RISC-V Architecture
&
Processor Design

Week 4

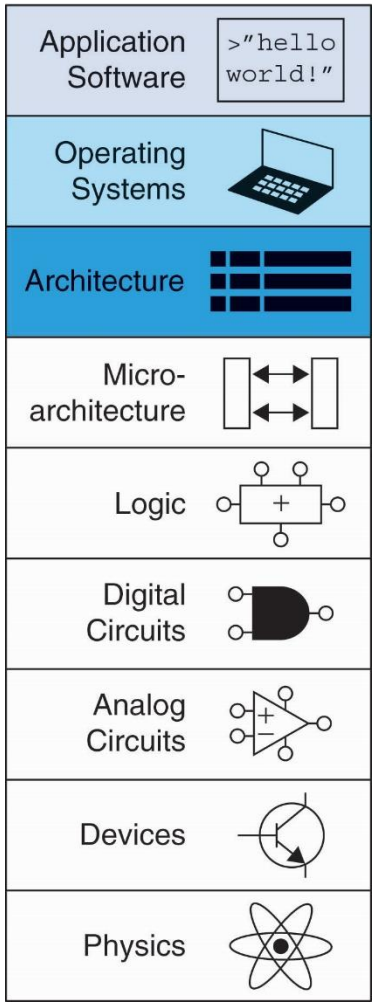
RISC-V ISA

and

RISC-V uArchitecture

RISC-V ISA (Cont'ed)

- **Lights, Camera, Action: Compiling, Assembly, & Loading**
- **Odds & Ends**



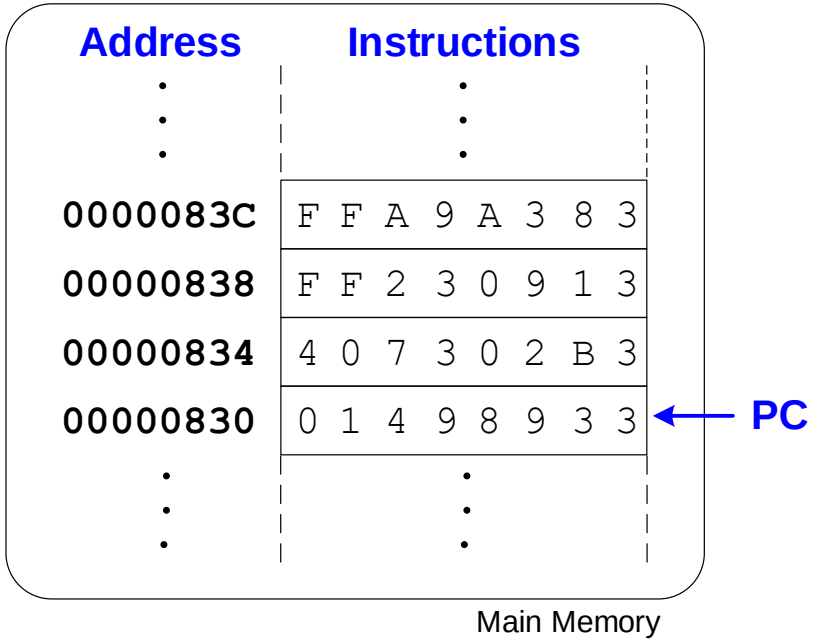
Compiling, Assembling, & Loading Programs

The Power of the Stored Program

- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation

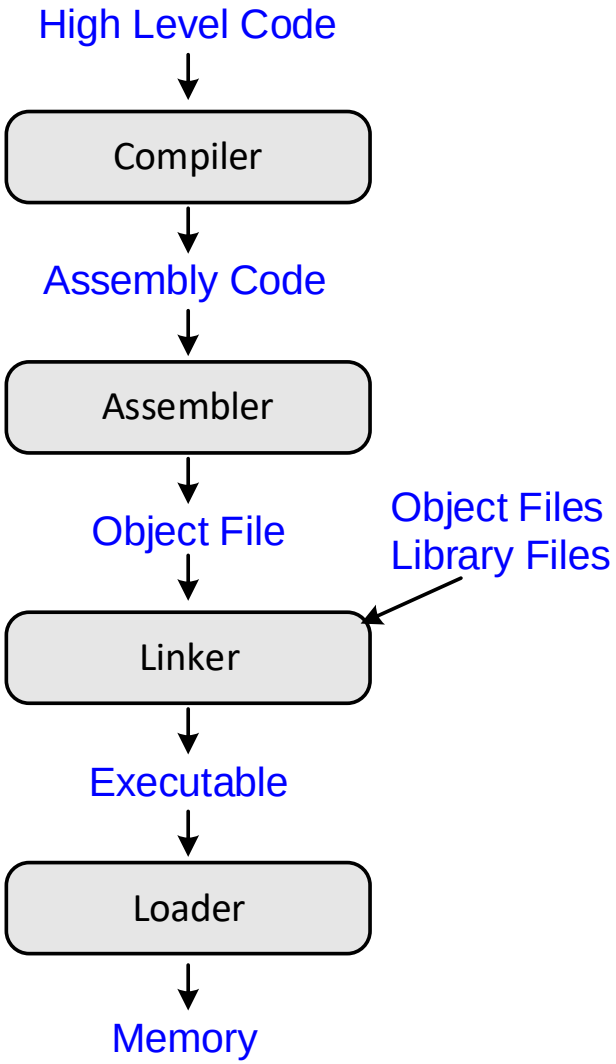
The Stored Program

Assembly Code	Machine Code
`add s2, s3, s4`	`0x01498933`
`sub t0, t1, t2`	`0x407302B3`
`addi s2, t1, -14`	`0xFF230913`
`lw t2, -6(s3)`	`0xFFA9A383`



Program Counter (PC): keeps track of current instruction

How to Compile & Run a Program

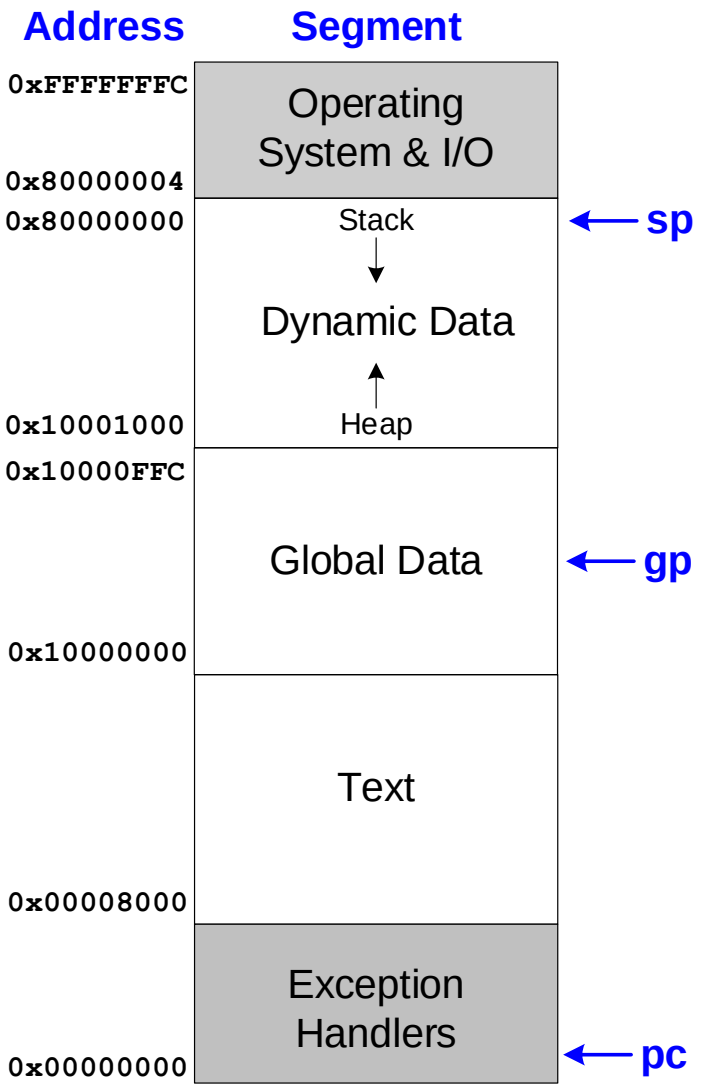


What is Stored in Memory?

- **Instructions** (also called *text*)
- **Data**
  - **Global/static:** allocated before program begins
  - **Dynamic:** allocated within program

- How **big** is memory?
  - At most 2^{32} = 4 gigabytes (4 GB)
  - From address 0x00000000 to 0xFFFFFFFF

Example RISC-V Memory Map



Example Program: C Code

```
int f, g, y; // global variables

int func(int a, int b) {
  if (b < 0)
    return (a + b);
  else
    return(a + func(a, b-1));
}

void main() {
  f = 2;
  g = 3;
  y = func(f,g);

  return;
}
```

Example Program: RISC-V Assembly

Address	Machine Code		RISC-V Assembly Code
10144:	ff010113	func:	addi sp,sp,-16
10148:	00112623		sw ra,12(sp)
1014c:	00812423		sw s0,8(sp)
10150:	00050413		mv s0,a0
10154:	00a58533		add a0,a1,a0
10158:	0005da63		bgez a1,1016c <func+0x28>
1015c:	00c12083		lw ra,12(sp)
10160:	00812403		lw s0,8(sp)
10164:	01010113		addi sp,sp,16
10168:	00008067		ret
1016c:	fff58593		addi a1,a1,-1
10170:	00040513		mv a0,s0
10174:	fd1ff0ef		jal ra,10144 <func>
10178:	00850533		add a0,a0,s0
1017c:	fe1ff06f		j 1015c <func+0x18>

Maintain **4-word alignment** of **`sp`** (for compatibility with RV128I) even though only space for 2 words needed.

Pseudoinstructions:
`mv`: `addi a0, s0, 0`
`ret` (return): `jr ra`

Example Program: RISC-V Assembly

Address	Machine Code		RISC-V Assembly Code
10180:	ff010113	main:	addi sp,sp,-16
10184:	00112623		sw ra,12(sp)
10188:	00200713		li a4,2
1018c:	c4e1a823		sw a4,-944(gp) # 11a30 <f>
10190:	00300713		li a4,3
10194:	c4e1aa23		sw a4,-940(gp) # 11a34 <g>
10198:	00300593		li a1,3
1019c:	00200513		li a0,2
101a0:	fa5ff0ef		jal ra,10144 <func>
101a4:	c4a1ac23		sw a0,-936(gp) # 11a38 <y>
101a8:	00c12083		lw ra,12(sp)
101ac:	01010113		addi sp,sp,16
101b0:	00008067		ret

gp = 0x11DE0

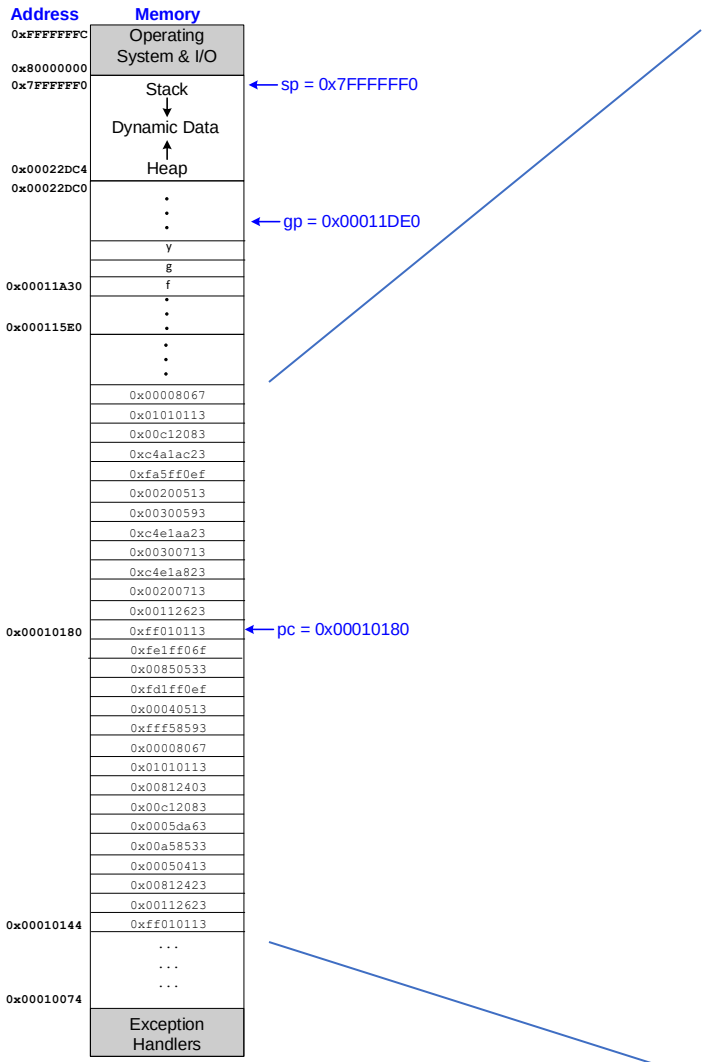
Put 2 and 3 in `f` and `g` (and argument registers) and call `func`. Then put result in `y` and return.

