierarchy.
  - ↳ EX stage and MEM stage roughly same time. (Well really all stages are all roughly the same time.)

Write Back (1/2)



Write Back (2/2)

Write Back

- Write back the calculated value to the register.
- Could be the result of some arithmetic operation.
- Could be the result of some memory load.
- If nothing is being written back (e.g. on a memory store) just *do nothing*.
- Not to be confused with write back cache policy.

Outline

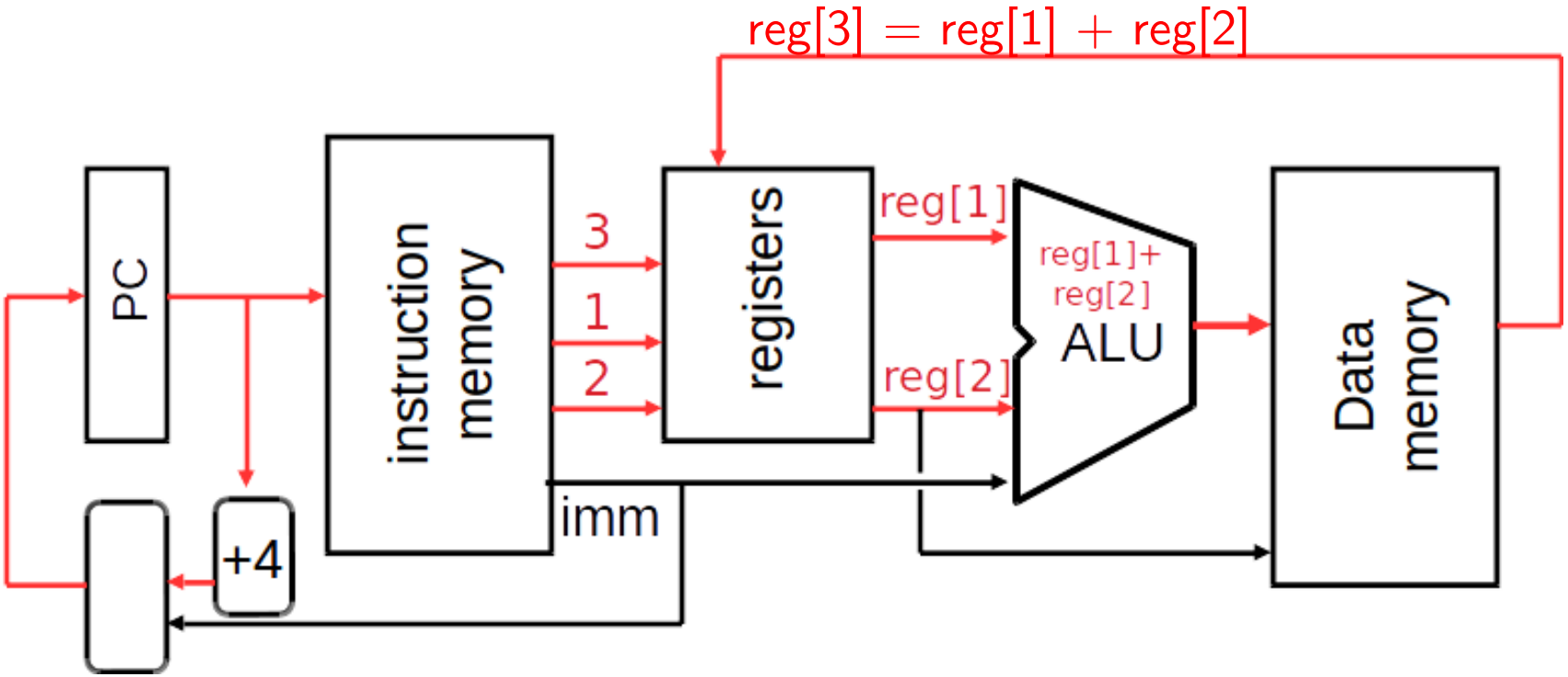
Example 1: add (1/2)

$$\texttt{add } \$3, \$1, \$2 \Rightarrow \$3 = \$1 + \$2$$

op	rs	rt	rd	shamt	funct
0	1	2	3	0	32
000000	00001	00010	00011	00000	100000

- IF: Fetch instruction and increment PC.
- ID: Read opcode, determine R-type instruction, read values of $rs, $rt.
- EX: Perform addition operation on values stored in $1 and $2.
- MEM: Do nothing.
- WB: Write the sum back to $3.

Example 1: add (2/2)




```
add $3, $1, $2
```

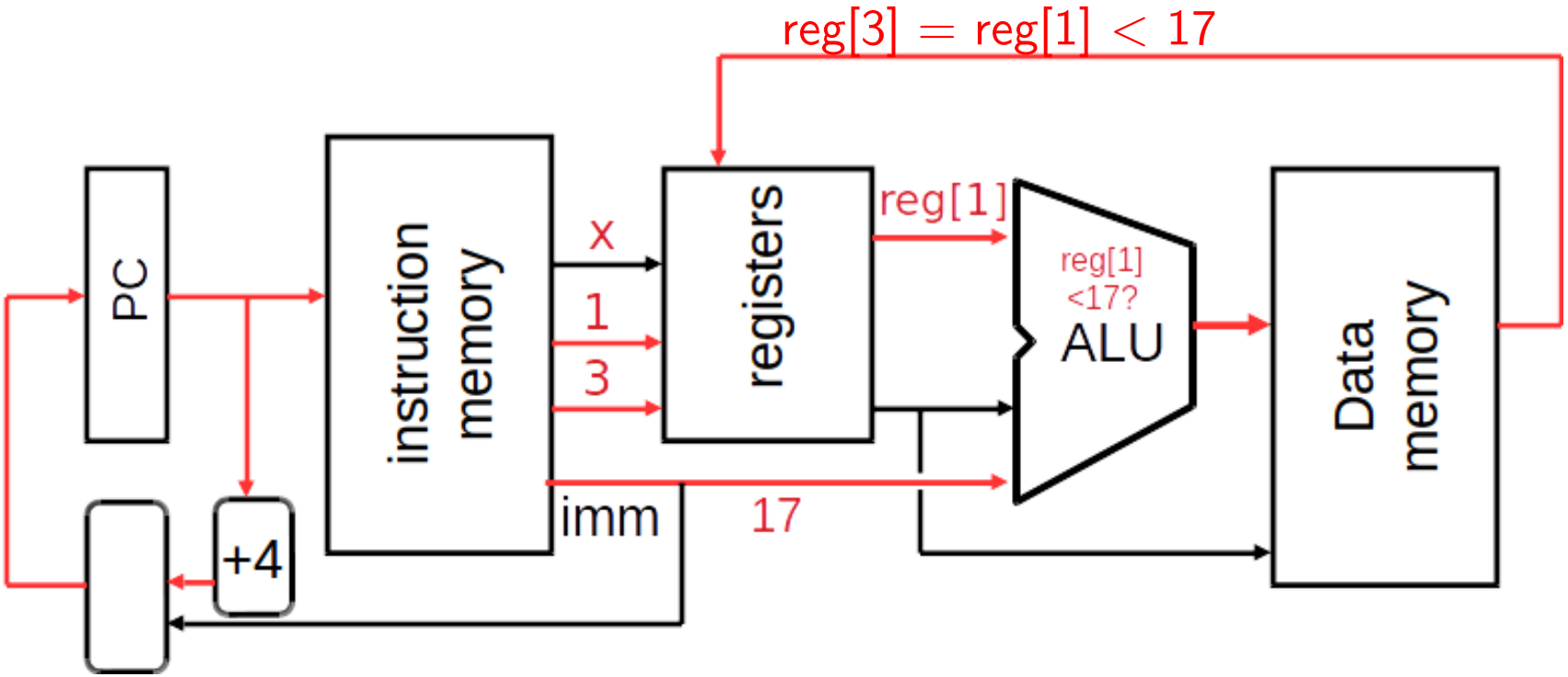
Example 2: `slti` (1/2)

$$\texttt{slti \$3, \$1, 17} \Rightarrow \$3 = (\$1 < 17)$$

op	rs	rt	immediate
001010	00001	00011	0000000000010001

- IF: Fetch instruction and increment PC.
- ID: Read opcode, determine I-type instruction, read values of $rs, immediate.
- EX: Perform comparison operation on value of $1 and immediate.
- MEM: Do nothing.
- WB: Write the comparison result back to $3.

Example 2: `slti` (2/2)




```
slti $3, $1, 17
```

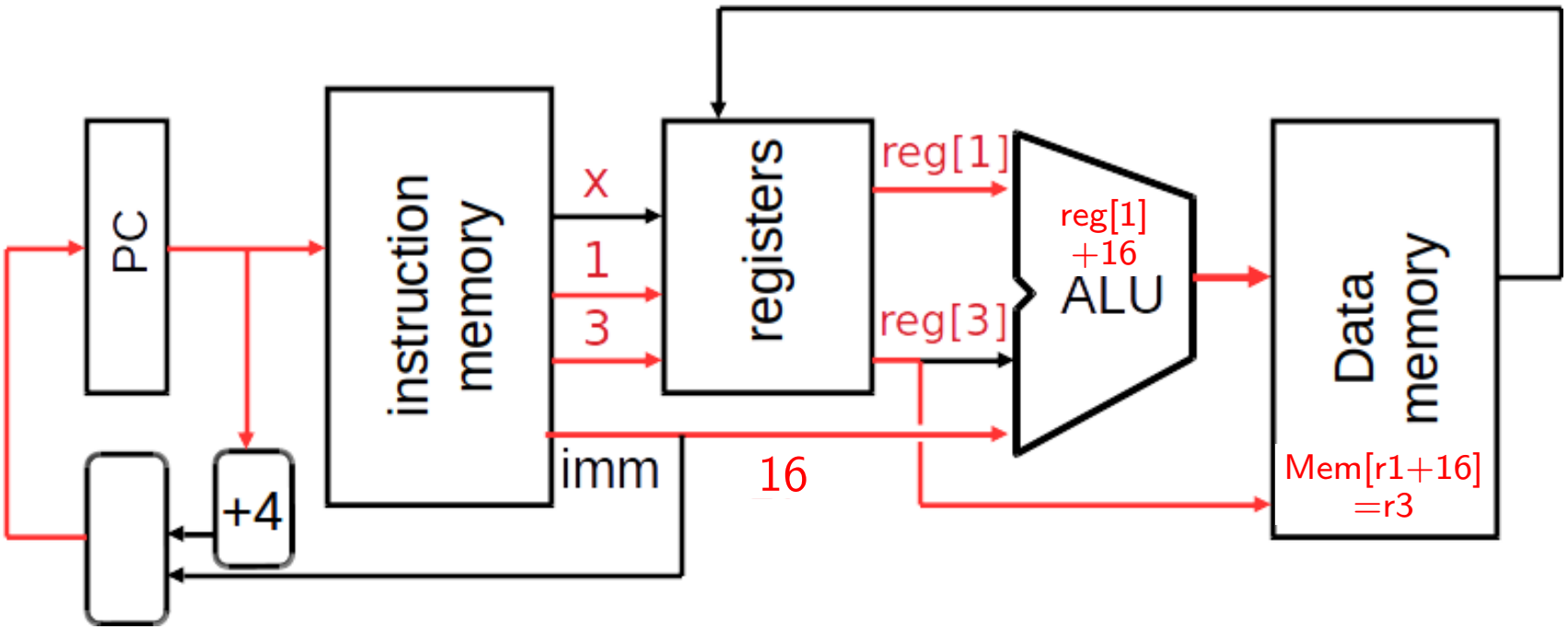
Example 3: `sw` (1/2)

$$\texttt{sw \$3, 16(\$1)} \Rightarrow \text{Mem}[\$1 + 16] = \$3$$

op	rs	rt	immediate
101011	00001	00011	0000000000010001

- IF: Fetch instruction and increment PC.
- ID: Read opcode, determine I-type instruction, read values of \$rs, \$rt, imm.
- EX: Calculate memory address from reg[1] and 16 (offset).
- MEM: Write value of \$3 into Mem[reg[1] + 16].
- WB: Do nothing.