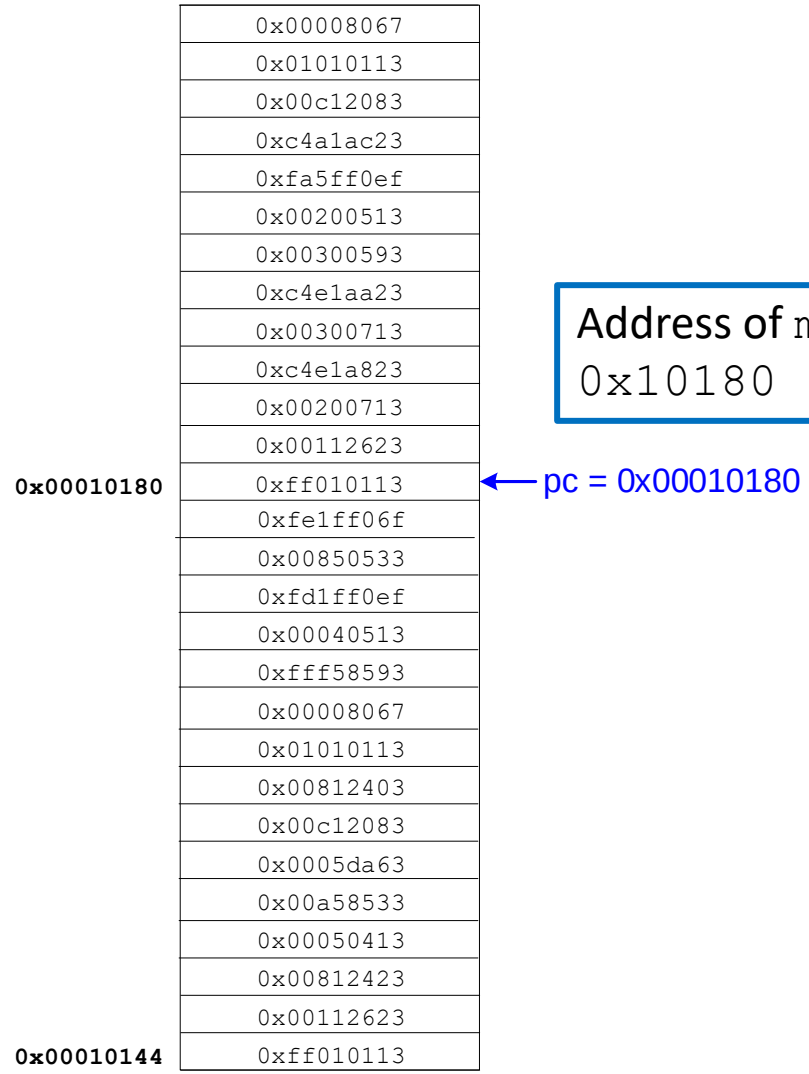
Example Program: Symbol Table

Address				Size	Symbol Name
00010074	l	d	.text	00000000	.text
000115e0	l	d	.data	00000000	.data
00010144	g	F	.text	0000003c	func
00010180	g	F	.text	00000034	main
00011a30	g	O	.bss	00000004	f
00011a34	g	O	.bss	00000004	g
00011a38	g	O	.bss	00000004	y

- text segment: address 0x10074
- data segment: address 0x115e0
- func function: address 0x10144 (size 0x3c bytes)
- main function: address 0x10180 (size 0x34 bytes)
- f: address 0x11a30 (size 0x4 bytes)
- g: address 0x11a34 (size 0x4 bytes)
- y: address 0x11a38 (size 0x4 bytes)

Example Program in Memory



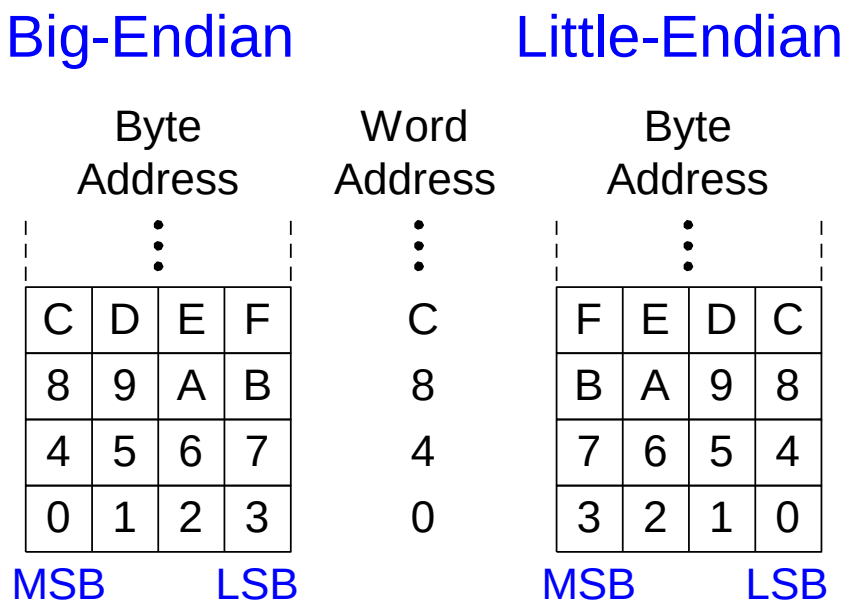


Address of `main`: `0x10180`

Endianness

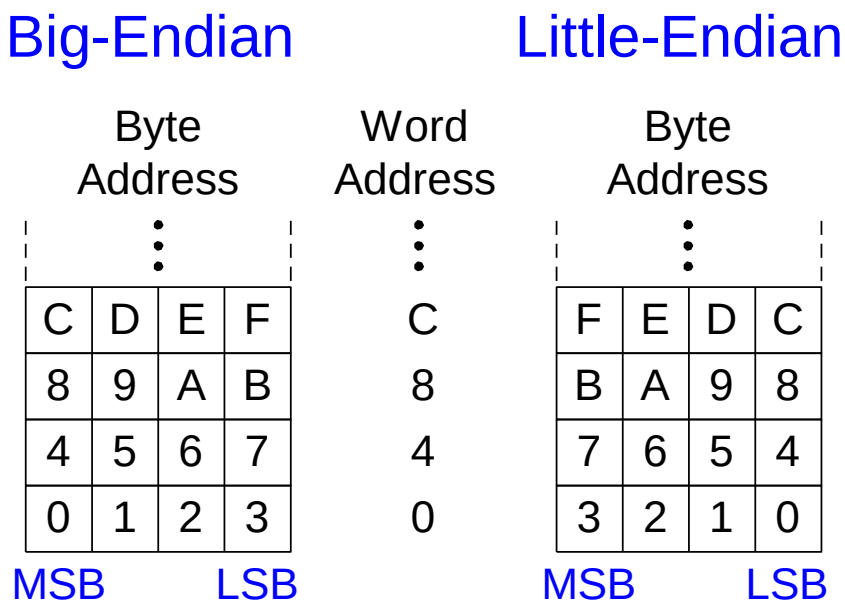
Big-Endian & Little-Endian Memory

- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian



Big-Endian & Little-Endian Memory

- Jonathan Swift's *Gulliver's Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn't really matter which addressing type used – except when the two systems need to share data!



Big-Endian & Little-Endian Example

- Suppose `t0` initially contains 0x23456789
- After following code runs on **big-endian** system, what value is `s0`?
- In a **little-endian** system?

```
sw t0, 0(zero)
lb s0, 1(zero)
```

- **Big-endian:** `s0` = 0x00000045
- **Little-endian:** `s0` = 0x00000067

Signed & Unsigned Instructions

Signed & Unsigned Instructions

- Multiplication and division
- Branches
- Set less than
- Loads
- Detecting overflow

Multiplication

- **Signed:** `mulh`
- **Unsigned:** `mulhu`, `mulhsu`
  - `mulhu`: treat both operands as unsigned
  - `mulhsu`: treat first operand as signed, second as unsigned
  - 32 lsbs are identical whether signed/unsigned; use mul

Example: s1 = 0x80000000; s2 = 0xC0000000

```
mulh s4, s1, s2
mul  s3, s1, s2
```

$s1 = -2^{31}$; $s2 = -2^{30}$
$s1 \times s2 = 2^{61}$
s4 = 0x20000000
s3 = 0x00000000

```
mulhu s4, s1, s2
mul   s3, s1, s2
```

$s1 = 2^{31}$; $s2 = 3 \times 2^{30}$
$s1 \times s2 = 3 \times 2^{61}$
s4 = 0x60000000
s3 = 0x00000000

```
mulhsu s4, s1, s2
mul    s3, s1, s2
```

$s1 = -2^{31}$; $s2 = 3 \times 2^{30}$
$s1 \times s2 = -3 \times 2^{61}$
s4 = 0xA0000000
s3 = 0x00000000

Division & Remainder

- **Signed:** `div, rem`
- **Unsigned:** `divu, remu`

Branches

- **Signed:** `blt, bge`
- **Unsigned:** `bltu, bgeu`

Examples: `s1` = 0x80000000; `s2` = 0x40000000

```
blt  s1, s2
```

`s1` = -2^{31}; s2 = 2^{30}

`taken`

```
bltu s1, s2
```

`s1` = 2^{31}; s2 = 2^{30}

`not taken`

Set Less Than

- **Signed:** `slt, slti`
- **Unsigned:** `sltu, sltiu`

Note: RISC-V always **sign-extends** the immediate, even for `sltiu`

Examples: `s1` = 0x80000000; `s2` = 0x40000000

```
slt  t0, s1, s2
```
`s1` = -2^{31}; s2 = 2^{30}
`t0` = 1

```
slti  t2, s1, -1   # -1 = 0xFFF
```
`s1` = -2^{31}; imm = 0xFFFFFFFF = -1
`t2` = 1

```
sltu t1, s1, s2
```
`s1` = 2^{31}; s2 = 2^{30}
`t1` = 0

```
sltiu t3, s1, -1   # -1 = 0xFFF
```
`s1` = 2^{31}; imm = 0xFFFFFFFF = 2^{32} - 1
`t3` = 1

Loads

- **Signed:**
  - Sign-extends to create 32-bit value to load into register
  - Load halfword: `lh`
  - Load byte: `lb`
- **Unsigned:**
  - Zero-extends to create 32-bit value
  - Load halfword unsigned: `lhu`
  - Load byte: `lbu`

Detecting Overflow

- RISC-V does not provide unsigned addition or instructions or overflow detection because it can be done with existing instructions:
- **Example: Detecting unsigned overflow:**

```
add  t0, t1, t2
bltu t0, t1, overflow
```

- **Example: Detecting signed overflow:**

```
add  t0, t1, t2
slti t3, t2, 0          # t3=1 if t2 neg.
slt  t4, t0, t1         # t4=1 if result < t1
bne  t3, t4, overflow # overflow if:
                        # t2 neg & result>=t1 or
                        # t2 pos & result<t1
```

Compressed Instructions

Compressed Instructions

- **16-bit** RISC-V instructions
- Replace common integer and floating-point instructions with 16-bit versions.
- Most RISC-V compilers/processors can use a **mix** of 32-bit and 16-bit instructions (and use 16-bit instructions whenever possible).
- Uses prefix: **c.**
- **Examples:**
  - `add` → `c.add`
  - `lw` → `c.lw`
  - `addi` → `c.addi`

Compressed Instructions Example

C Code

```
int i;
int scores[200];


for (i=0; i<200; i=i+1)

  scores[i] = scores[i]+10;
```

RISC-V assembly code

```
# s0 = scores base address, s1 = i

  c.li s1, 0              # i = 0
  addi t2, zero, 200      # t2 = 200

for:
  bge    s1, t2, done     # I >= 200? done
  c.lw   a3, 0(s0)        # a3 = scores[i]
  c.addi a3, 10           # a3 = scores[i]+10
  c.sw   a3, 0(s0)        # scores[i] = a3
  c.addi s0, 4            # next element
  c.addi s1, 1            # i = i+1
  c.j    for              # repeat
done:
```

- 200 is too big to fit in compressed immediate, so noncompressed `addi` used instead.
- **`c.addi s0,4`** is equivalent to **`addi s0,s0,4`**.
- `c.bge` doesn't exist, so `bge` is used.