Example 3: `sw` (2/2)




```
sw $3, 16($1)
```

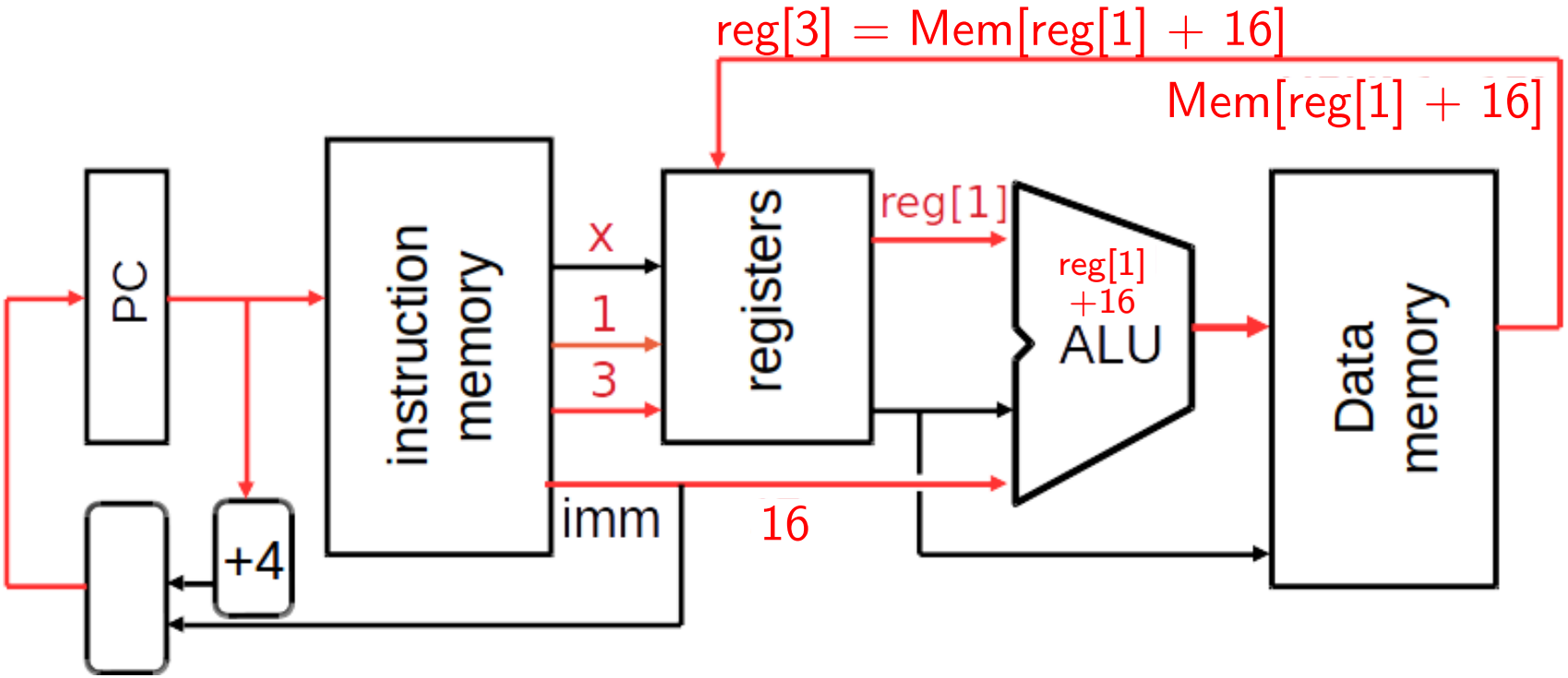
Example 4: `lw` (1/2)

$$\texttt{lw \$3, 16(\$1)} \Rightarrow \$3 = \text{Mem}[\$1 + 16]$$

op	rs	rt	immediate
101011	00001	00011	0000000000010001

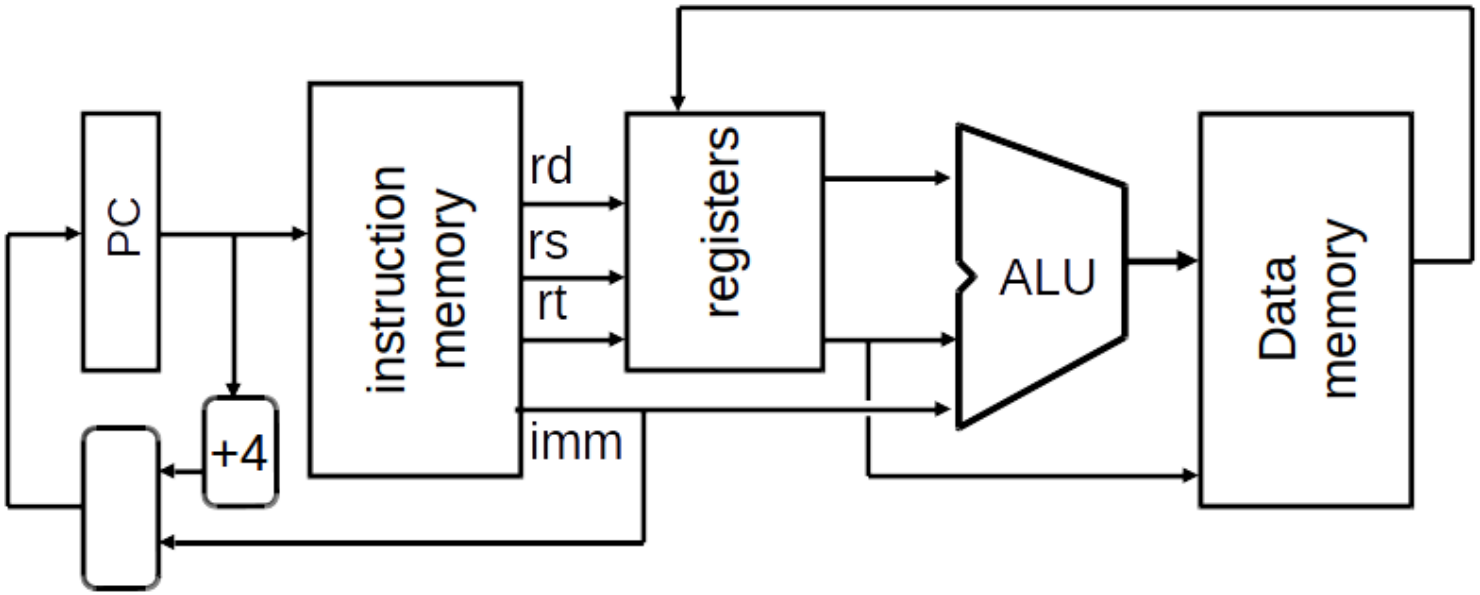
- IF: Fetch instruction and increment PC.
- ID: Read opcode, determine I-type instruction, read values of \$rs, imm.
- EX: Calculate memory address from reg[1] and 16 (offset).
- MEM: Read value of Mem[reg[1] + 16].
- WB: Write value of Mem[reg[1] + 16] to \$3.

Example 4: `lw` (2/2)




```
lw $3, 16($1)
```

Exercise: beq



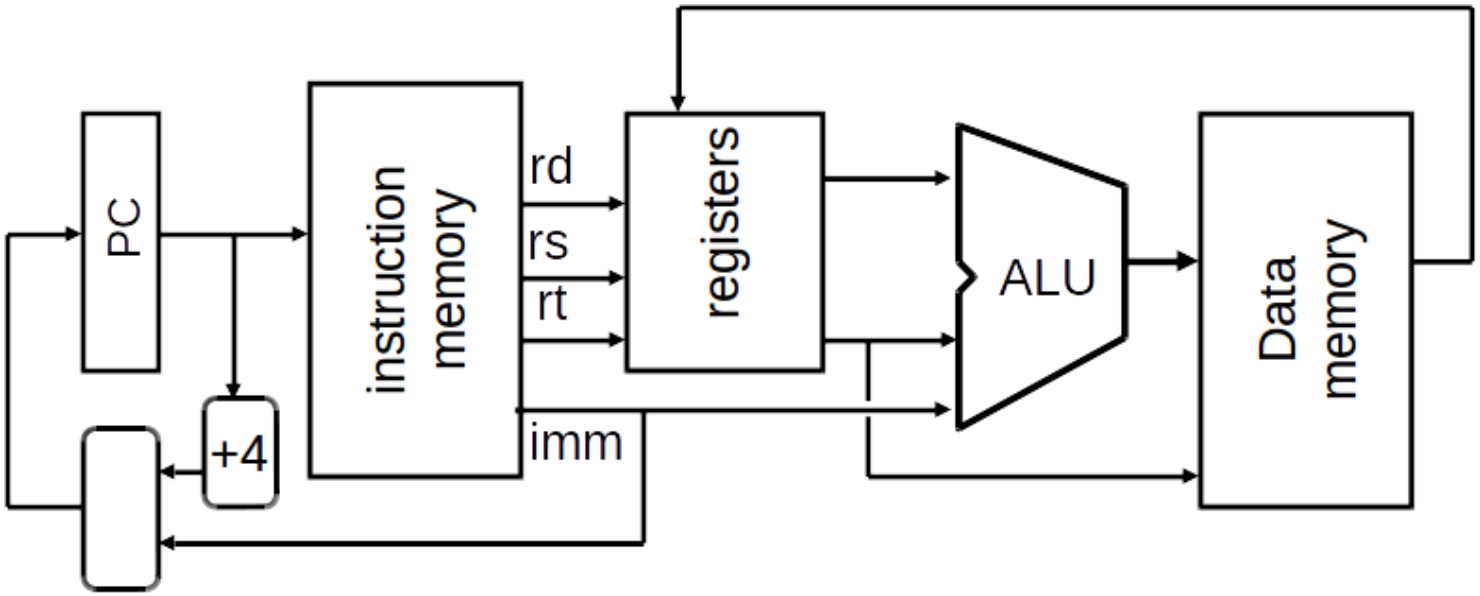

```
beq $8, $9, 128
```

Outline

Satisfying the ISA

- *Recall:* The specification of the ISA and the datapath are highly coupled.
  - ↳ We need enough circuity to accommodate every possible instruction in the ISA.
- Instructions belong to a few general categories. We need circuitry for to satisfy each and every one.
  - ↳ All instructions use PC and instruction memory.
  - ↳ Arithmetic: ALU, Registers.
  - ↳ Data transfer: Register, Memory.
  - ↳ Conditional jumping: PC, Registers, Comparator (ALU).
  - ↳ Unconditional jumping: PC, Registers.
- `lw` is one instruction which makes use of *every* stage.

Missing Datapath Details

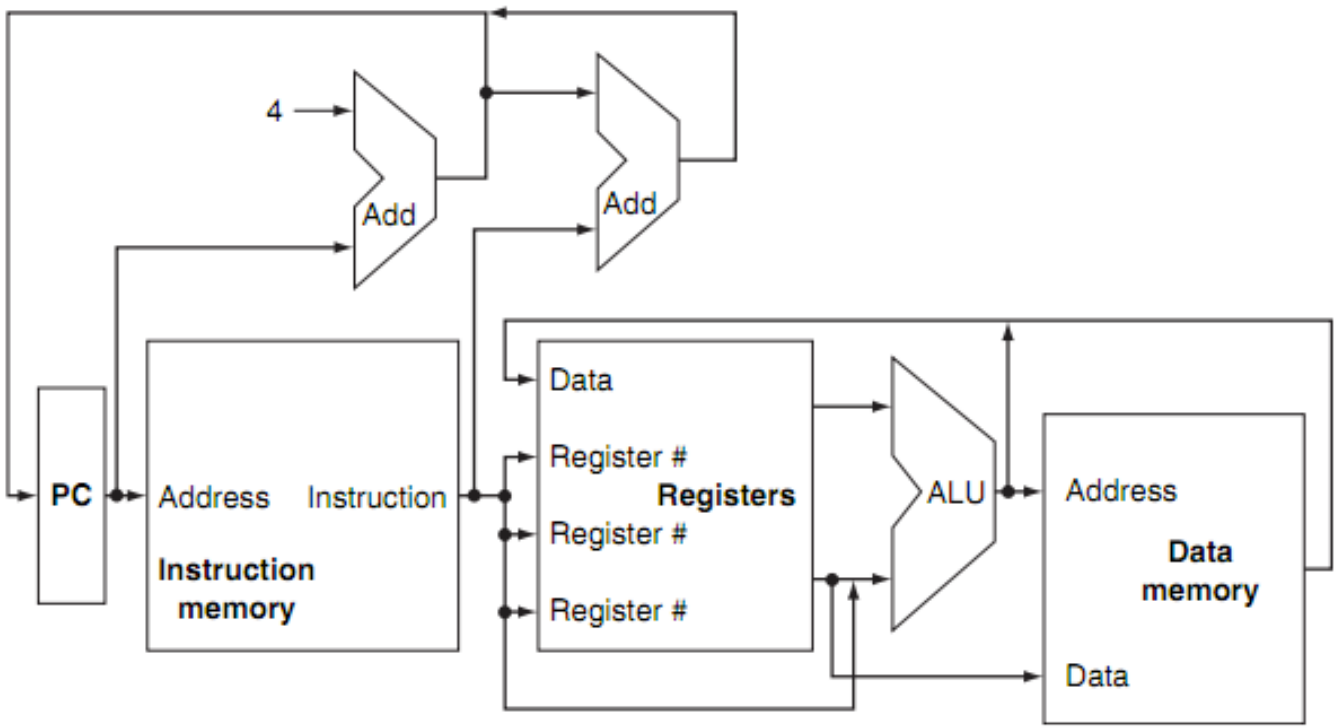


Many subtle details are missing from this simplified datapath.

- Multiplexers needed to control flow to/from registers, ALU, memory.
- Control which operation ALU performs.
- Control whether reading or writing write to memory, registers.

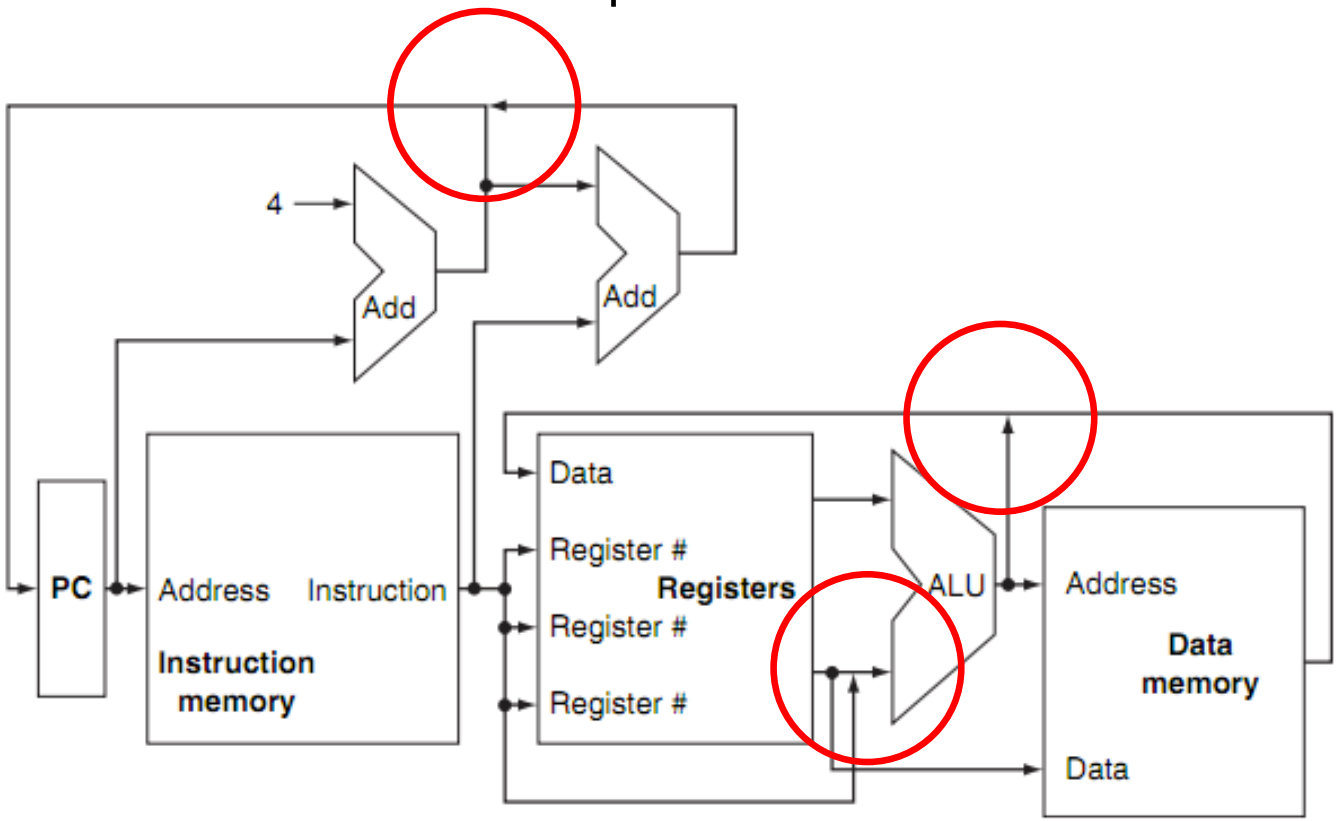
Multiplexers in the Datapath

Where do we need multiplexers to control data flow?

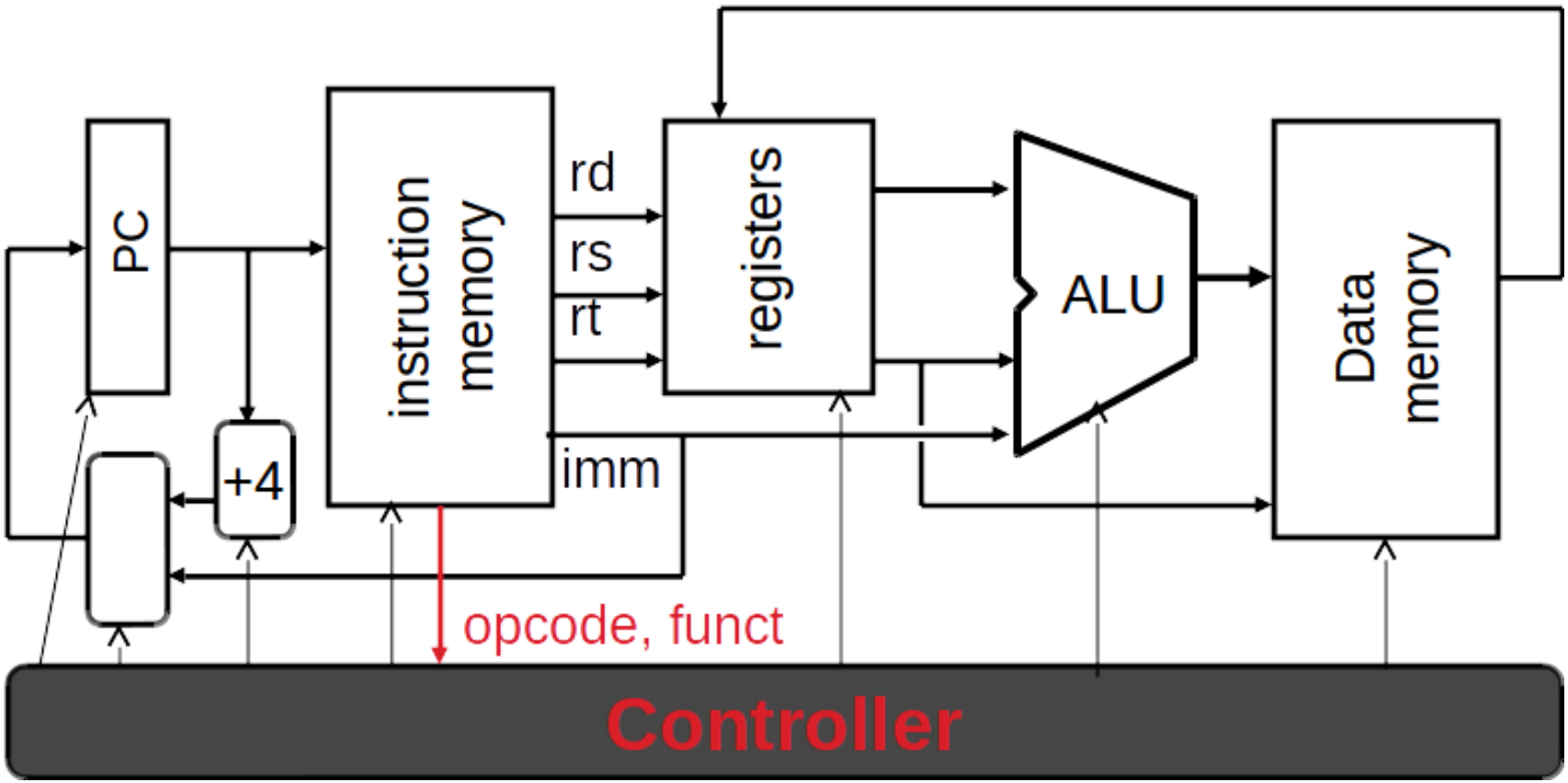


Multiplexers in the Datapath

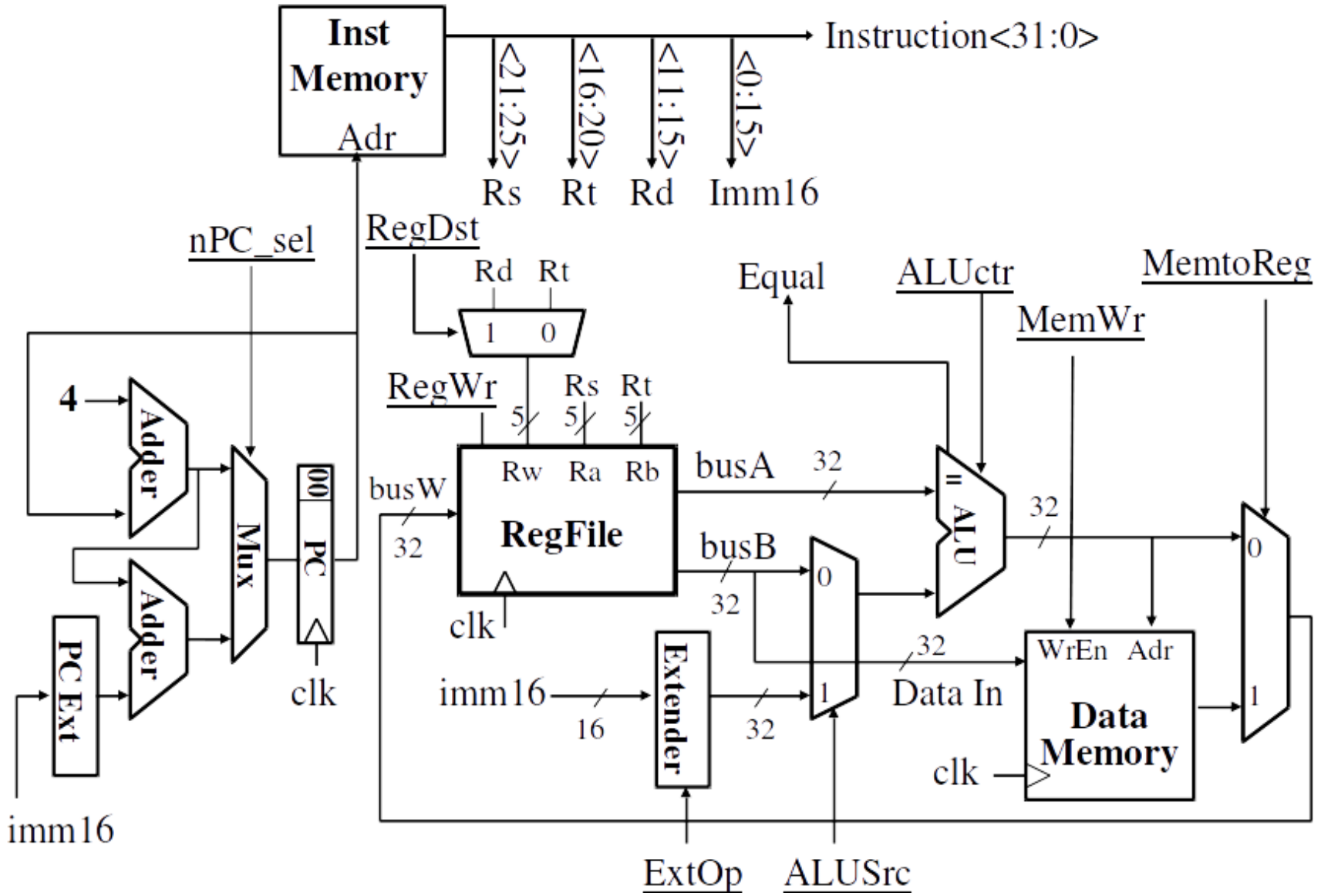
Where do we need multiplexers to control data flow?



Controlling the Multiplexers, ALU, Circuitry



MIPS Datapath with Control Signals



Datapath Summary

- ISA and circuitry highly coupled.
- 5 Stages: IF, ID, EX, MEM, WB.
- Some stages go unused for some instructions.
- Single cycle: clock cycle determined by propagation delay of entire datapath.
- Multi-cycle: clock cycle determined by propagation delay of *slowest* stage.
- Additional control (multiplexers, ALU, read/write) needed for the datapath.

CS3350B Computer Organization
Chapter 3: CPU Control & Datapath
Part 3: CPU Control

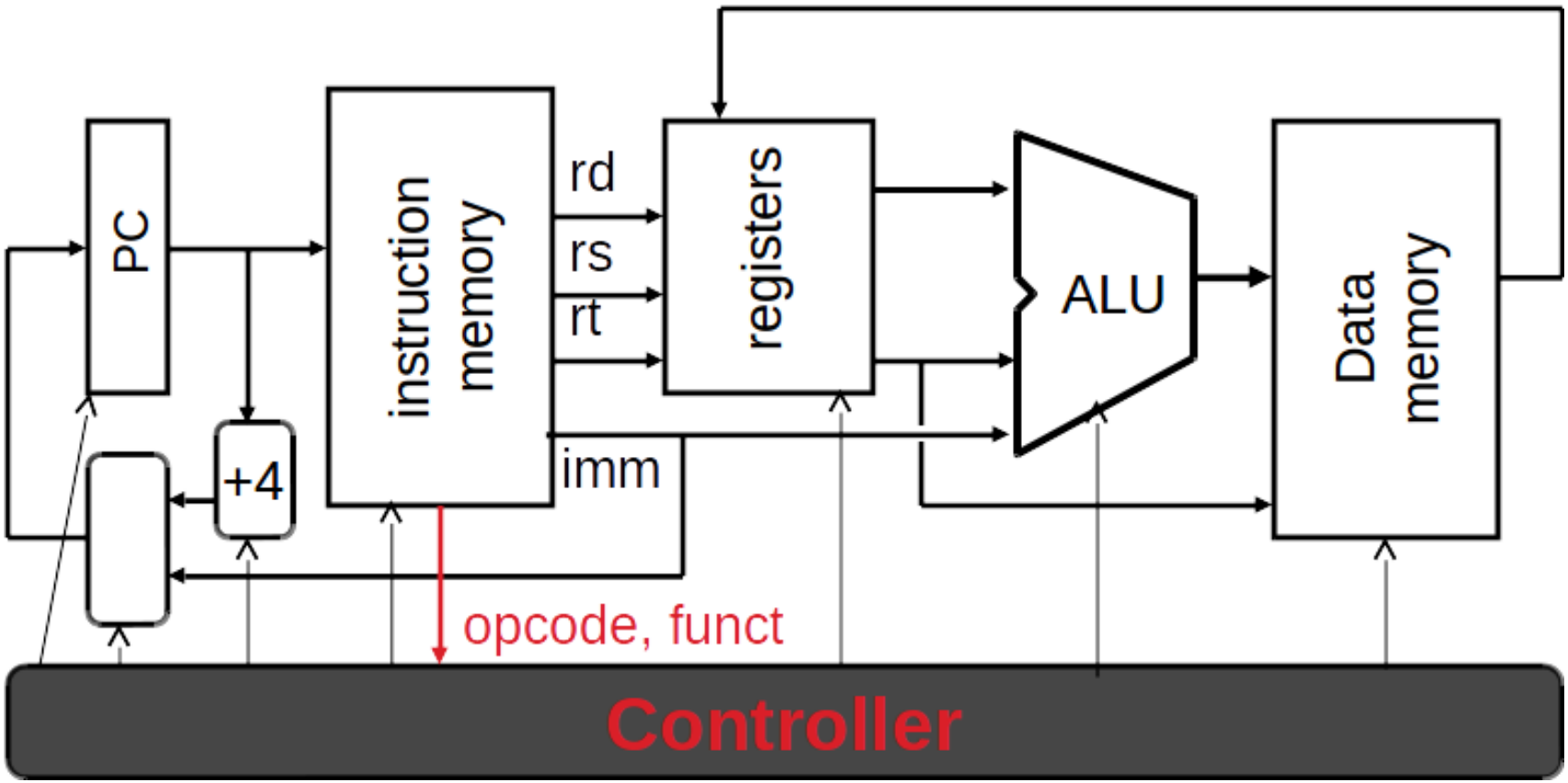
Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Thursday February 28, 2019