Compressed Machine Formats

- Some compressed instructions use a **3-bit register code** (instead of 5-bit). These specify registers `x8` to `x15`.
- **Immediates** are 6-11 bits.
- **Opcode** is 2 bits.

Compressed Machine Formats

15 14	13 12	11 10	9 8 7	6 5	4 3 2	1 0	
funct4		rd/rs1		rs2		op	**CR-Type**
funct3	imm	rd/rs1		imm		op	**CI-Type**
funct3	imm		rs1'	imm	rs2'	op	**CS-Type**
funct6			rd'/rs1'	funct2	rs2'	op	**CS'-Type**
funct3	imm		rs1'	imm		op	**CB-Type**
funct3	imm	funct	rd'/rs1'	imm		op	**CB'-Type**
funct3	imm					op	**CJ-Type**
funct3	imm			rs2		op	**CSS-Type**
funct3	imm				rd'	op	**CIW-Type**
funct3	imm		rs1'	imm	rd'	op	**CL-Type**

Floating-Point Instructions

RISC-V Floating-Point Extensions

- RISC-V offers three floating point extensions:
  - **RVF:** single-precision (32-bit)
    - 8 exponent bits, 23 fraction bits
  - **RVD:** double-precision (64-bit)
    - 11 exponent bits, 52 fraction bits
  - **RVQ:** quad-precision (128-bit)
    - 15 exponent bits, 112 fraction bits

Floating-Point Registers

- **32** Floating point registers
- **Width** is highest precision – for example, if RVQ is implemented, registers are 128 bits wide
- When multiple floating point extensions are implemented, the lower-precision values occupy the lower bits of the register

Floating-Point Registers

Name	Register Number	Usage
`ft0-7`	f0-7	Temporary variables
`fs0-1`	f8-9	Saved variables
`fa0-1`	f10-11	Function arguments/Return values
`fa2-7`	f12-17	Function arguments
`fs2-11`	f18-27	Saved variables
`ft8-11`	f28-31	Temporary variables

Floating-Point Instructions

- Append .s (single), .d (double), .q (quad) for precision. I.e., `fadd.s`, `fadd.d`, and `fadd.q`
- **Arithmetic operations**:

  `fadd`, `fsub`, `fdiv`, `fsqrt`, `fmin`, `fmax`, multiply-add (`fmadd`, `fmsub`, `fnmadd`, `fnmsub`)
- **Other instructions:**

  move (`fmv.x.w`, `fmv.w.x`)
  convert (`fcvt.w.s`, `fcvt.s.w`, etc.)
  comparison (`feq`, `flt`, `fle`)
  classify (`fclass`)
  sign injection (`fsgnj`, `fsgnjn`, `fsgnjx`)

Floating-Point Multiply-Add

- `fmadd` is the most critical instruction for signal processing programs.
- Requires four registers.

```
fmadd.f f1, f2, f3, f4     # f1 = f2 x f3 + f4
```

Floating-Point Example

C Code

```
int i;
float scores[200];

for (i=0; i<200; i=i+1)

  scores[i]=scores[i]+10;
```

RISC-V assembly code

```
# s0 = scores base address, s1 = i
  addi s1, zero, 0          # i = 0
  addi t2, zero, 200        # t2 = 200
  addi t0, zero, 10         # ft0 = 10.0
  fcvt.s.w ft0,  t0
for:
  bge    s1, t2, done       # i>=200? done
  slli   t0, s1, 2          # t0 = i*4
  add    t0, t0, s0         # scores[i] address
  flw    ft1, 0(t0)         # ft1=scores[i]
  fadd.s ft1, ft1, ft0      # ft1=scores[i]+10
  fsw    ft1, 0(t0)         # scores[i] = t1
  addi   s1, s1, 1          # i = i+1
  j      for                # repeat
done:
```

Floating-Point Instruction Formats

- Use R-, I-, and S-type formats
- Introduce another format for multiply-add instructions that have 4 register operands: R4-type

R4-Type

31:27	26:25	24:20	19:15	14:12	11:7	6:0
rs3	funct2	rs2	rs1	funct3	rd	op
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Exceptions

Exceptions

- Unscheduled function call to *exception handler*
- Caused by:
  - Hardware, also called an *interrupt*, e.g., keyboard
  - Software, also called *traps*, e.g., undefined instruction
- When exception occurs, the processor:
  - Records the cause of the exception
  - Jumps to exception handler
  - Returns to the program

Exception Causes

Exception	Cause
Instruction address misaligned	0
Instruction access fault	1
Illegal instruction	2
Breakpoint	3
Load address misaligned	4
Load access fault	5
Store address misaligned	6
Store access fault	7
Environment call from U-Mode	8
Environment call from S-Mode	9
Environment call from M-Mode	11

RISC-V Privilege Levels

- In RISC-V, exceptions occur at various **privilege levels**.
- Privilege levels limit access to memory or certain (privileged) instructions.
- RISC-V privilege modes are (from highest to lowest):
  - **Machine** mode (bare metal)
  - **System** mode (operating system)
  - **User** mode (user program)
  - **Hypervisor** mode (to support virtual machines)
- For example, a program running in M-mode (machine mode) can access all memory or instructions – it has the highest privilege level.

Exception Registers

- Each privilege level has registers to handle exceptions
- These registers are called control and status registers (**CSRRs**)
- We discuss **M-mode** (machine mode) exceptions, but other modes are similar
- M-mode registers used to handle exceptions are:
  - `mtvec`, `mcause`, `mepc`, `mscratch`

(Likewise, S-mode exception registers are: `stvec`, `scause`, `sepc`, and `mscratch`; and so on for the other modes.)

Exception Registers

- CSRRs are not part of register file
- M-mode CSRRs used to handle exceptions
  - **`mtvec:`** holds address of exception handler code
  - **`mcause`**: Records cause of exception
  - **`mepc`** (Exception PC): Records PC where exception occurred
  - **`mscratch`**: scratch space in memory for exception handlers

Exception-Related Instructions

- Called **privileged instructions** (because they access CSRRs)
  - **`csrr`**: CSR register read
  - **`csrw`**: CSR register write
  - **`csrrw`**: CSR register read/write
  - **`mret`**: returns to address held in mepc
- **Examples:**

```
csrr t1, mcause          # t1 = mcause
csrw mepc, t2            # mepc = t2
cwrrw t0, mscratch, t1 # t0 = mscratch
                         # mscratch = t1
```

Exception Handler Summary

- When a processor **detects an exception**:
  - It **jumps to exception handler** address in `mtvec`
  - The exception handler then:
    - **saves registers** on small stack pointed to by `mscratch`
    - Uses `csrr` (CSR read) to **look at cause** of exception (in `mcause`)
    - **Handles exception**
    - When finished, optionally **increments `mepc` by 4** and **restores registers** from memory
    - And then either **aborts** the program or **returns to user code** (using `mret`, which returns to address held in `mepc`)

Example Exception Handler Code

- Check for **two types of exceptions**:
  - **Illegal instruction** (`mcause` = 2)
  - **Load address misaligned** (`mcause` = 4)

Example Exception Handler Code

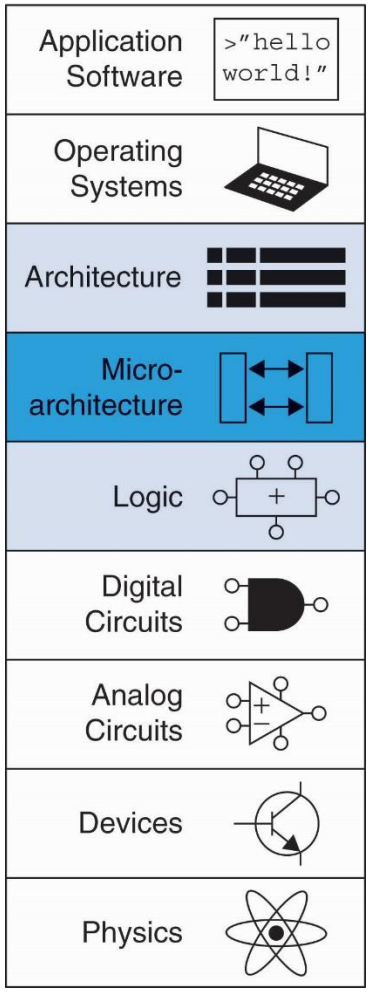
```
# save registers that will be overwritten
  csrrw t0, mscratch, t0     # swap t0 and mscratch
  sw    t1, 0(t0)            # [mscratch]   = t1
  sw    t2, 4(t0)            # [mscratch+4] = t2
# check cause of exception
  csrr  t1, mcause           # t1=mcause
  addi  t2, x0, 2            # t2=2 (illegal instruction exception code)
illegalinstr:
  bne   t1, t2, checkother   # branch if not an illegal instruction
  csrr  t2, mepc             # t2=exception PC
  addi  t2, t2, 4            # increment exception PC
  csrw  mepc, t2             # mepc=t2
  j     done                 # restore registers and return
checkother:
  addi  t2, x0, 4            # t2=4 (load address misaligned exception code)
  bne   t1, t2, done         # branch if not a misaligned load
  j     exit                 # exit program
# restore registers and return from the exception
done:
  lw    t1, 0(t0)            # t1 = [mscratch]
  lw    t2, 4(t0)            # t2 = [mscratch+4]
  csrrw t0, mscratch, t0     # swap t0 and mscratch
  mret                       # return to program
exit:
  ...
```

Checks for two types of exceptions:

- **Illegal instruction** (`mcause` = 2)
- **Load address misaligned** (`mcause` = 4)

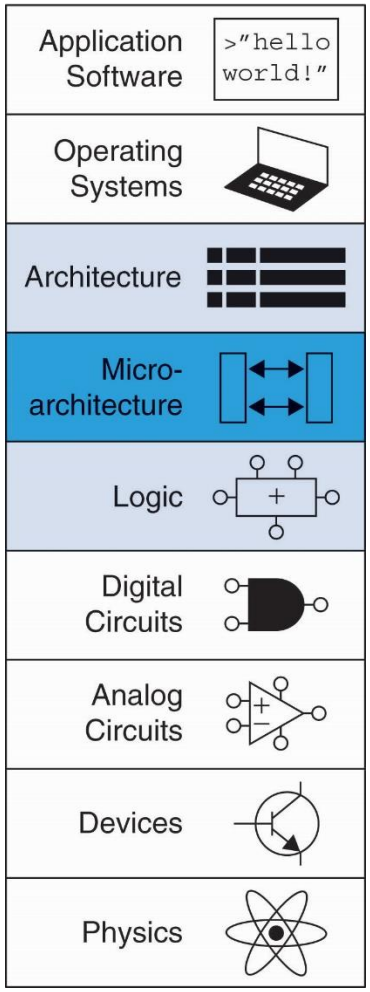
RISC-V Microarchitecture

- **Introduction**
- **Performance Analysis**
- **Single-Cycle Processor**



Introduction

- **Microarchitecture:** how to implement an architecture in hardware
- Processor:
  - **Datapath:** functional blocks
  - **Control:** control signals



Microarchitecture

- **Multiple implementations** for a single architecture:
  - **Single-cycle:** Each instruction executes in a single cycle
  - **Multicycle:** Each instruction is broken up into series of shorter steps
  - **Pipelined:** Each instruction broken up into series of steps & multiple instructions execute at once

Processor Performance

- **Program execution time**

Execution Time = (#instructions)(cycles/instruction)(seconds/cycle)

- **Definitions:**
  - CPI: Cycles/instruction
  - clock period: seconds/cycle
  - IPC: instructions/cycle = IPC
- **Challenge is to satisfy constraints of:**
  - Cost
  - Power
  - Performance