Outline

Controlling the Datapath

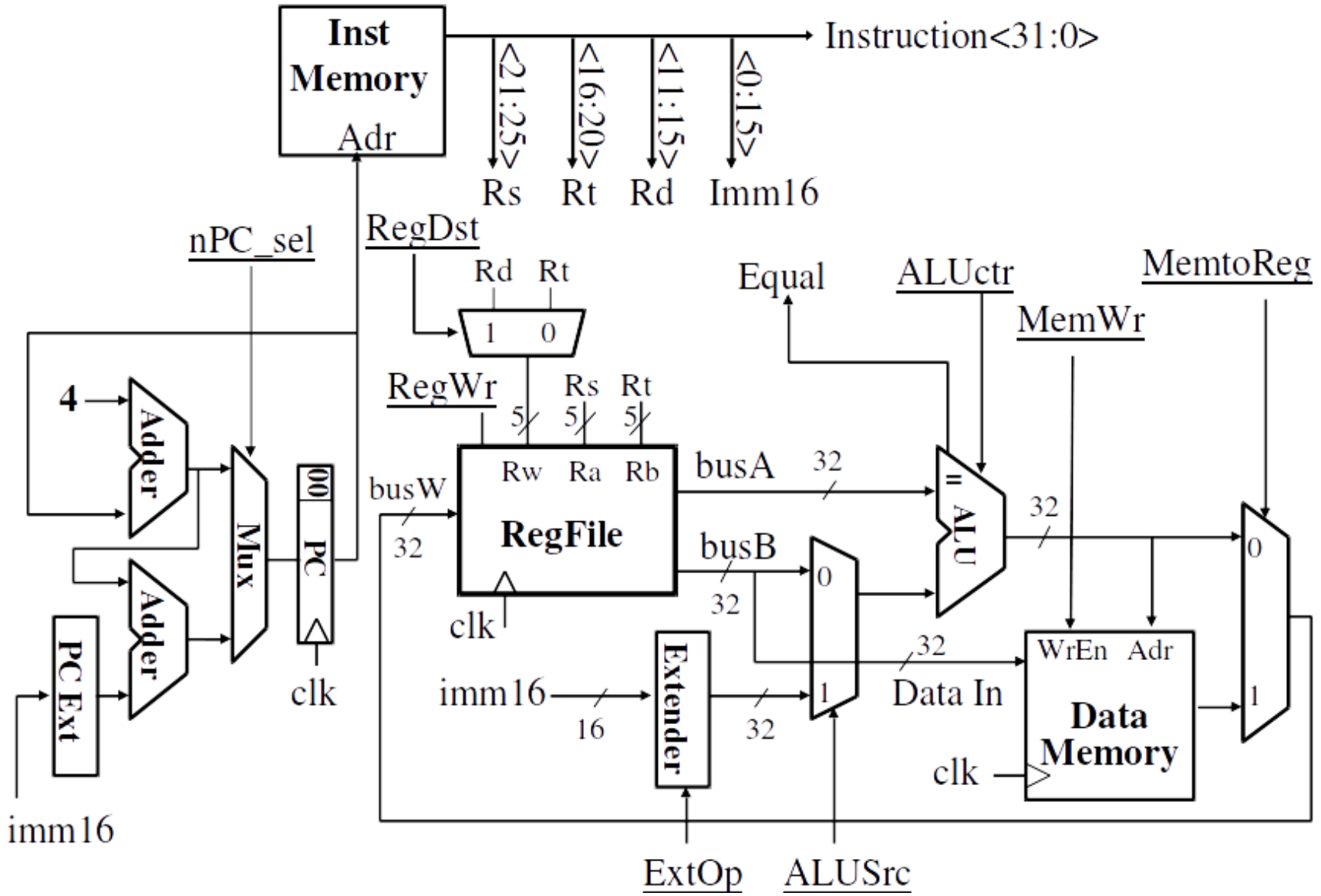


Control Signals

- Just as we saw with circuits like MUX, DEMUX, ALU, some circuits need **control signals** to help data flow or control the operation of the circuit.

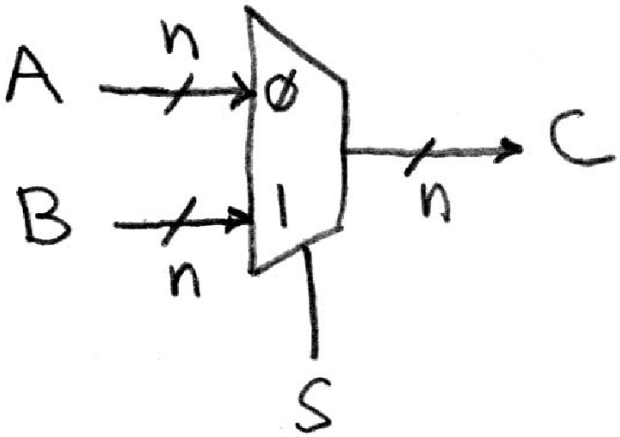
- For an entire CPU datapath, this is called the **CPU controller**.
  - ↳ The controller contains the logic which interprets instructions and sends out control signals.
  - ↳ Many independent control signals are sent from the controller to each stage.
  - ↳ Sometimes multiple signals are sent to one stage, each controlling a different component within a stage.

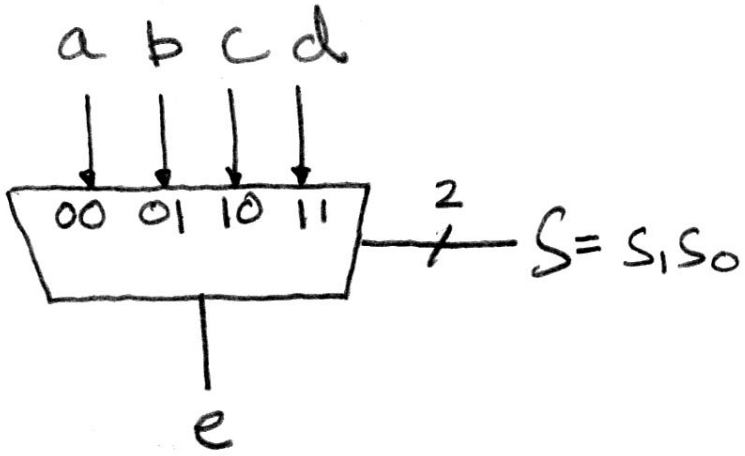
MIPS Datapath with Control Signals



Outline

Review: MUX

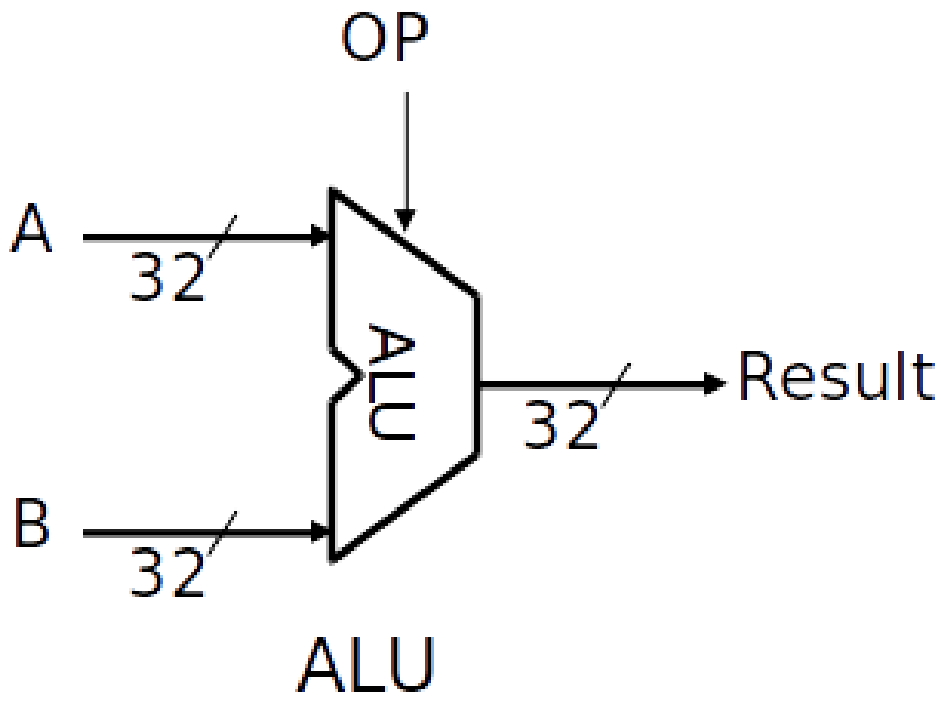




The control signal **S** determines which input is used to set the output.

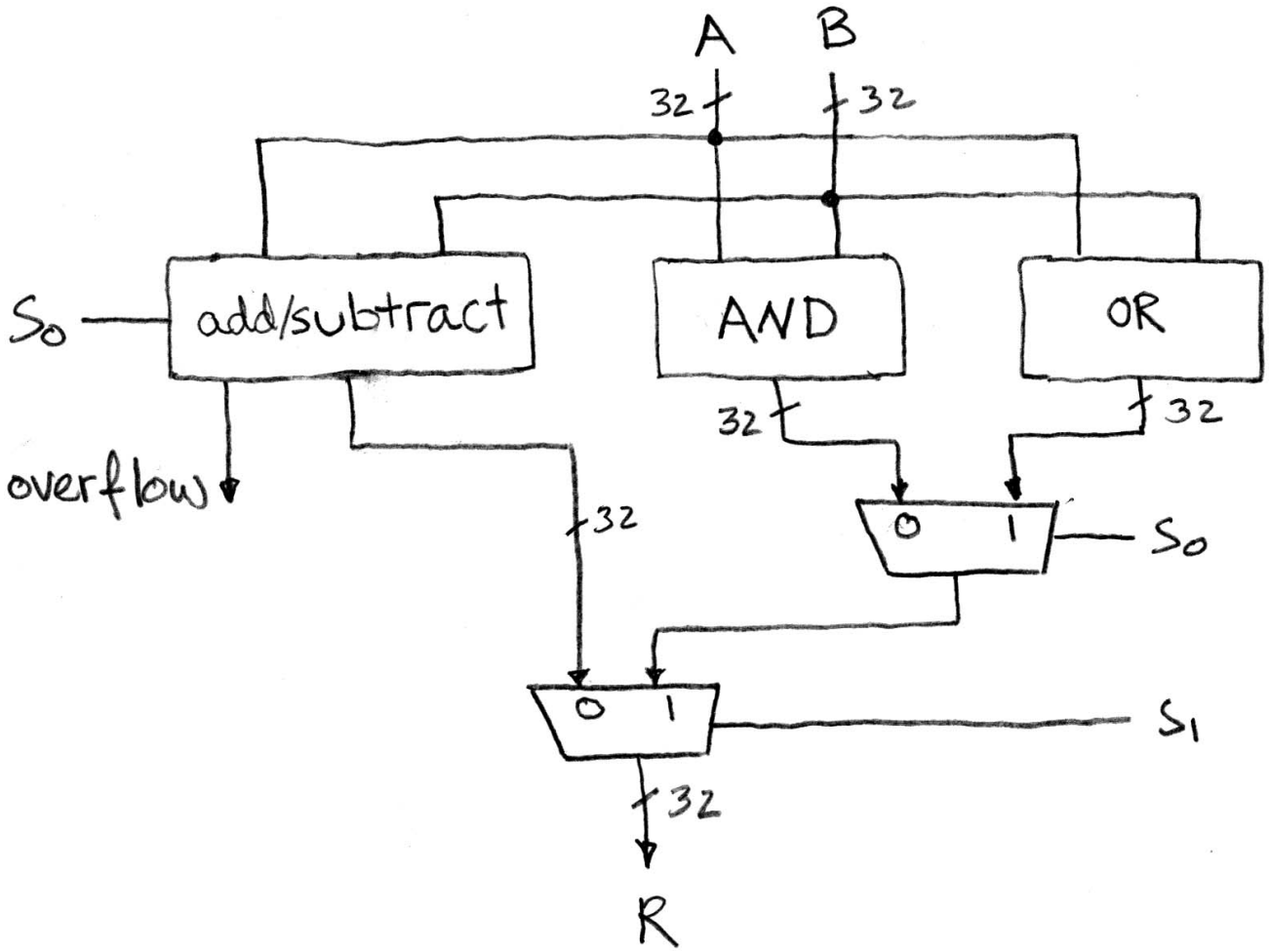
- Controls the flow of data.
- Bit-width of control signal determined by *number* of inputs to choose between, not the bit-width of the input.

Review: ALU



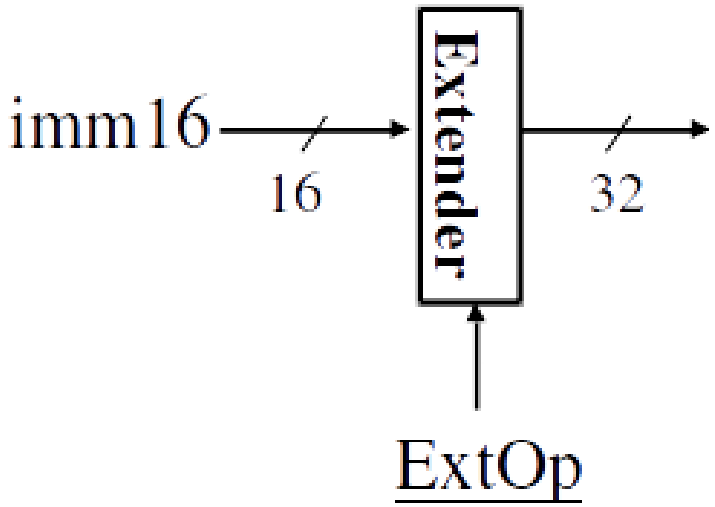
The control signal **OP** determines which arithmetic or logical operation is actually performed.

Review: ALU Implementation



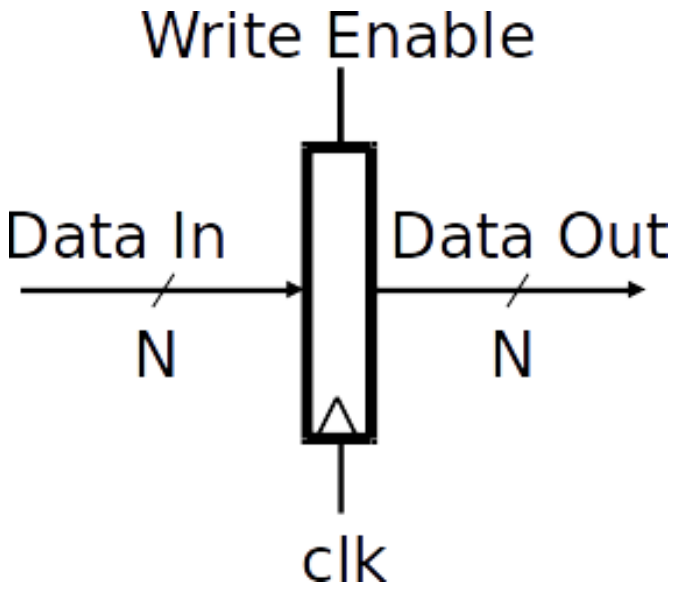
One possible ALU implementation. Do all of the operations, and control signal just controls a MUX to output.

Controlling Number Extenders



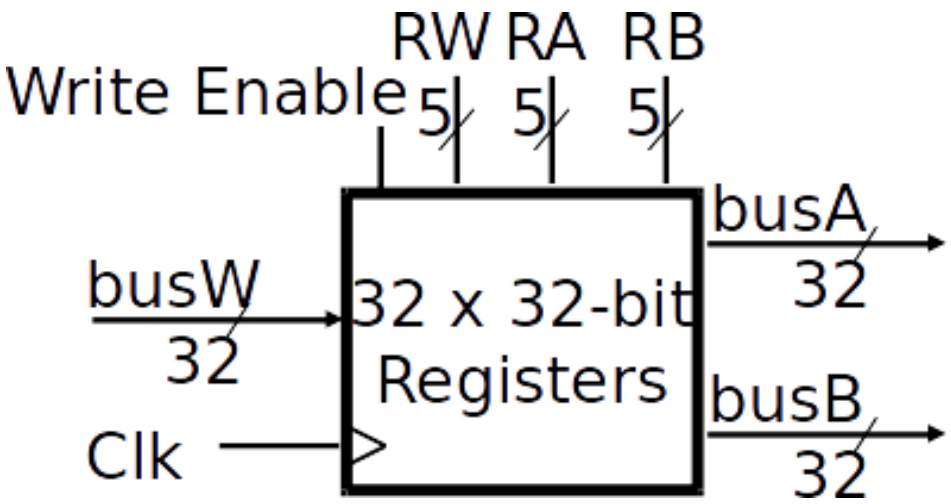
- **Extender:** A circuit which extends the bit-width of wire while maintaining its numerical value.
- *Recall:* we have both unsigned and signed numbers.
- Need a control signal to determine which to perform: `ExtOp`.

Controlling Data Storage: Register



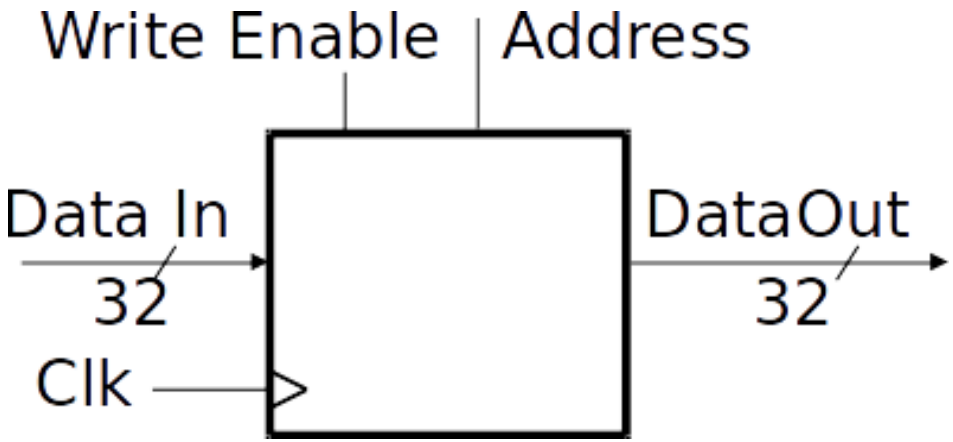
- Normally, registers are controlled by the clock.
- But, we can have special registers whose states are only updated when a special control signal is activated.
- These registers are updated when the control signal is 1 and the clock tic occurs simultaneously.

Controlling Data Storage: Many Registers



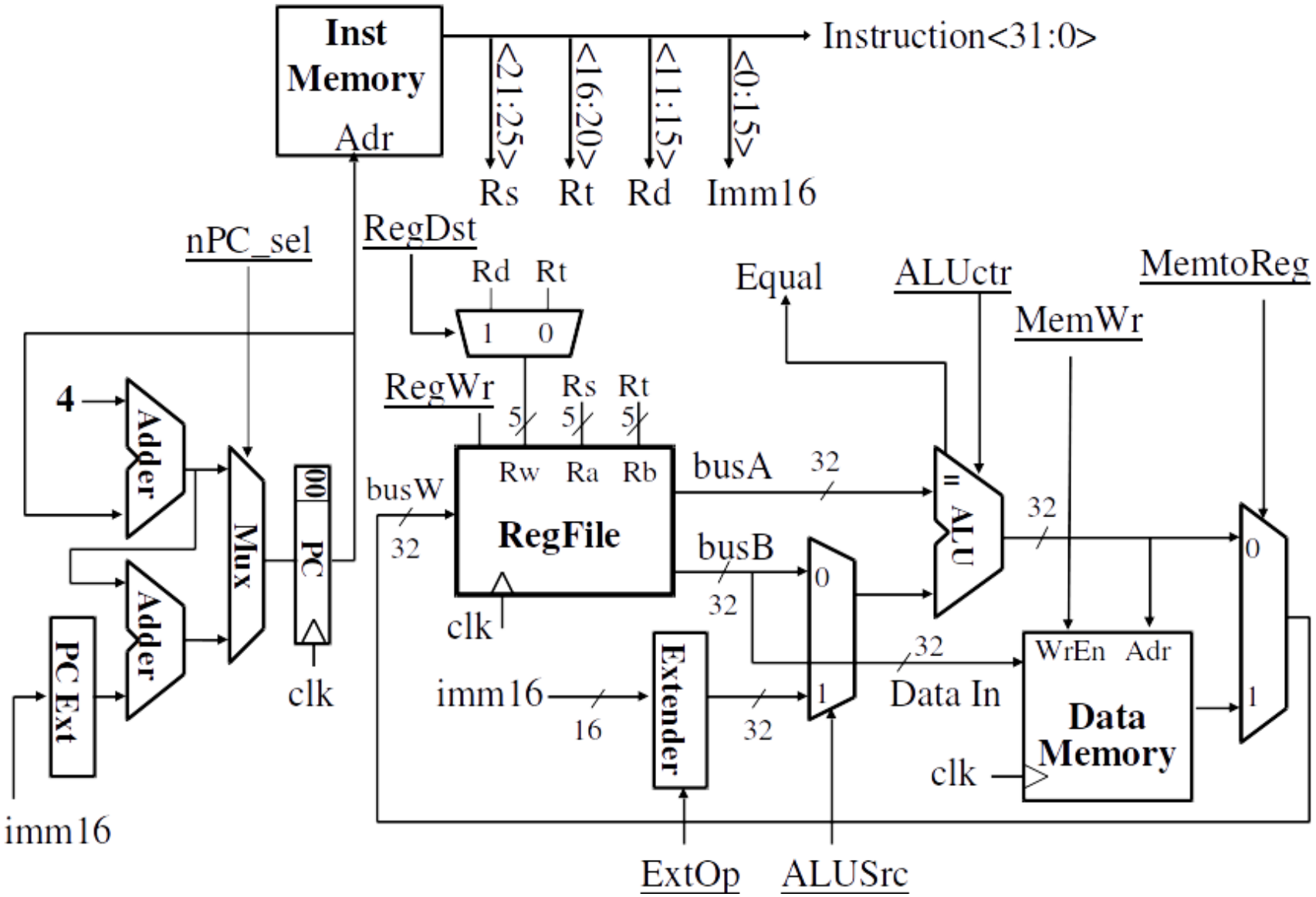
- A **register file** is a collection of registers put together.
- `RA` and `RB` are the *indices* of the registers we want to read from.
- `RW` is the *index* of the register we want to write to.
  - ↳ On the clock, **if** write enable control signal is 1, then write the data on `busW` to register `RW`.
- Clock does not affect *reads*, only *writes*.

Controlling Data Storage: Data Memory



- A simplified **data memory** works much like a register file.
- `Address` specifies the memory address to read from or write to.
- `DataOut` is the data read from memory.
- `DataIn` is the data to be written.
- A write only occurs on the clock tic and when `WriteEnable` is 1.
- Clock does not affect reads.

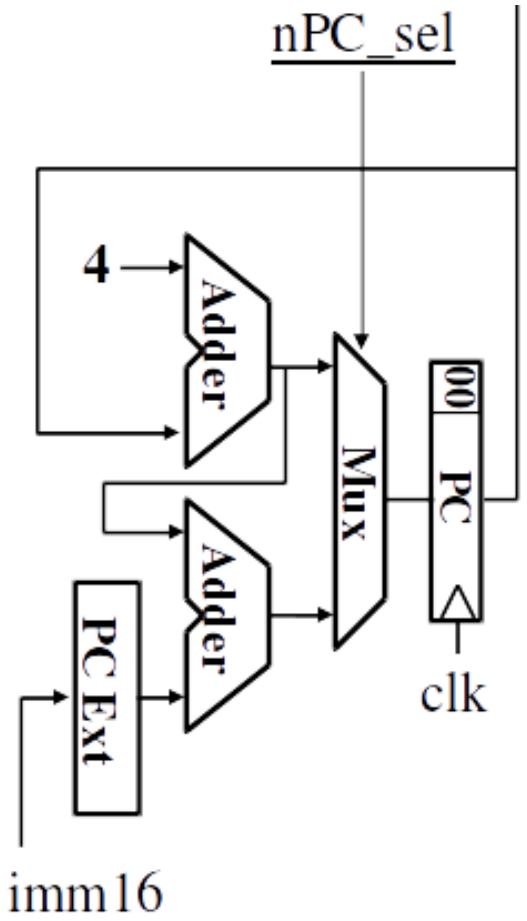
MIPS with Control Signals



Outline

Controlling "Instruction Fetch Unit" and PC

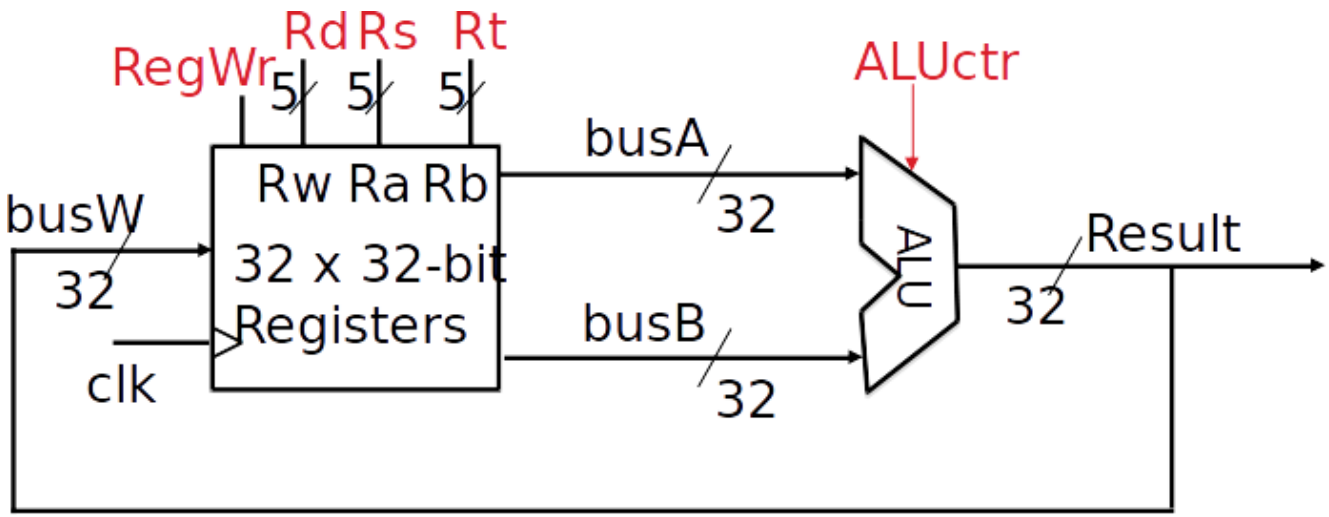
- For most instructions simply perform `PC = PC + 4`.
- For `branch` inst. we must do a special extension of the immediate value and then add it to PC..
- The next PC is actually decided by MUX and the `nPC_sel` control signal.
- If the branch condition evaluates to true, then the control signal is set to 1 and the MUX chooses the branch address.