RISC-V Processor

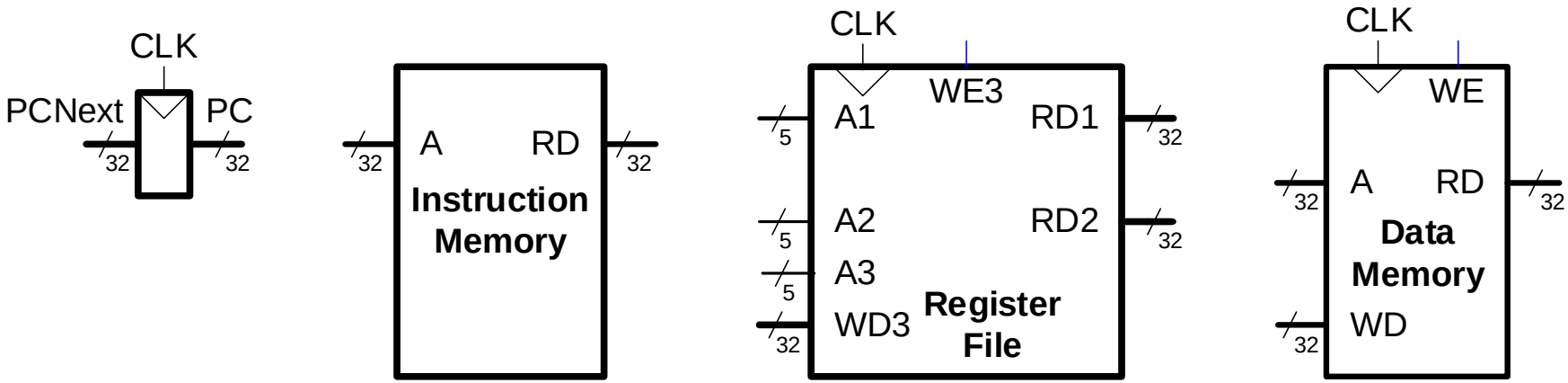
- Consider **subset** of RISC-V instructions:
  - **R-type ALU instructions:**
    - `add`, `sub`, `and`, `or`, `slt`
  - **Memory instructions:**
    - `lw`, `sw`
  - **Branch instructions:**
    - `beq`

Architectural State Elements

Determines everything about a processor:

- **Architectural state:**
  - 32 registers
  - PC
  - Memory

RISC-V Architectural State Elements



Single-Cycle RISC-V Processor

Single-Cycle RISC-V Processor

- Datapath
- Control

Example Program

- Design datapath
- View example program executing

Example Program:

Address	Instruction	Type	Fields						Machine Language
			$imm_{11:0}$		rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100		01001	010	00110	0000011	FFC4A303
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Single-Cycle RISC-V Processor

- **Datapath:** start with `lw` instruction
- **Example:** `lw x6, -4(x9)`

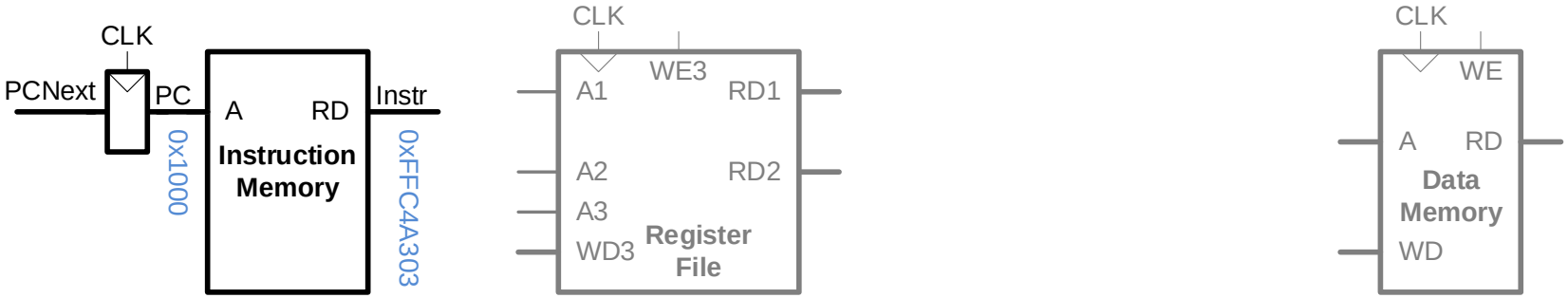
  `lw rd, imm(rs1)`

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

Single-Cycle Datapath: `lw` fetch

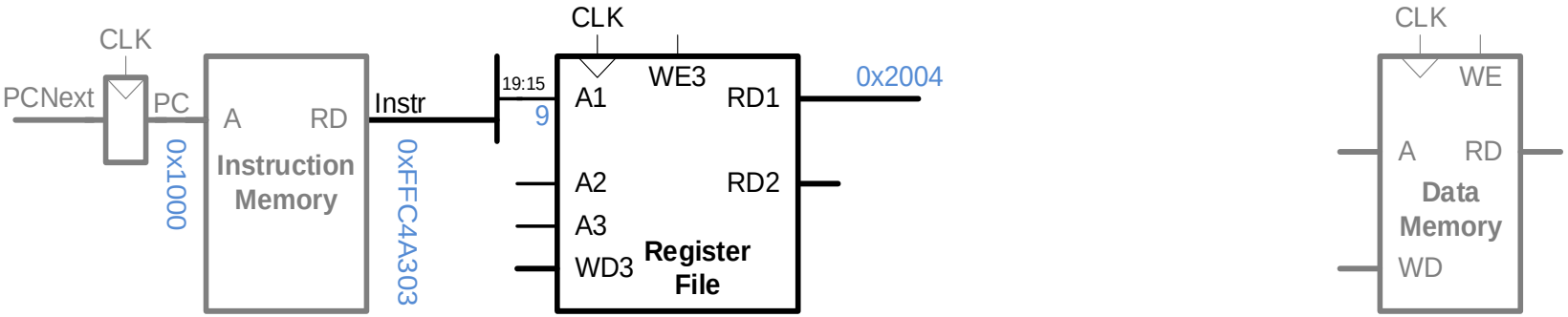
STEP 1: Fetch instruction



Address	Instruction	Type	Fields				Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Reg Read

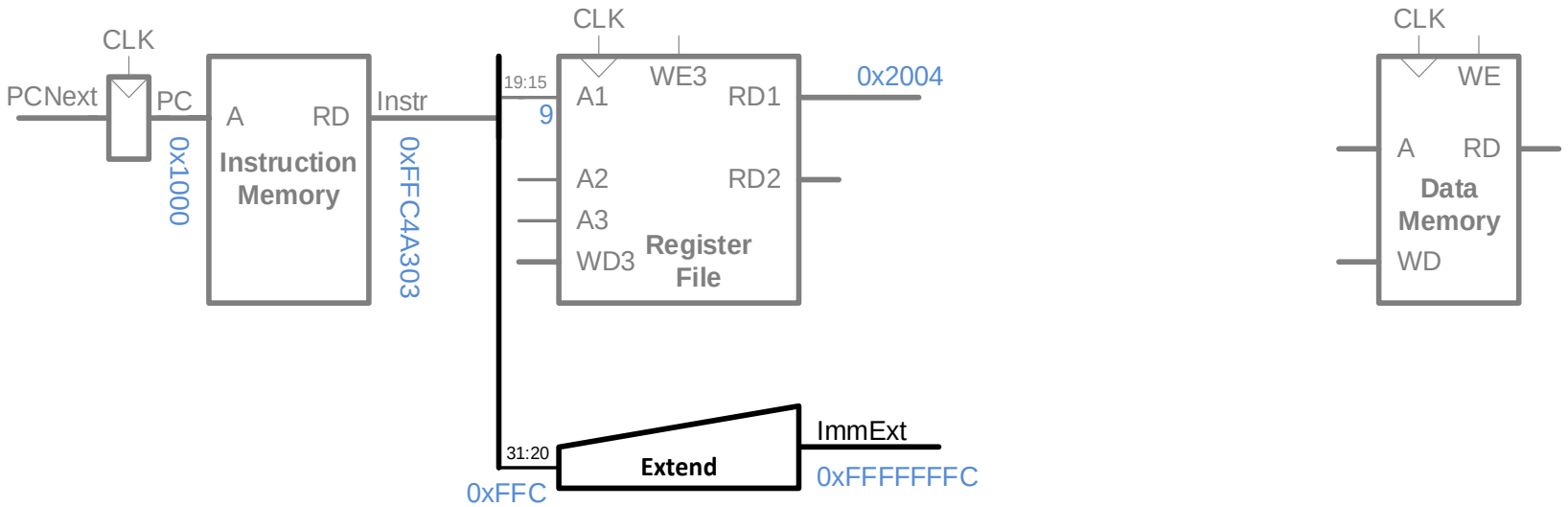
STEP 2: Read source operand (**rs1**) from RF



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Immediate

STEP 3: Extend the immediate

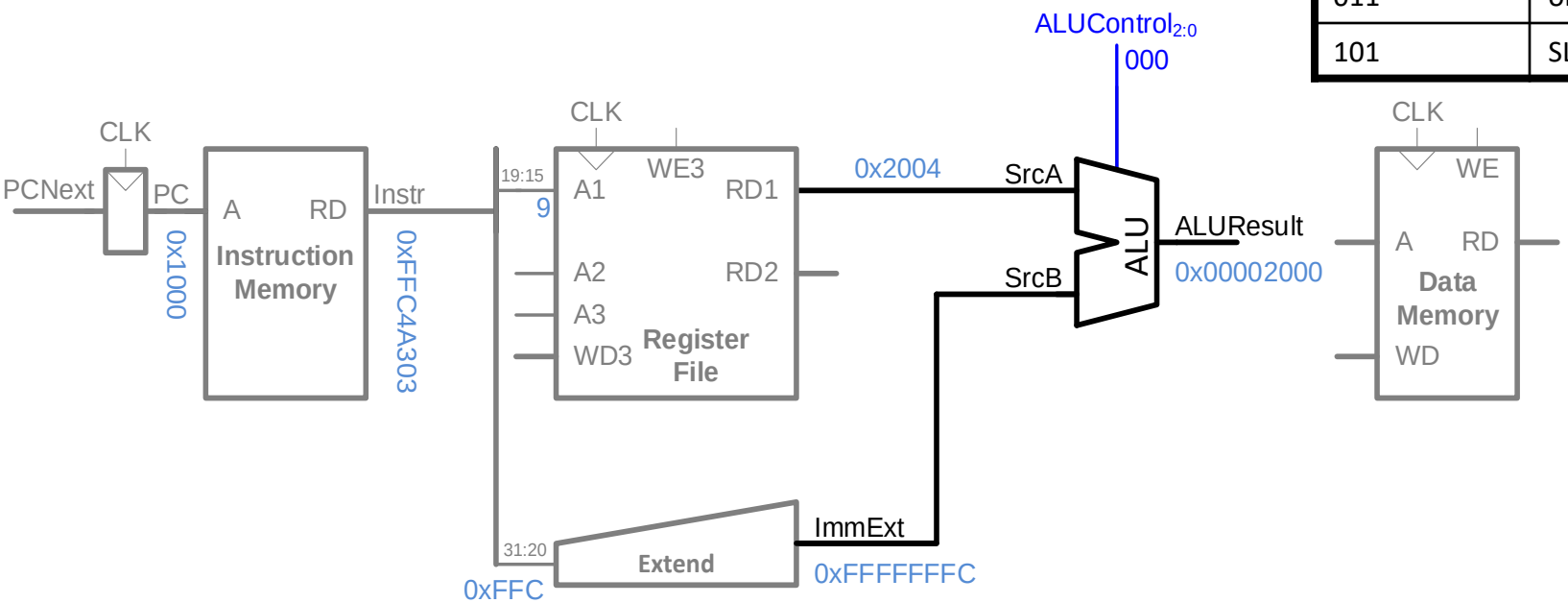


Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Address

STEP 4: Compute the memory address

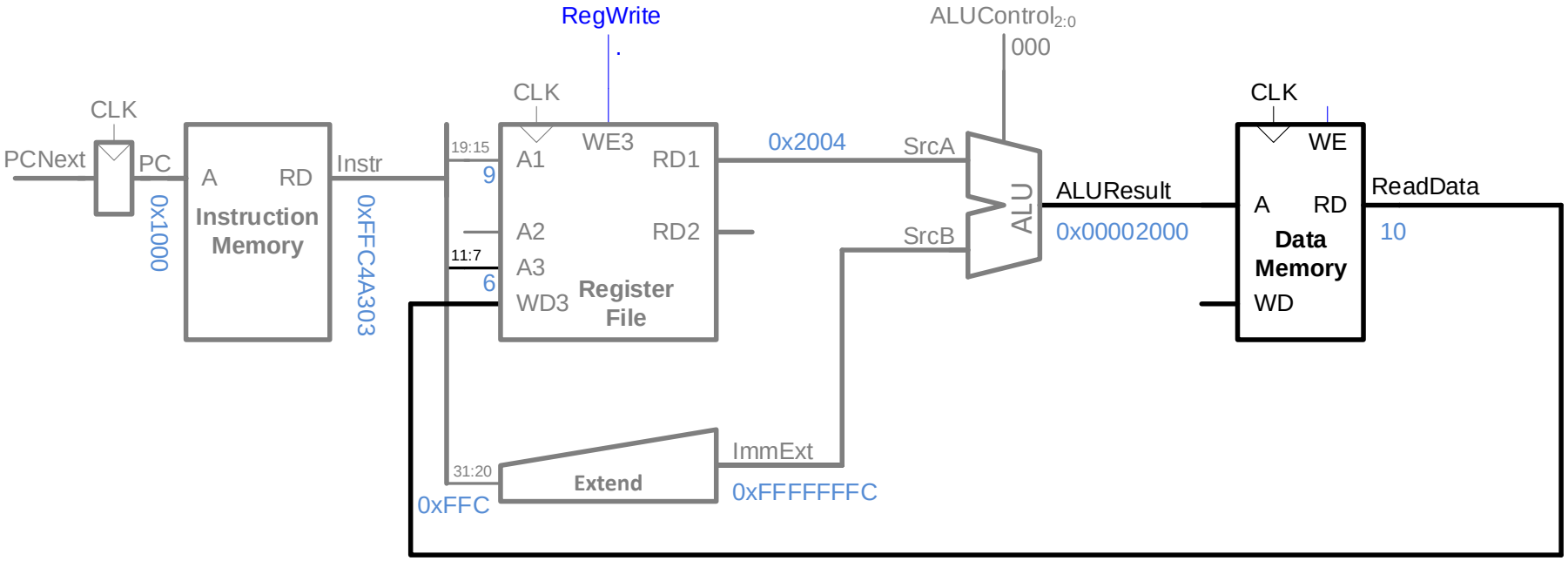
$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: `lw` Mem Read