Recall: on branch instructions `PC = PC + 4 + (imm << 2)`.
The + 4 will become clear in next chapter.

Tracing add (1/2)

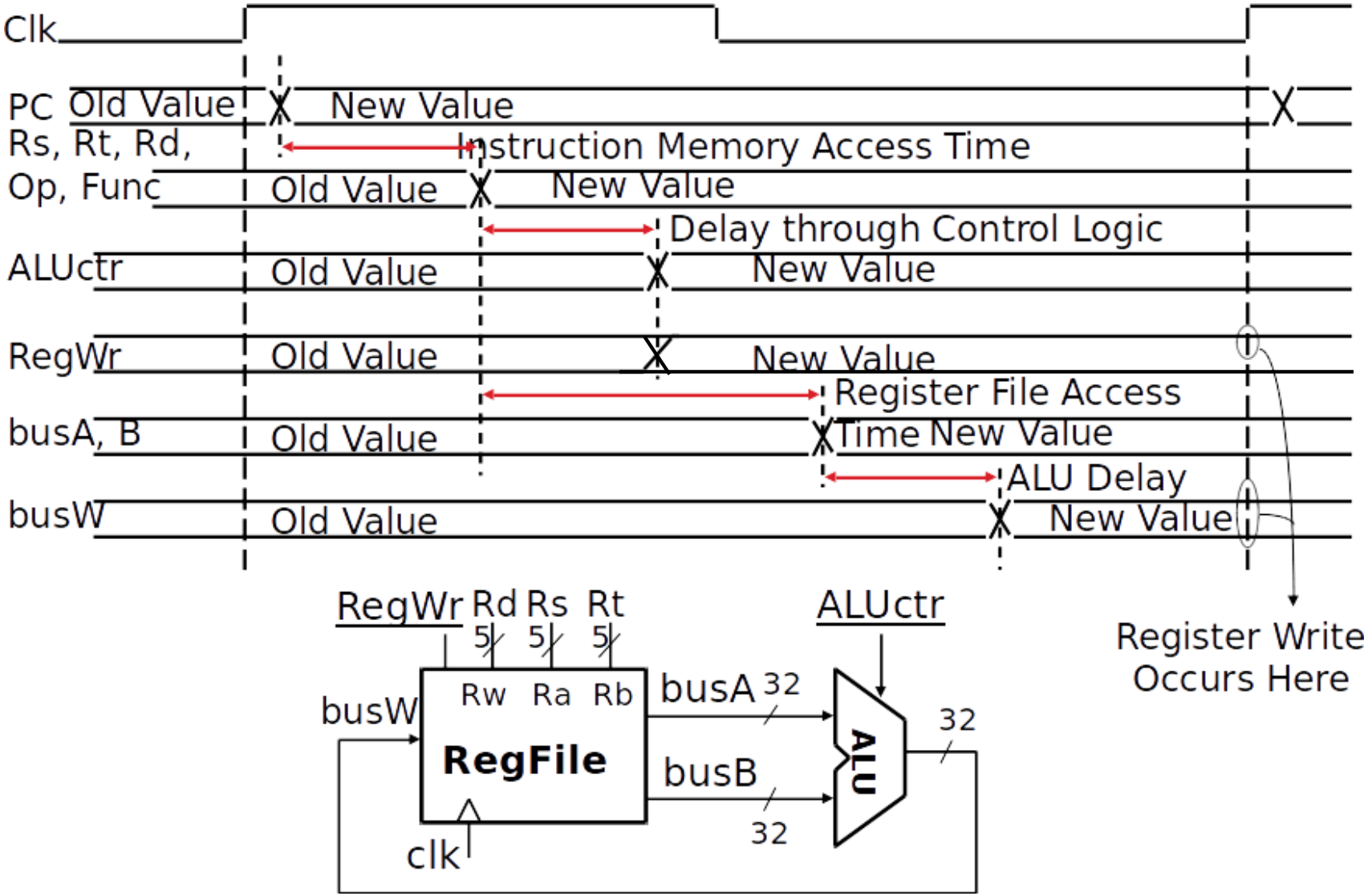
```
add $rd, $rs, $rt
```



- Inst. *writes* to a register so the `RegWr` control signal must be true.
- The `ALUctr` signal is decided from the instruction ⇒ `op` and `funct`.
- `add`, `addu`, `sub`, `subu`, `or`, `and`, . . . , all have opcode = 0 but a different `funct`.

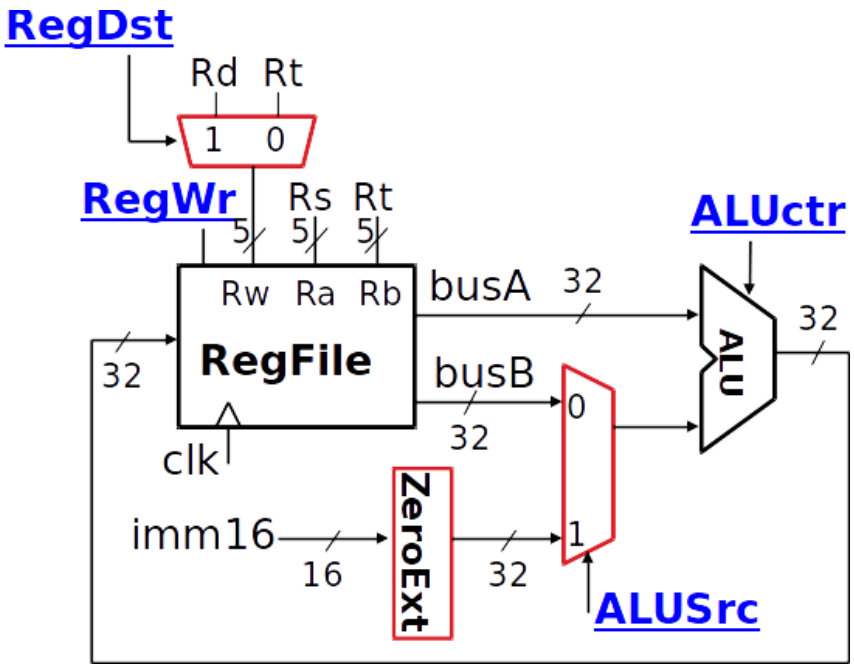
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Tracing add (2/2)



Tracing addui

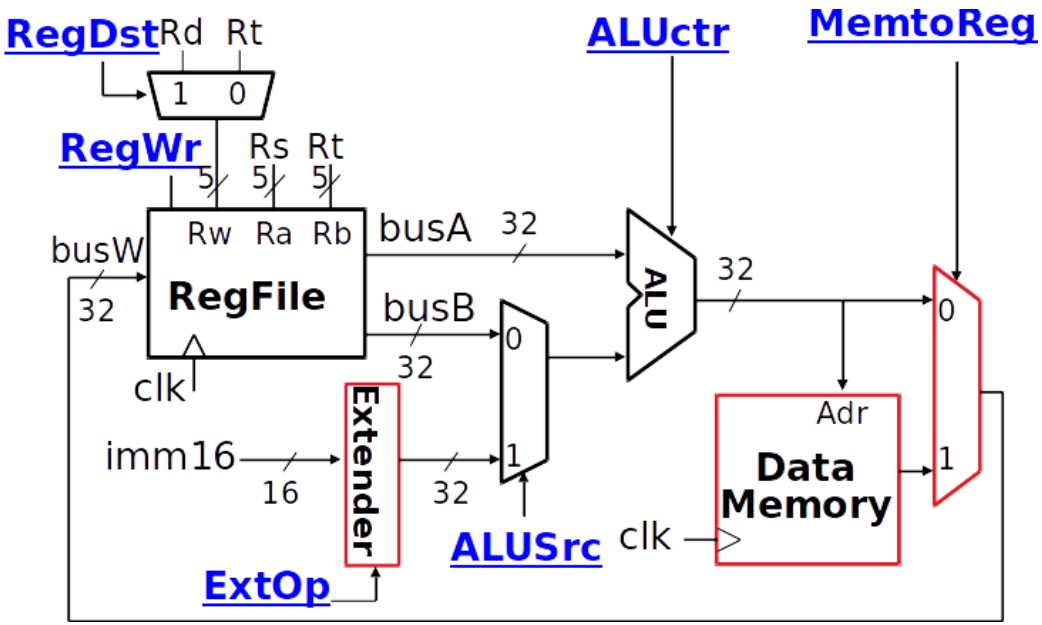
```
addui $rt, $rs, imm
```



- Modify previous path to allow register or immediate as input to ALU.
- Modify previous path to allow write to `rd` or `rt`.
- R-type inst. have `RegDst` = 'rd'; I-type have `RegDst` = 'rt'.
- R-type have `ALUSrc` = 'rt' or 'busB'; I-type have `ALUSrc` = 'imm.'.

Tracing `lw`

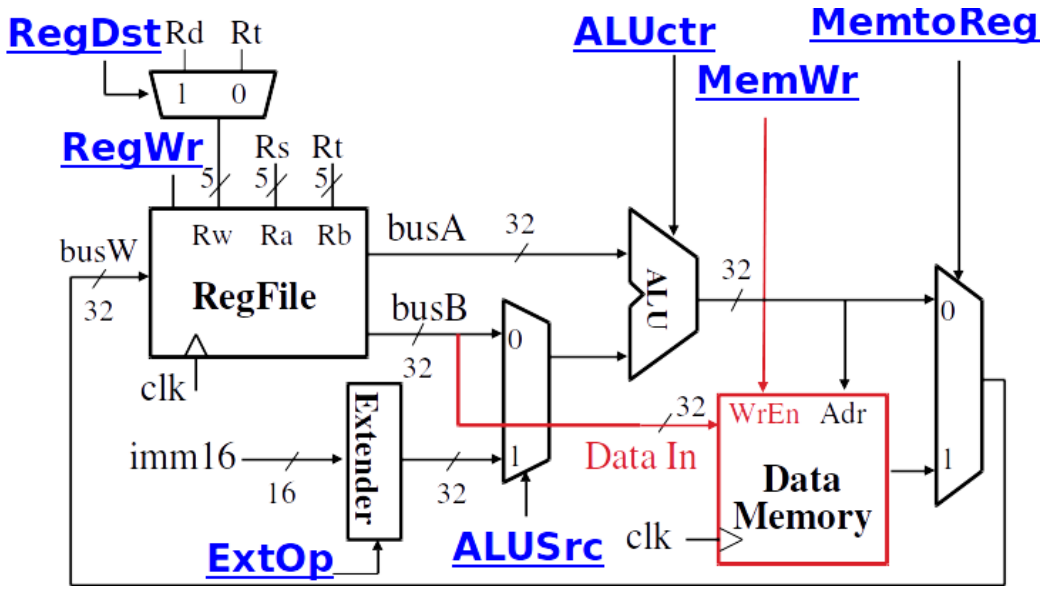
```
lw $rt, off($rs)
```



- Add `ExtOp` to allow for negative immediates.
- Add `MemToReg` to choose between ALU result and data read from memory.
- Arithmetic still occurs with `$rs` + `off` to get memory address.

Tracing sw

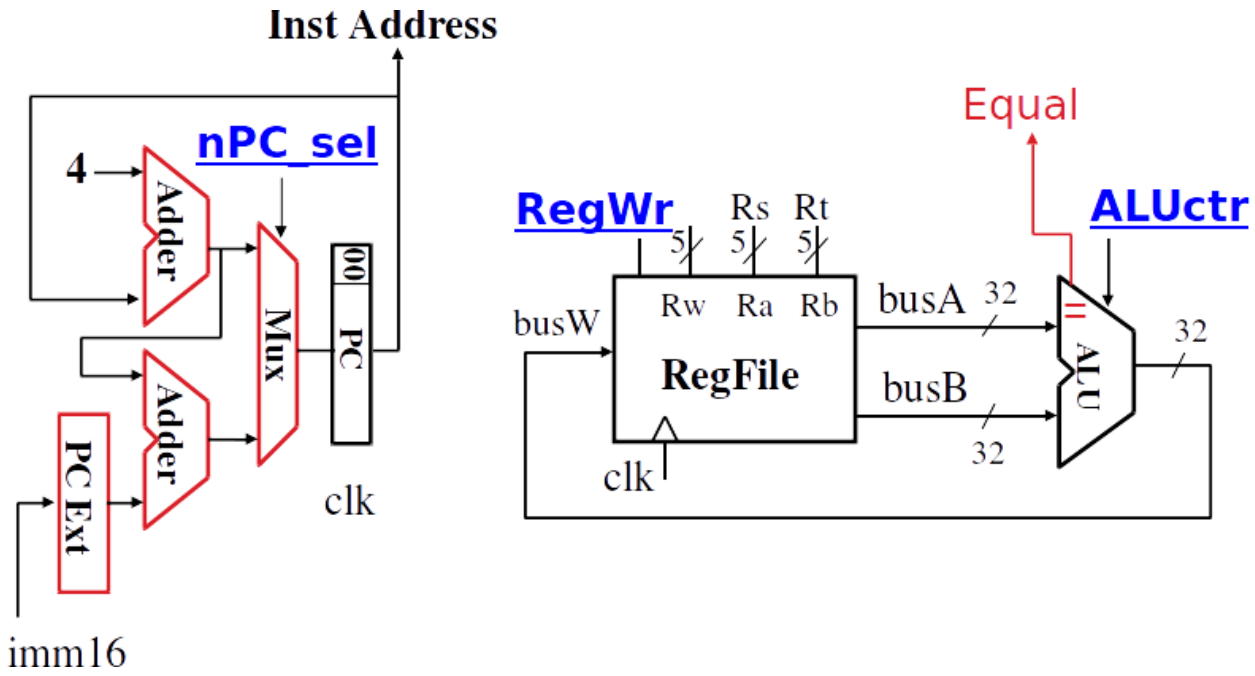
```
sw $rt, off($rs)
```



- Add a wire direct from register file to data memory.
- Add `MemWr` control to only write the read register value on a `store` instruction.
- Arithmetic still occurs with `$rs` + `off` to get memory address.

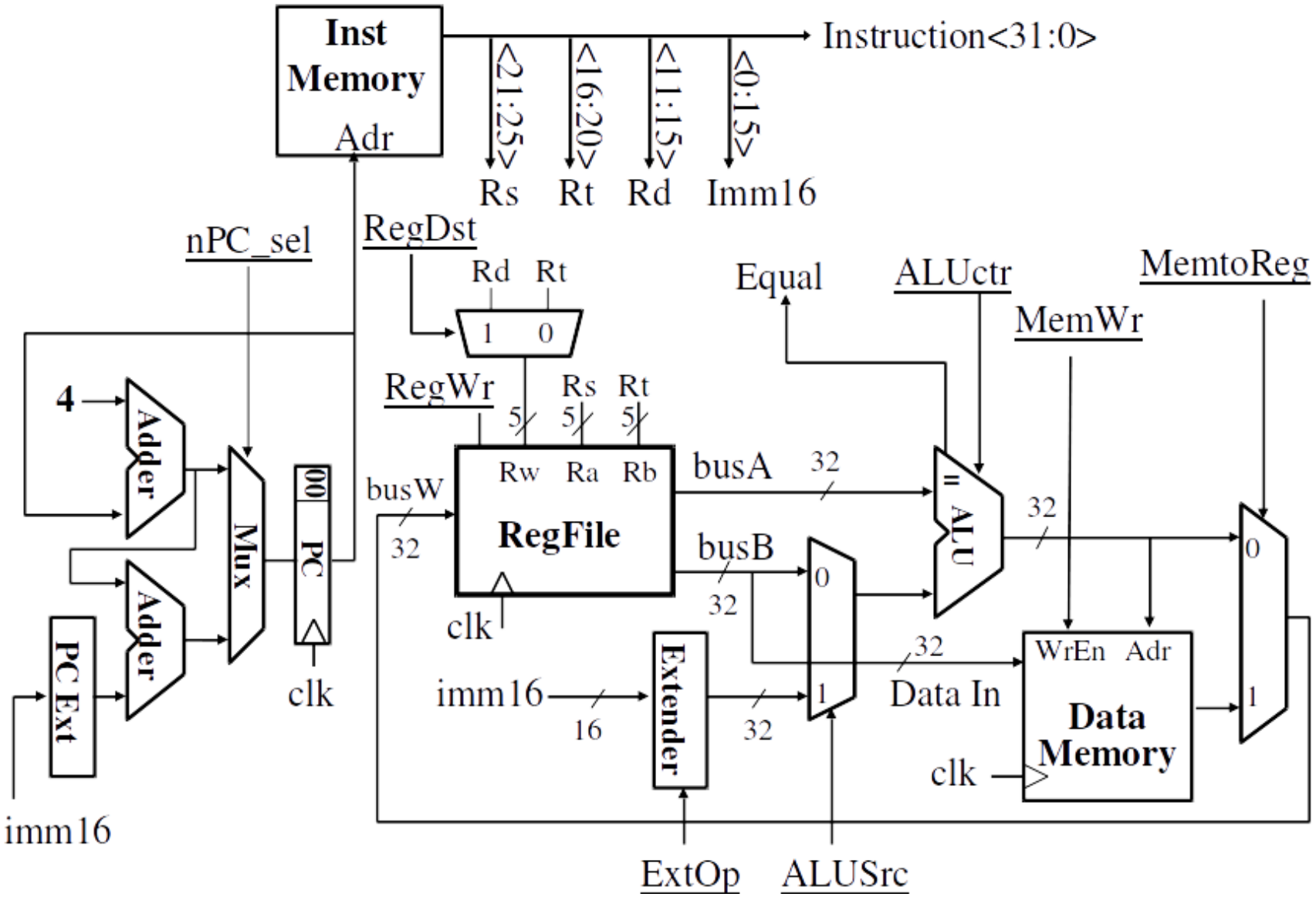
Controlling Instruction Fetch Unit: Branch instructions

`beq $rt, $rs, imm.`

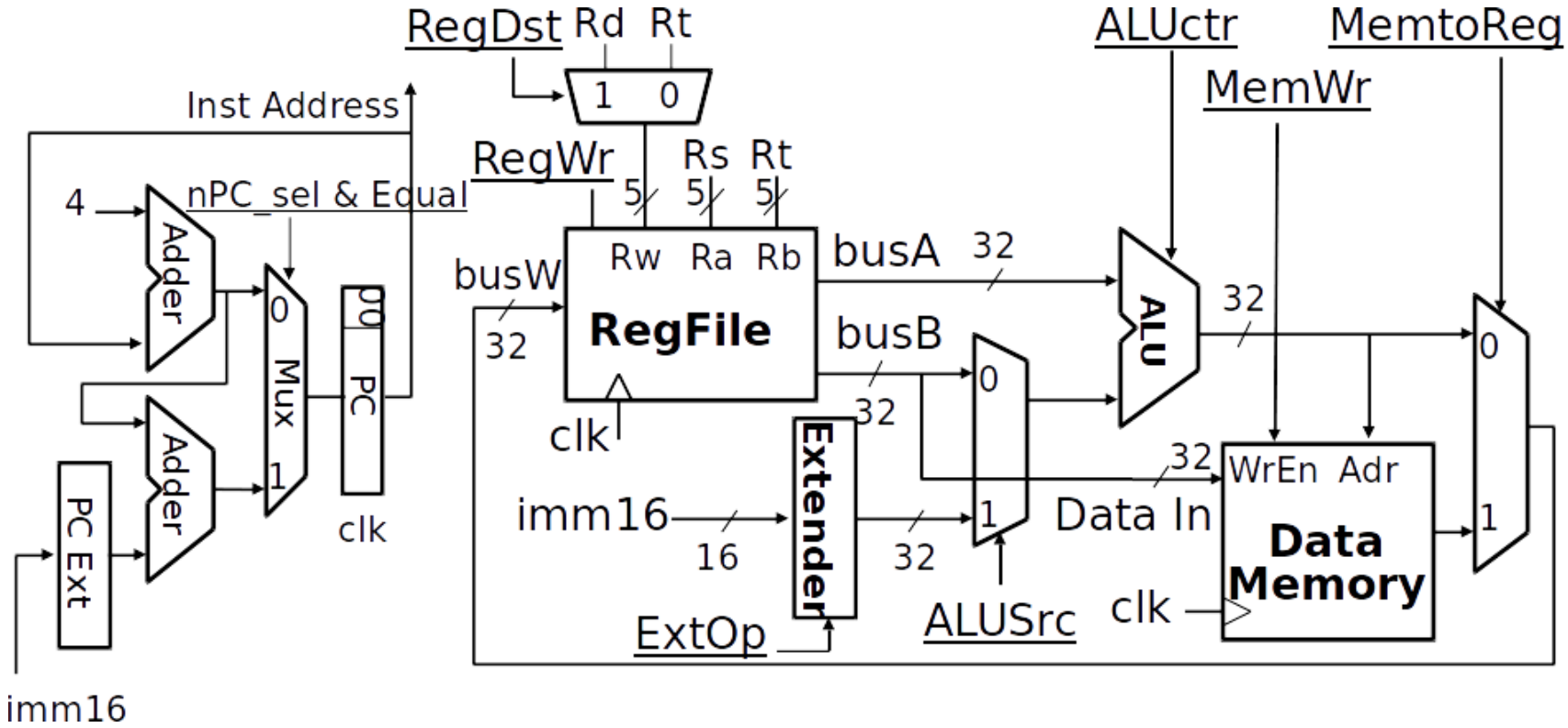


- `nPC_sel` ≡ `Equal` ∧ (`opcode` is a branch)
- If branch condition fails (if `$rs` ≠ `$rt`) or instruction is not a branch type, `PC = PC + 4`.
- Remember: datapath generally computes everything, control signals determine which results are actually read/redirected/stored/etc.

Cumulative Datapath with Control Signals



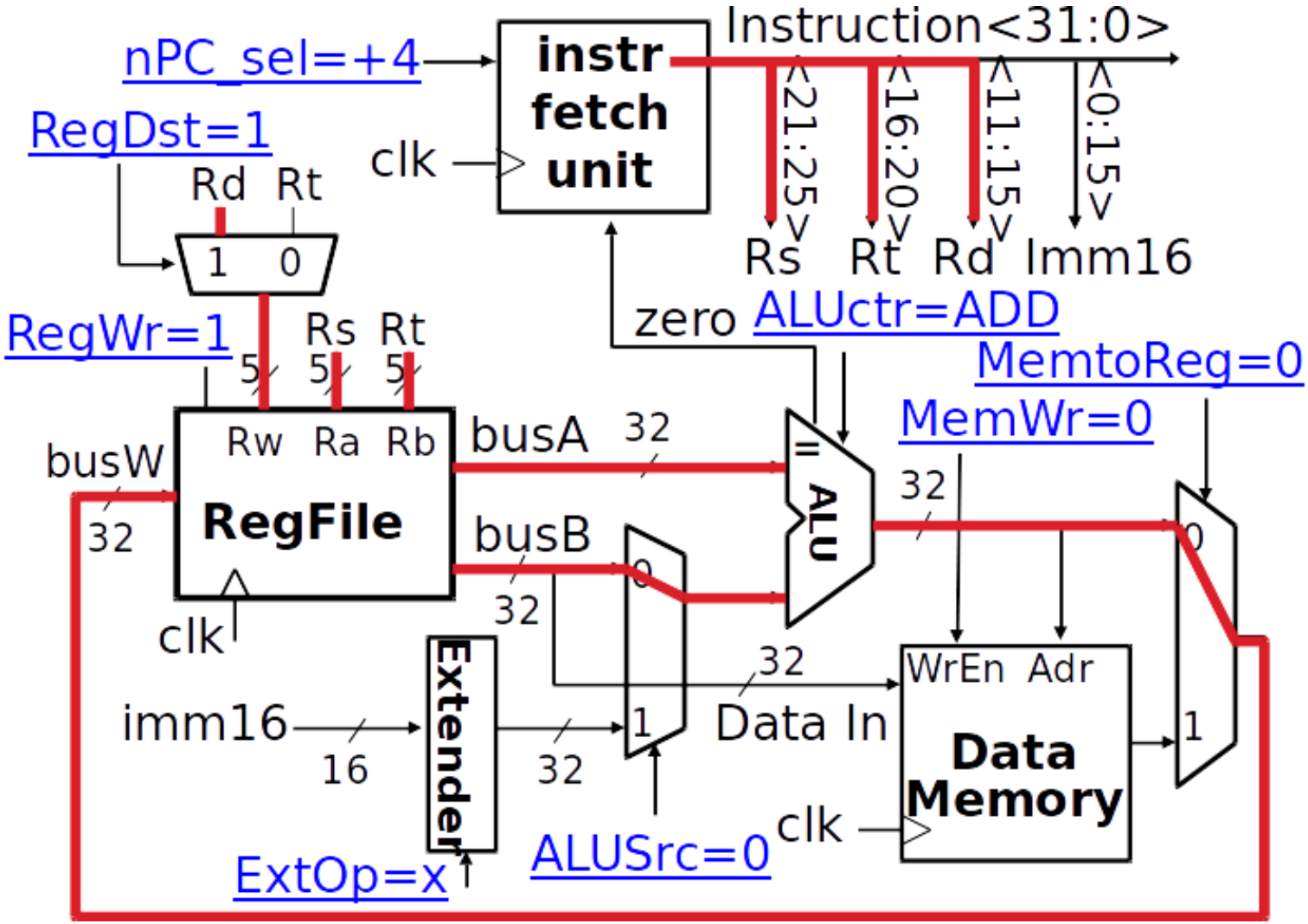
Control Signals Values



- nPC_sel: '+4', 'branch'
- RegDst: 'rd', 'rt'
- RegWr: 1 ⇒ 'write'
- ExtOp: 'zero', 'signed'
- ALUSrc: 'rt'/'busB', imm.
- MemWr: 1 ⇒ 'write'
- MemToReg: 'ALU', 'Mem'
- ALUCtr: 'add', 'sub', '<', '>', '==', '!=', 'or', 'and',

Tracing add in full

```
add $rd, $rs, $rt
```



Summary of Control Signals

func	10 0000	10 0010	Doesn't Matter				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	`add`	`sub`	`ori`	`lw`	`sw`	`beq`	`jump`
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
nPC_sel	0	0	0	0	0	1	?
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr	Add	Subtract	Or	Add	Add	Equal	x