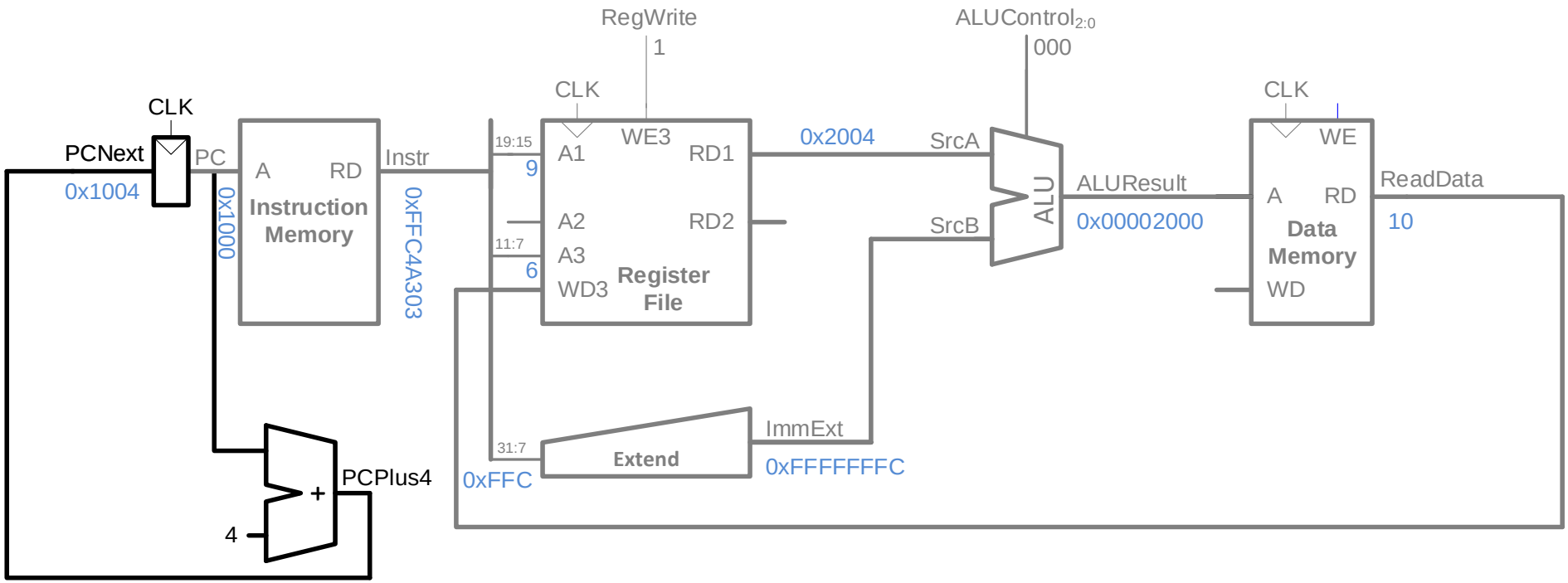
STEP 5: Read data from memory and write it back to register file



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: PC Increment

STEP 6: Determine address of next instruction

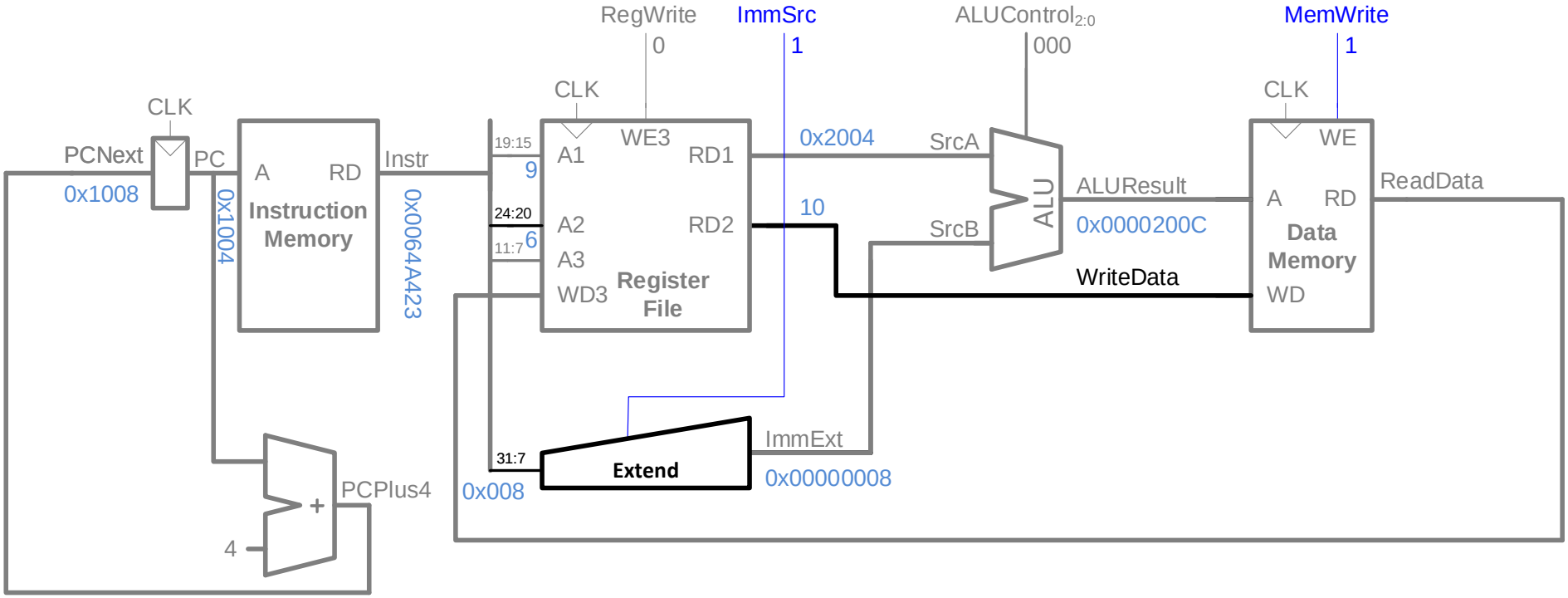


Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Single-Cycle Datapath: Other Instructions

Single-Cycle Datapath: `sw`

- **Immediate:** now in {instr[31:25], instr[11:7]}
- **Add control signals:** ImmSrc, MemWrite



Address	Instruction	Type	Fields						Machine Language
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

Single-Cycle Datapath: Immediate

ImmSrc	ImmExt	Instruction Type
0	{{20{instr[31]}}, **instr[31:20]**}	I-Type
1	{{20{instr[31]}}, **instr[31:25], instr[11:7]**}	S-Type

I-Type

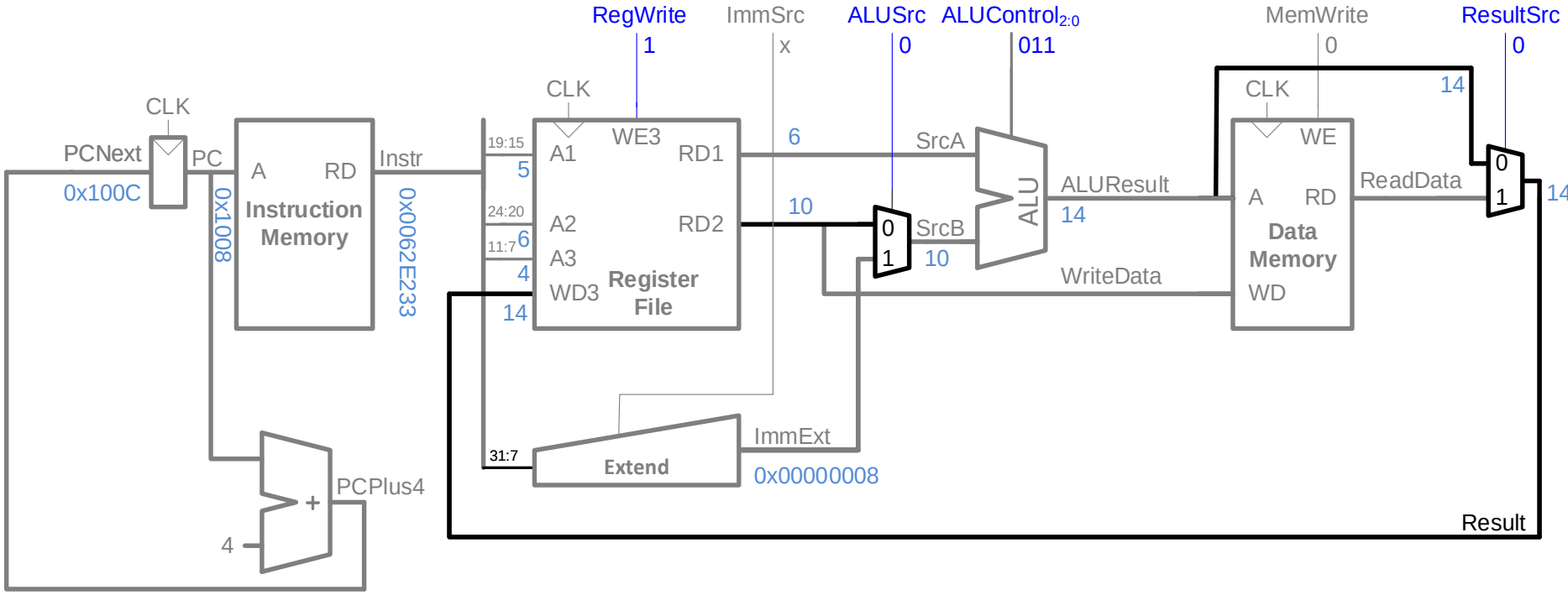
31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

S-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Single-Cycle Datapath: R-type

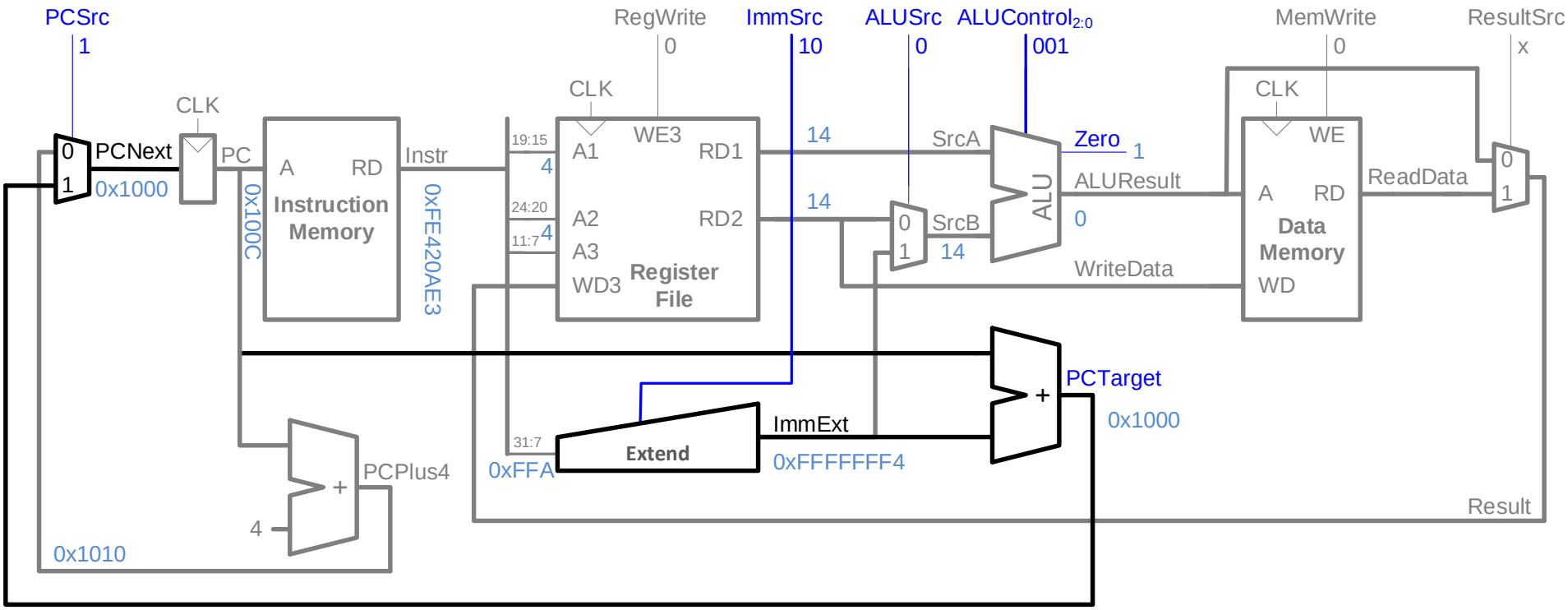
- Read from **rs1** and **rs2** (instead of **imm**)
- Write *ALUResult* to **rd**



Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233

Single-Cycle Datapath: beq

Calculate **target address:** PCTarget = PC + imm



Address	Instruction	Type	Fields						Machine Language
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Single-Cycle Datapath: ImmExt

$ImmSrc_{1:0}$	ImmExt	Instruction Type
00	{{20{instr[31]}}, **instr[31:20]**}	I-Type
01	{{20{instr[31]}}, **instr[31:25], instr[11:7]**}	S-Type
10	{{19{instr[31]}}, **instr[31], instr[7], instr[30:25], instr[11:8], 1'b0**}	B-Type

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

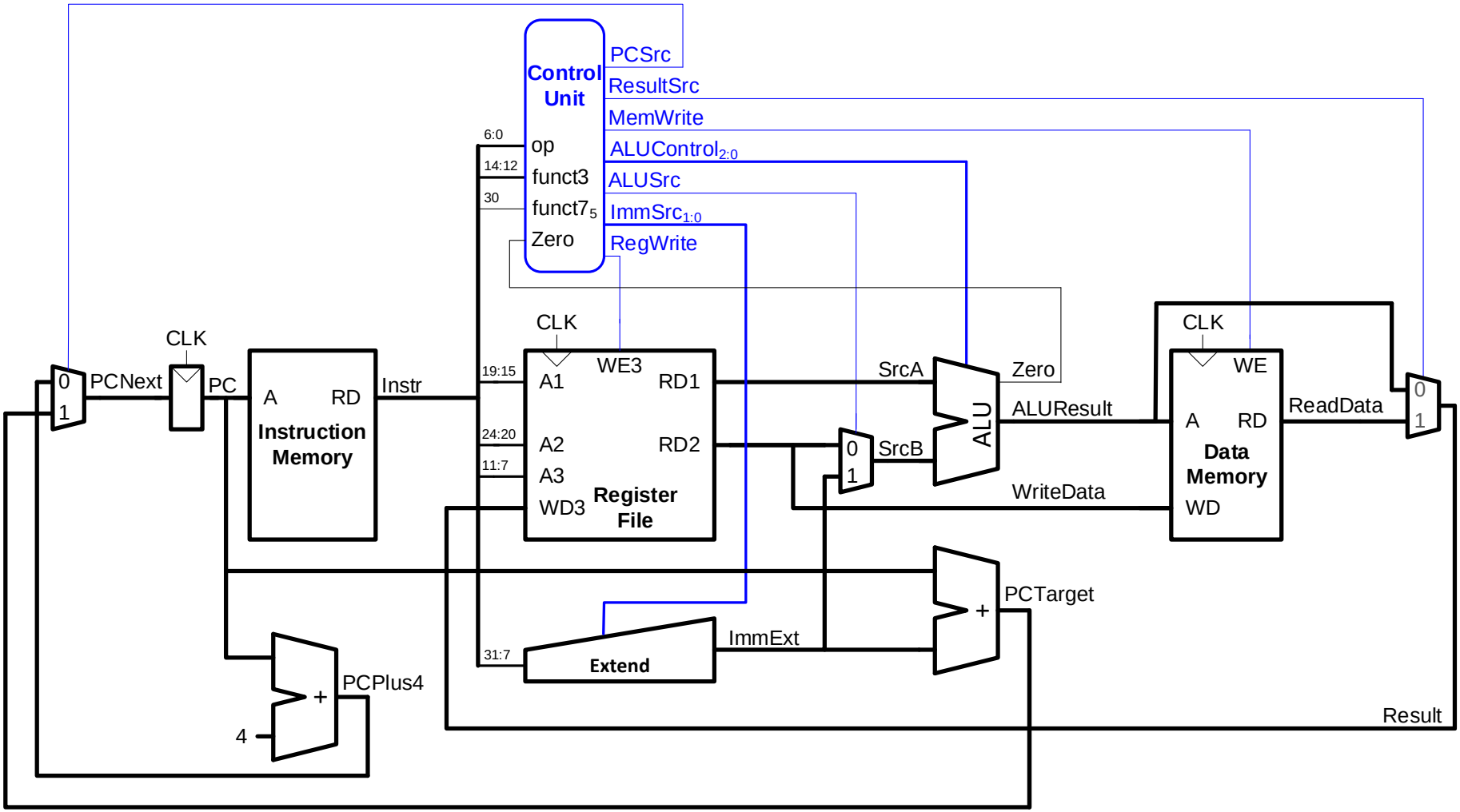
S-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

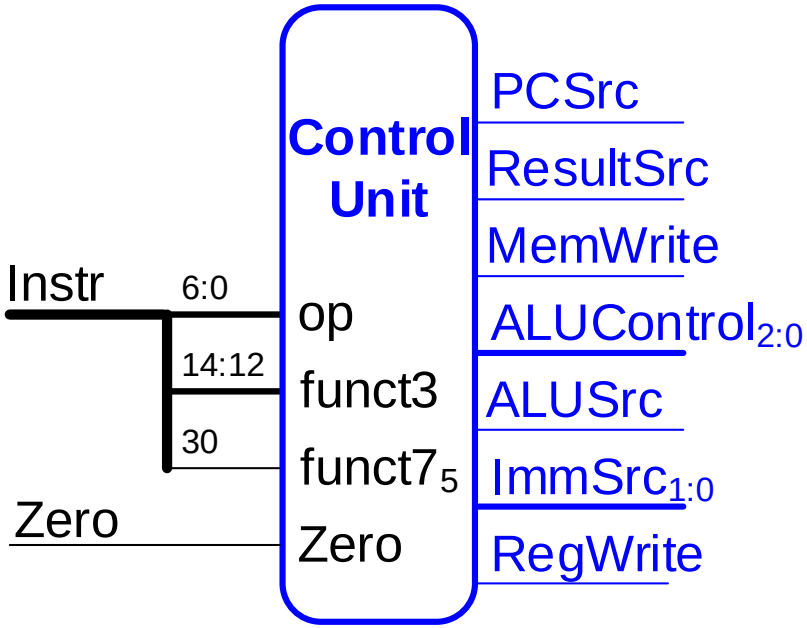
Single-Cycle RISC-V Processor



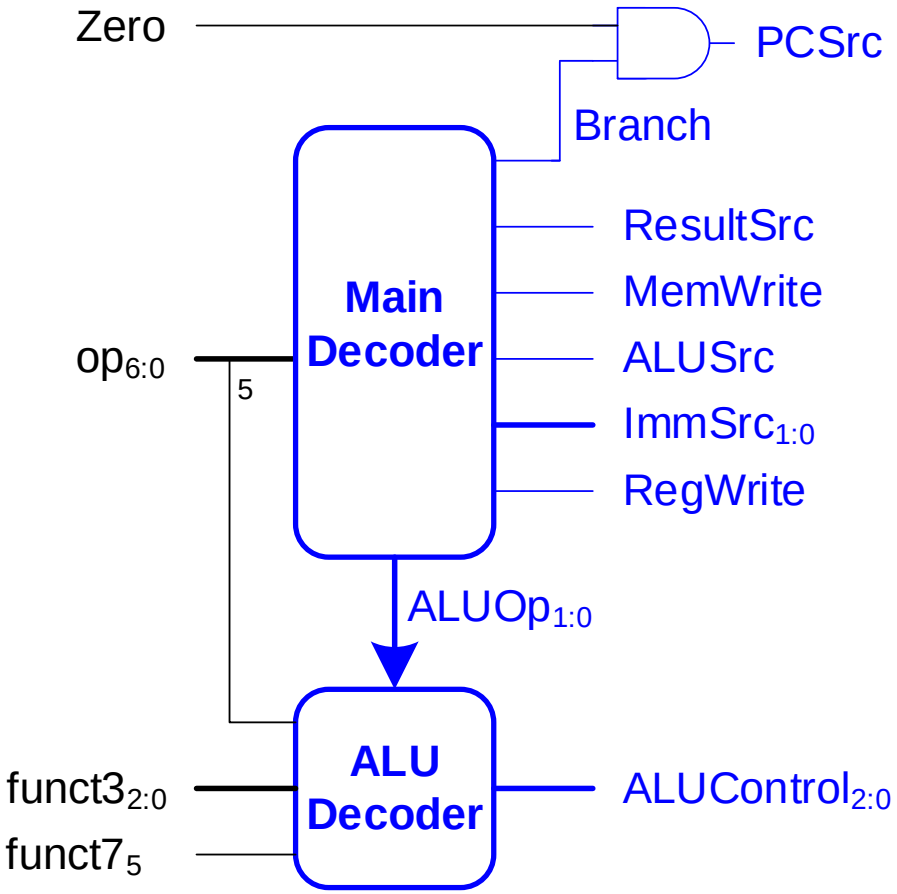
Single-Cycle Control

Single-Cycle Control

High-Level View

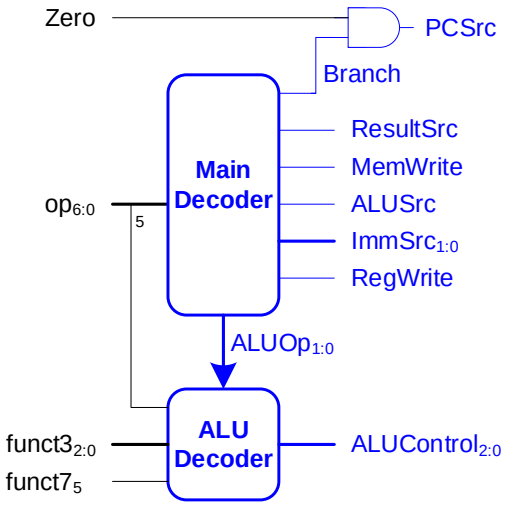


Low-Level View



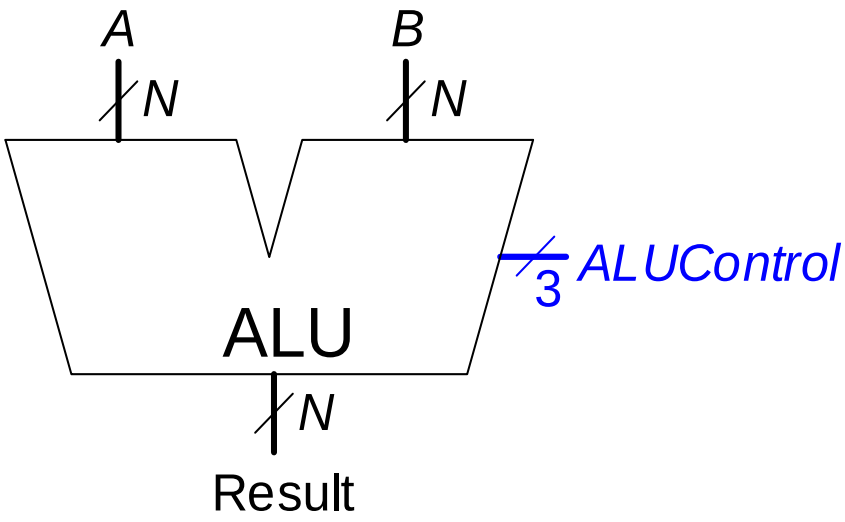
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	`lw`							
35	`sw`							
51	**R-type**							
99	`beq`							