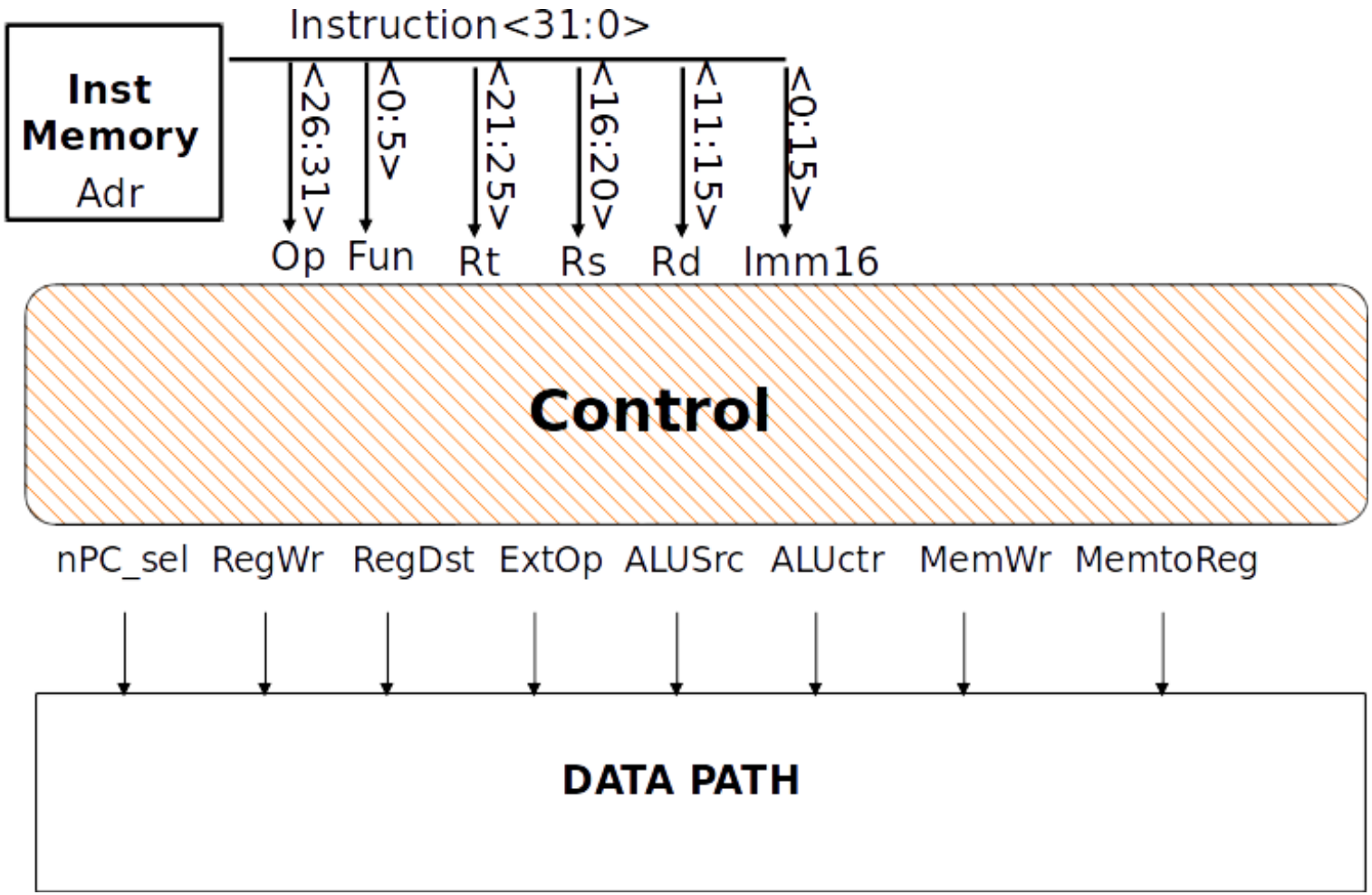
x = Don't care / Doesn't matter

Note: numeric values not really important. Just gives semantic meaning.

- ↳ e.g. `RegDst` = '`rd`'
- ↳ e.g. `nPC_sel` = '`branch`'

Outline

The Controller: Many Inputs, Many Ouputs

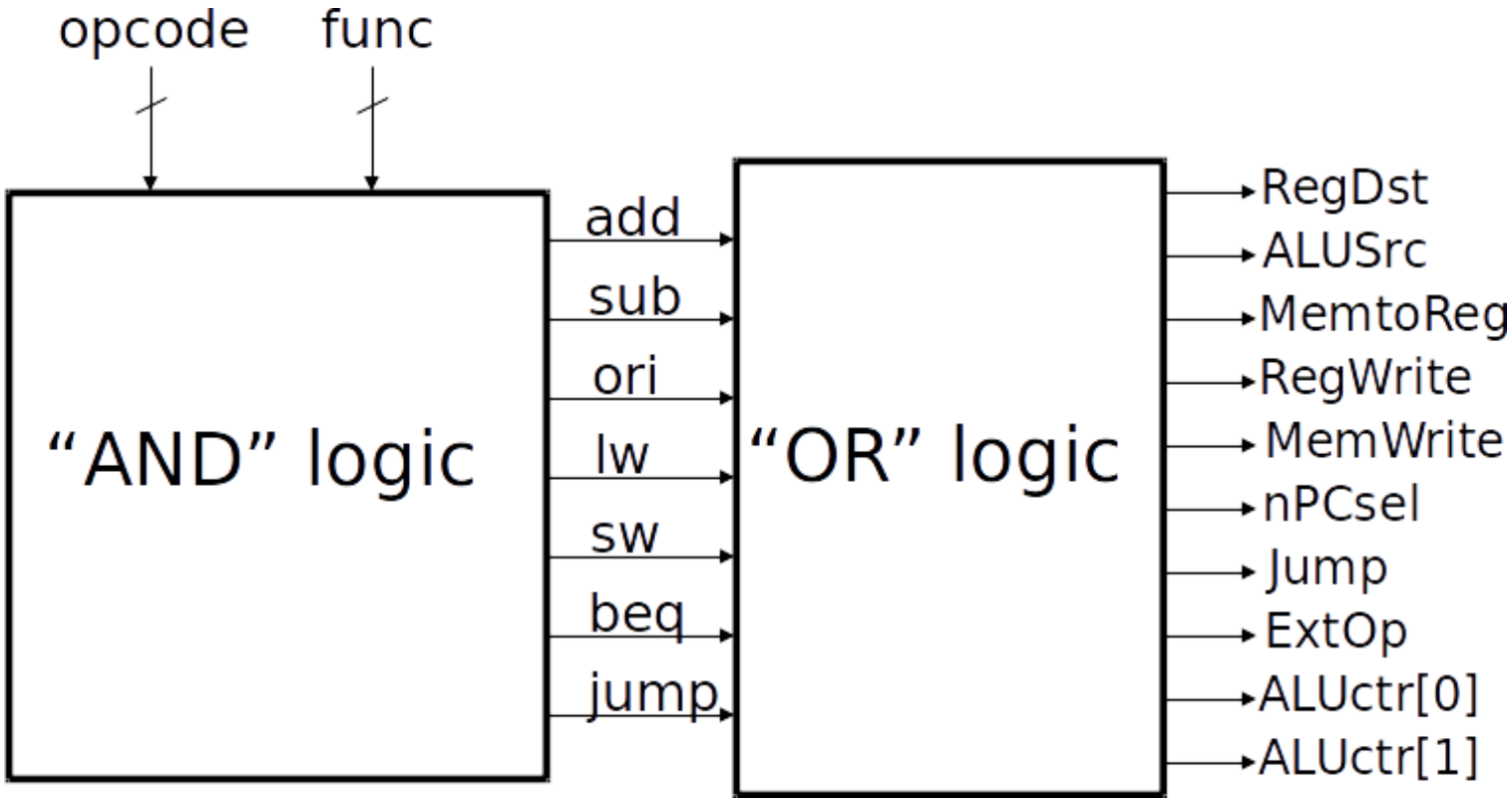


Possible Boolean Expressions for Controller

RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPCsel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq (assume ALUctr is 00 ADD, 01 SUB, 10 OR)
ALUctr[1] = or

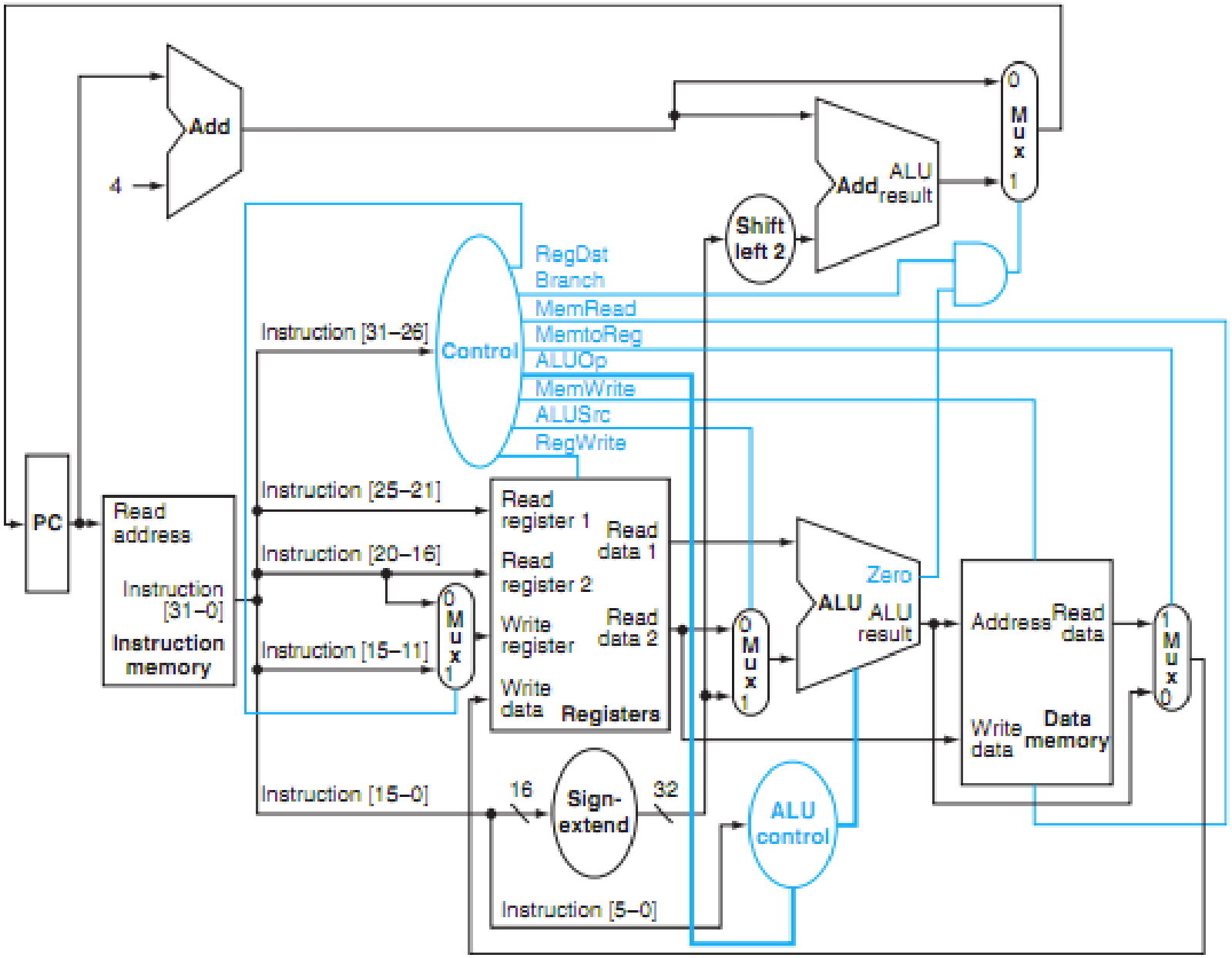
$$\text{rtype} = \overline{op_5} \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot \overline{op_1} \cdot \overline{op_0},$$
$$\text{ori} = \overline{op_5} \cdot \overline{op_4} \cdot op_3 \cdot op_2 \cdot \overline{op_1} \cdot op_0$$
$$\text{lw} = op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1 \cdot op_0$$
$$\text{sw} = op_5 \cdot \overline{op_4} \cdot op_3 \cdot \overline{op_2} \cdot op_1 \cdot op_0$$
$$\text{beq} = \overline{op_5} \cdot \overline{op_4} \cdot \overline{op_3} \cdot op_2 \cdot \overline{op_1} \cdot \overline{op_0}$$
$$\text{jump} = \overline{op_5} \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1 \cdot \overline{op_0}$$
$$\text{add} = \text{rtype} \cdot func_5 \cdot \overline{func_4} \cdot \overline{func_3} \cdot \overline{func_2} \cdot \overline{func_1} \cdot \overline{func_0}$$
$$\text{sub} = \text{rtype} \cdot func_5 \cdot \overline{func_4} \cdot \overline{func_3} \cdot \overline{func_2} \cdot func_1 \cdot \overline{func_0}$$

Implementing The Controller



Look familiar? Same scheme as a programmable logic array (PLA).

Patterson & Hennessy: Controller



Single Cycle Processor: Summary

- Instruction Set Architecture $\leftrightarrow$ Datapath.
  - ↳ Instructions determine circuits needed in datapath.
  - ↳ Limitations of circuits influence allowable instructions.
- Classic RISC Datapath: `IF`, `ID`, `EX`, `MEM`, `WB`.
- Clock cycle must be long enough to account for time of *critical path* through datapath.
- MUX control flow of data through datapath.
- Controller takes opcode and `funct` as input, outputting the control signals that control MUXs, ALU, writing.
  - ↳ Boolean logic here is complex must account for every possibly combination of instructions and data.