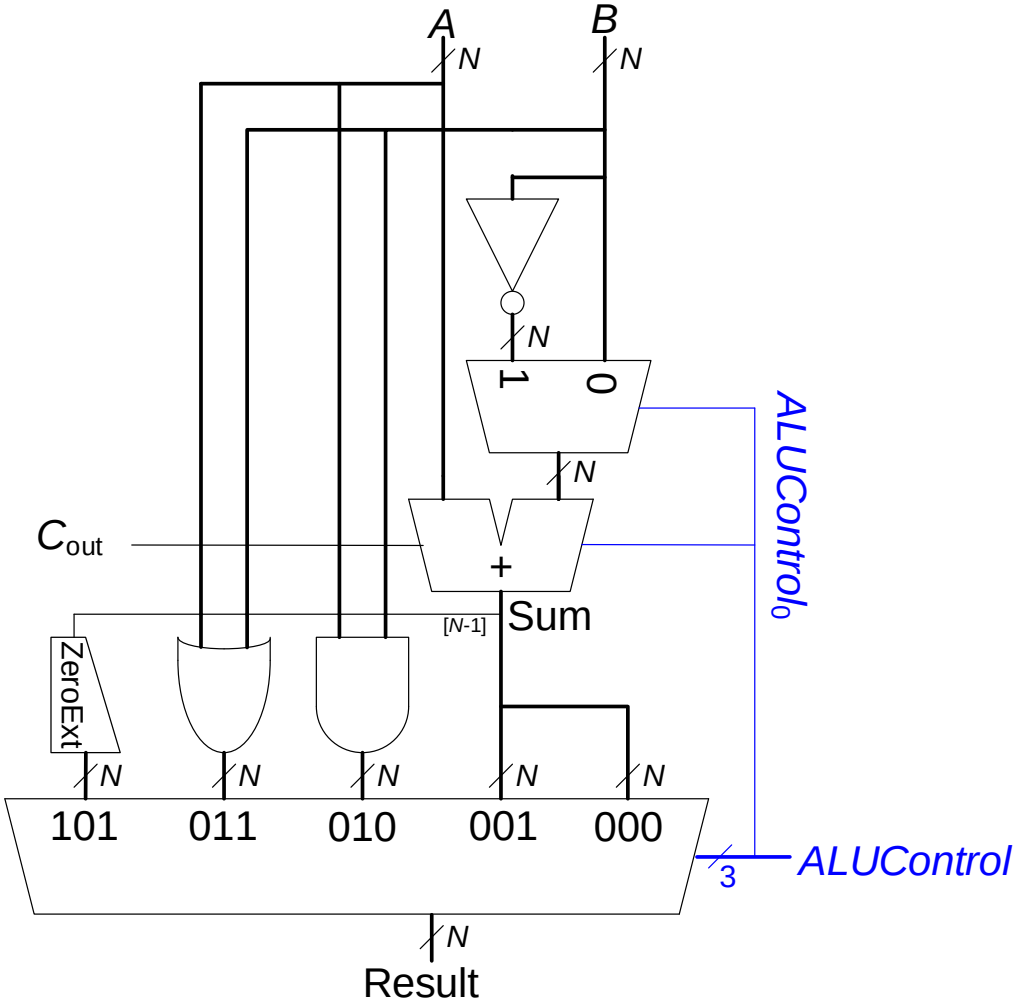
Review: ALU

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT

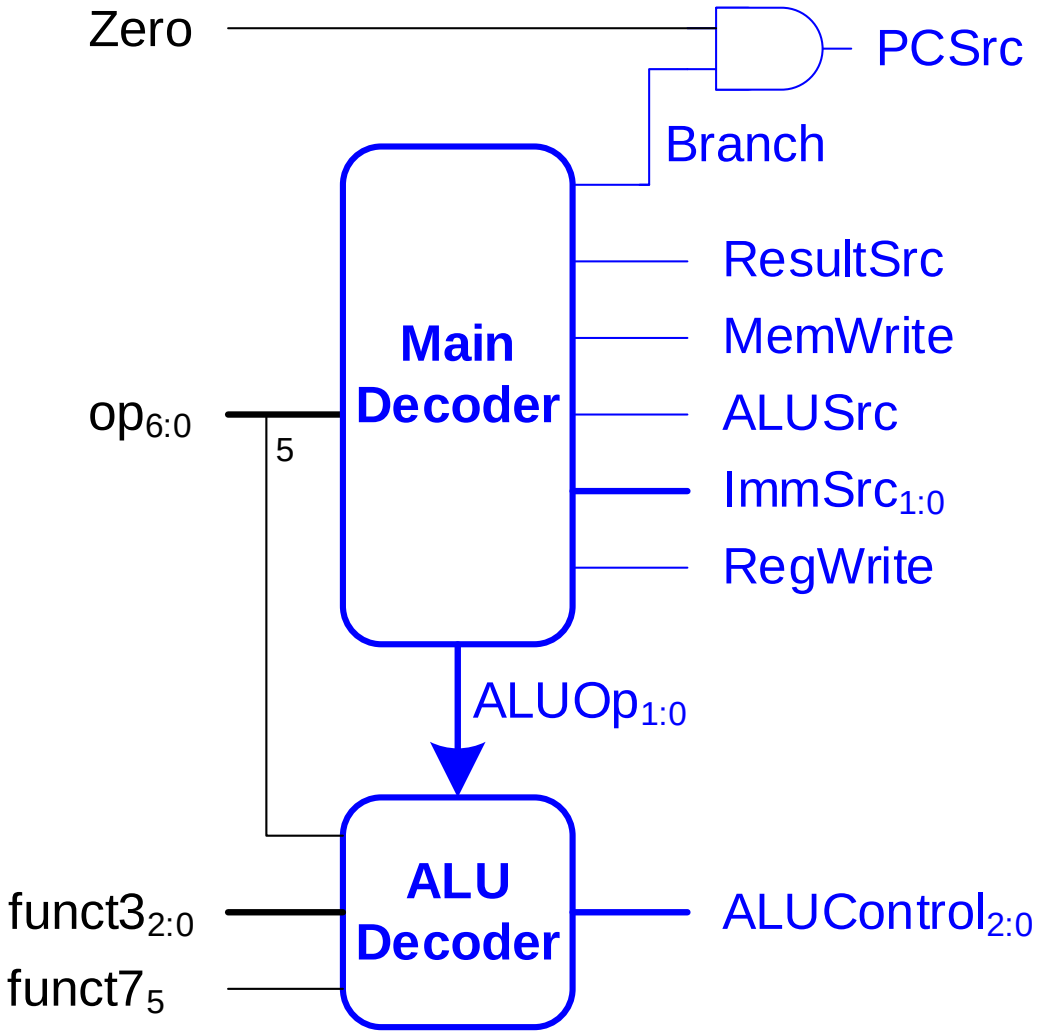


Review: ALU

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT

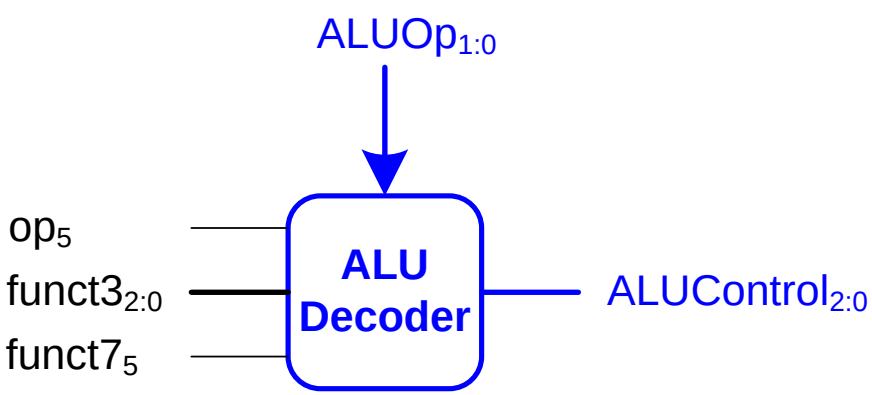


Single-Cycle Control: ALU Decoder



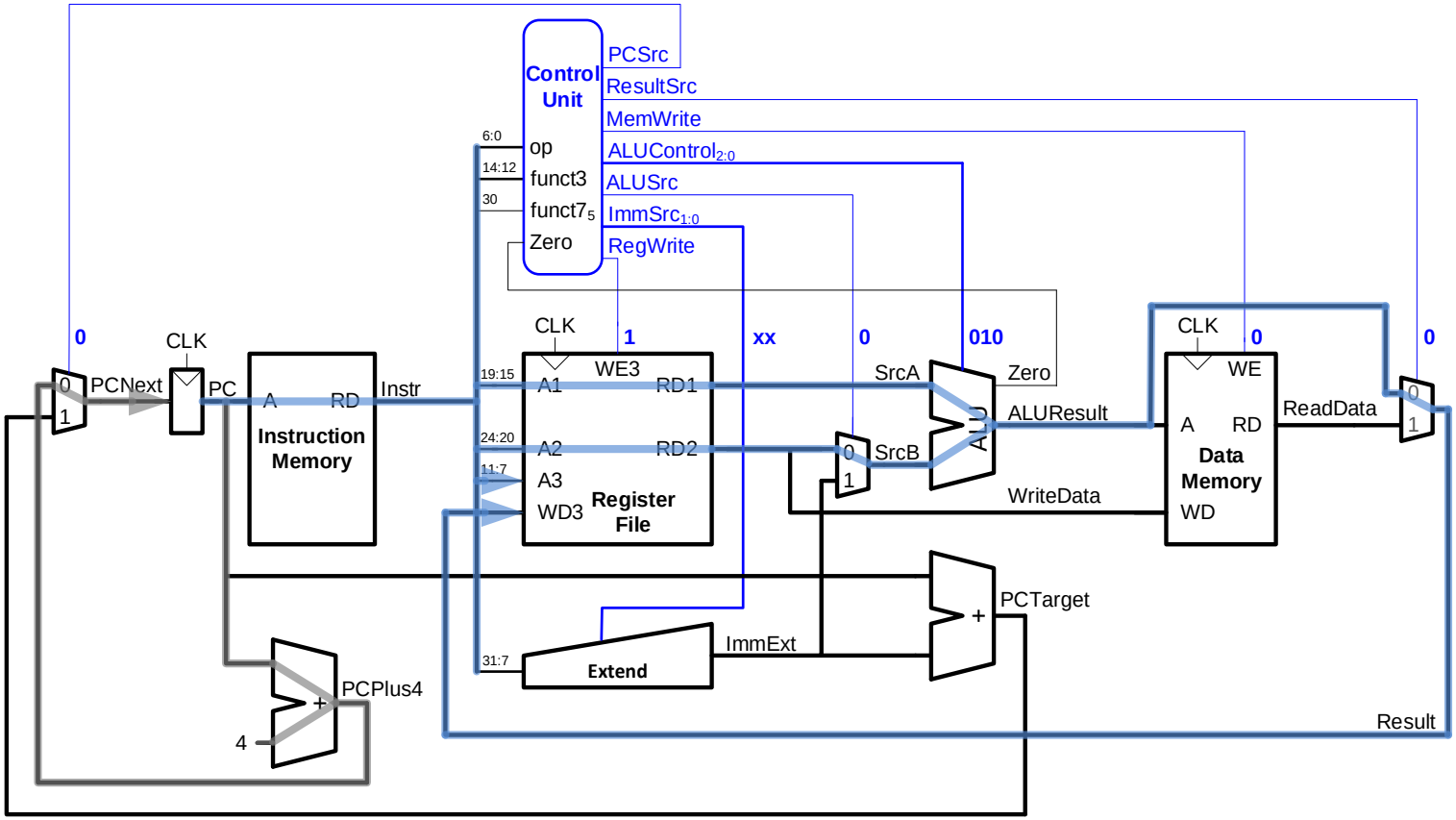
Single-Cycle Control: ALU Decoder

ALUOp	funct3	op_5 , $funct7_5$	Instruction	$ALUControl_{2:0}$
00	x	x	`lw, sw`	000 (add)
01	x	x	`beq`	001 (subtract)
10	000	00, 01, 10	`add`	000 (add)
	000	11	`sub`	001 (subtract)
	010	x	`slt`	101 (set less than)
	110	x	`or`	011 (or)
	111	x	`Slt`	010 (and)



Example: and

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
51	**R-type**	1	XX	0	0	0	0	10




```
and x5, x6, x7
```

Extending the Single-Cycle Processor

Extended Functionality: I-Type ALU

Enhance the single-cycle processor to handle **I-Type ALU instructions**: `addi`, `andi`, `ori`, and `slti`

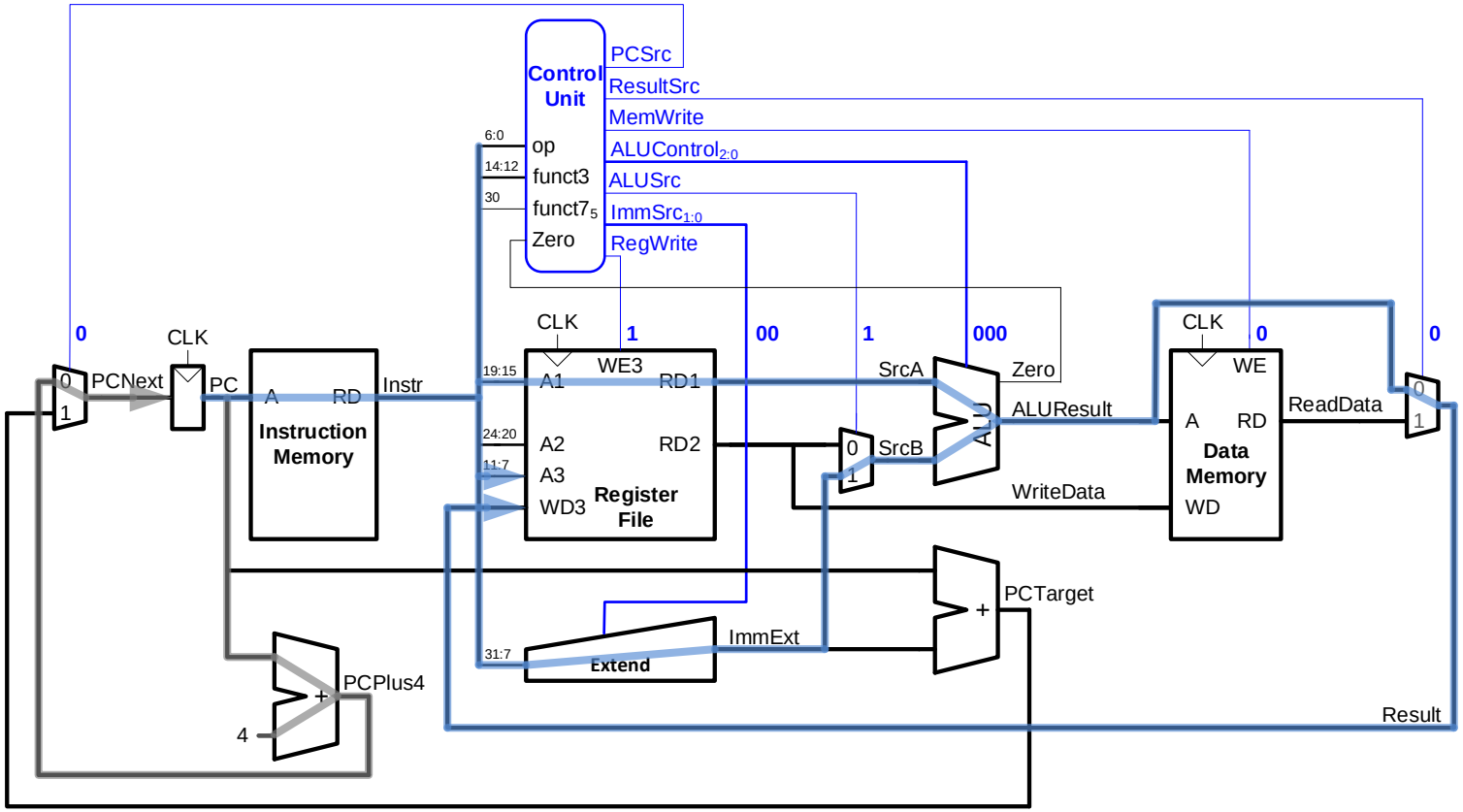
- **Similar to R-type** instructions
- But **second source** comes from **immediate**
- Change ***ALUSrc*** to select the immediate
- And ***ImmSrc*** to pick the correct immediate

Extended Functionality: I-Type ALU

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	`lw`	1	00	1	0	1	0	00
35	`sw`	0	01	1	1	X	0	00
51	**R-type**	**1**	**XX**	**0**	**0**	**0**	**0**	**10**
99	`beq`	0	10	0	0	X	1	01
19	**I-type**	**1**	**00**	**1**	**0**	**0**	**0**	**10**

Extended Functionality: `addi`

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
19	I-type	1	00	1	0	0	0	10



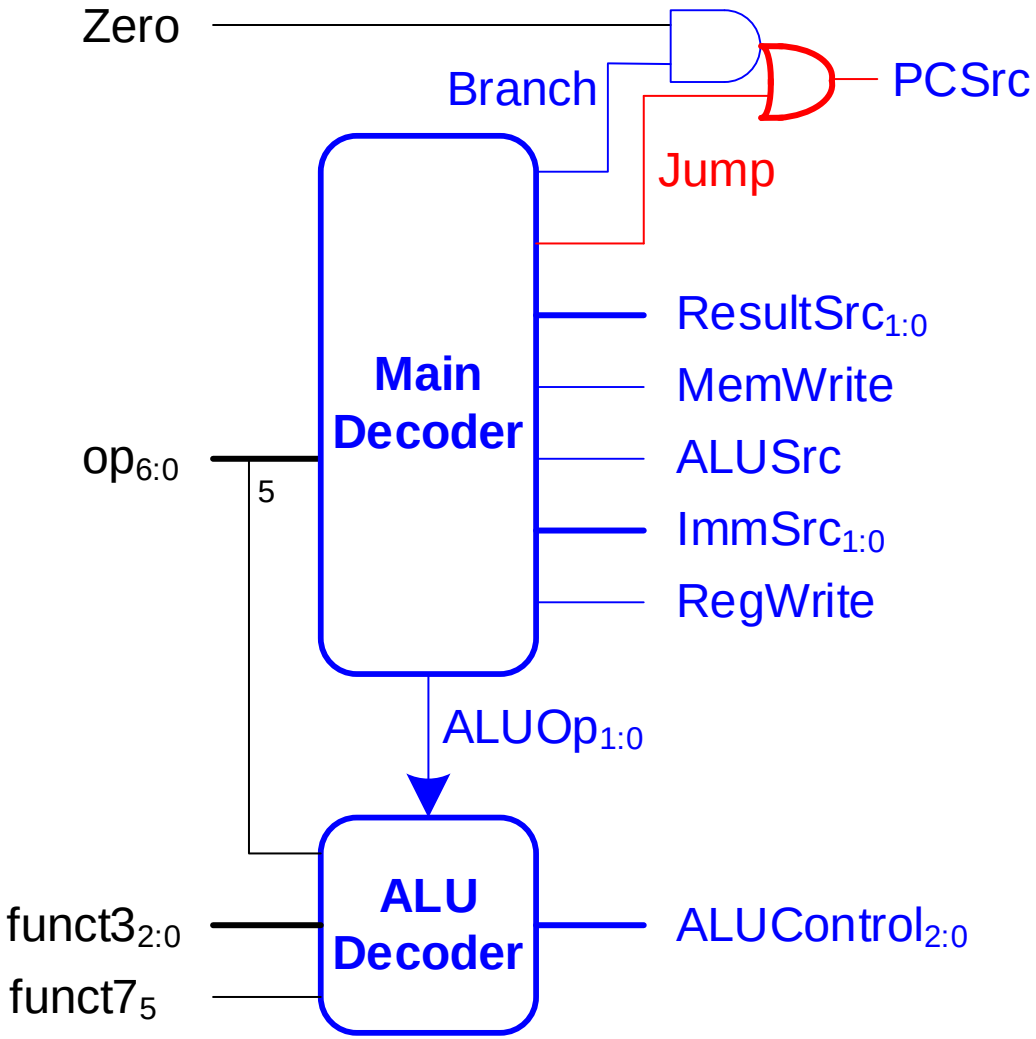

```
addi x5, x6, -33
```

Extended Functionality: `jal`

Enhance the single-cycle processor to handle `jal`

- **Similar to `beq`**
- But jump is **always taken**
  - *PCSrc* should be 1
- **Immediate format** is different
  - Need a new *ImmSrc* of 11
- And `jal` must **compute PC+4** and **store in `rd`**
  - Take PC+4 from adder through ResultMux

Extended Functionality: `jal`



Extended Functionality: *ImmExt*

$ImmSrc_{1:0}$	ImmExt	Instruction Type
00	{{20{instr[31]}}, instr[31:20]}	I-Type
01	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S-Type
10	{{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type
11	**{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}**	**J-Type**

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

S-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

J-Type

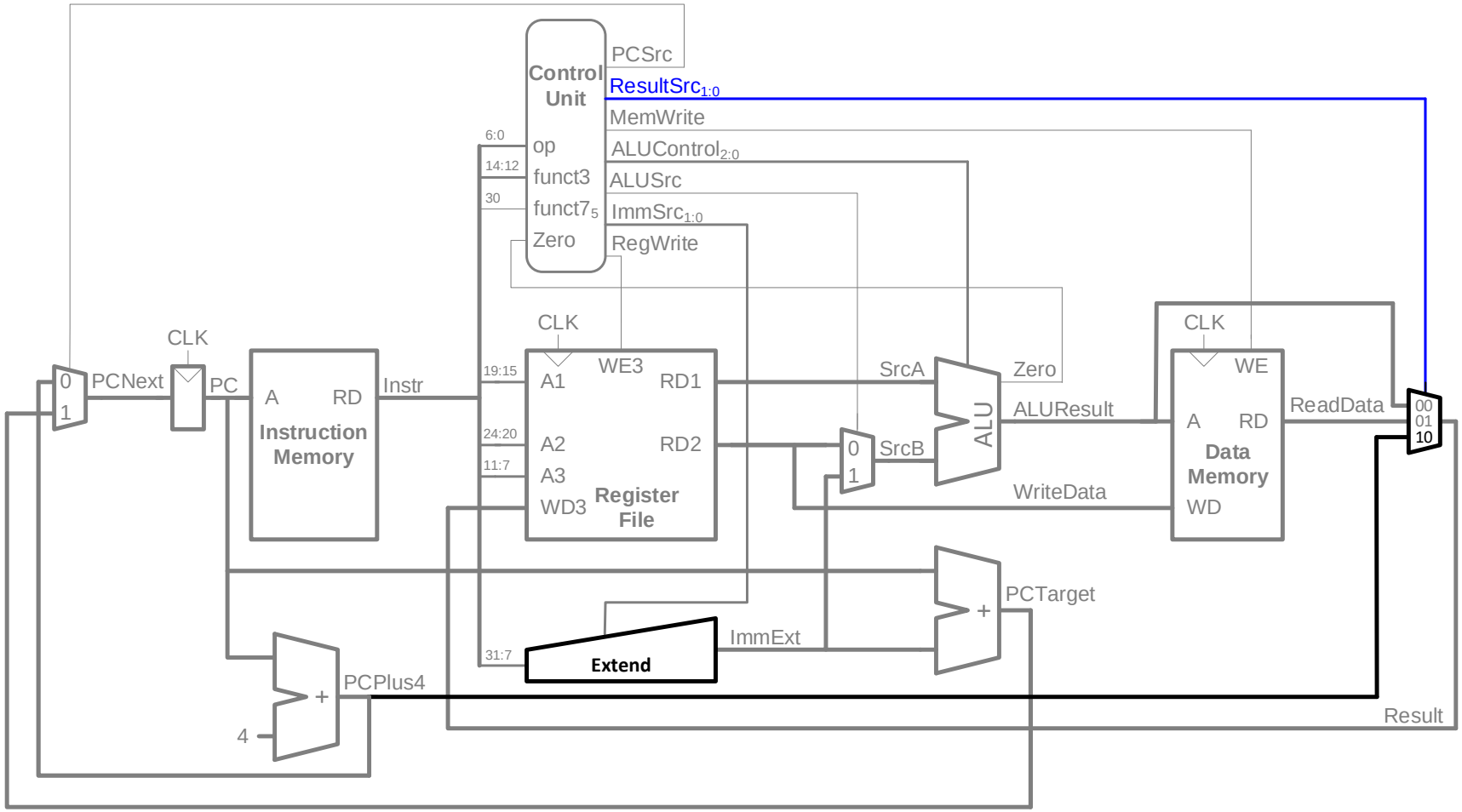
31:12	11:7	6:0
$imm_{20,10:1,11,19:12}$	rd	op
20 bits	5 bits	7 bits

Extended Functionality: `jal`

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
3	**`lw`**	1	00	1	0	10	0	00	**0**
35	**`sw`**	0	01	1	1	XX	0	00	**0**
51	**R-type**	1	XX	0	0	01	0	10	**0**
99	**`beq`**	0	10	0	0	XX	1	01	**0**
19	**I-type**	1	00	1	0	01	0	10	**0**
111	**`jal`**	**1**	**11**	**X**	**0**	**10**	**0**	**XX**	**1**

Extended Functionality: `jal`

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
111	`jal`	1	11	X	0	10	0	XX	1



Single-Cycle Performance

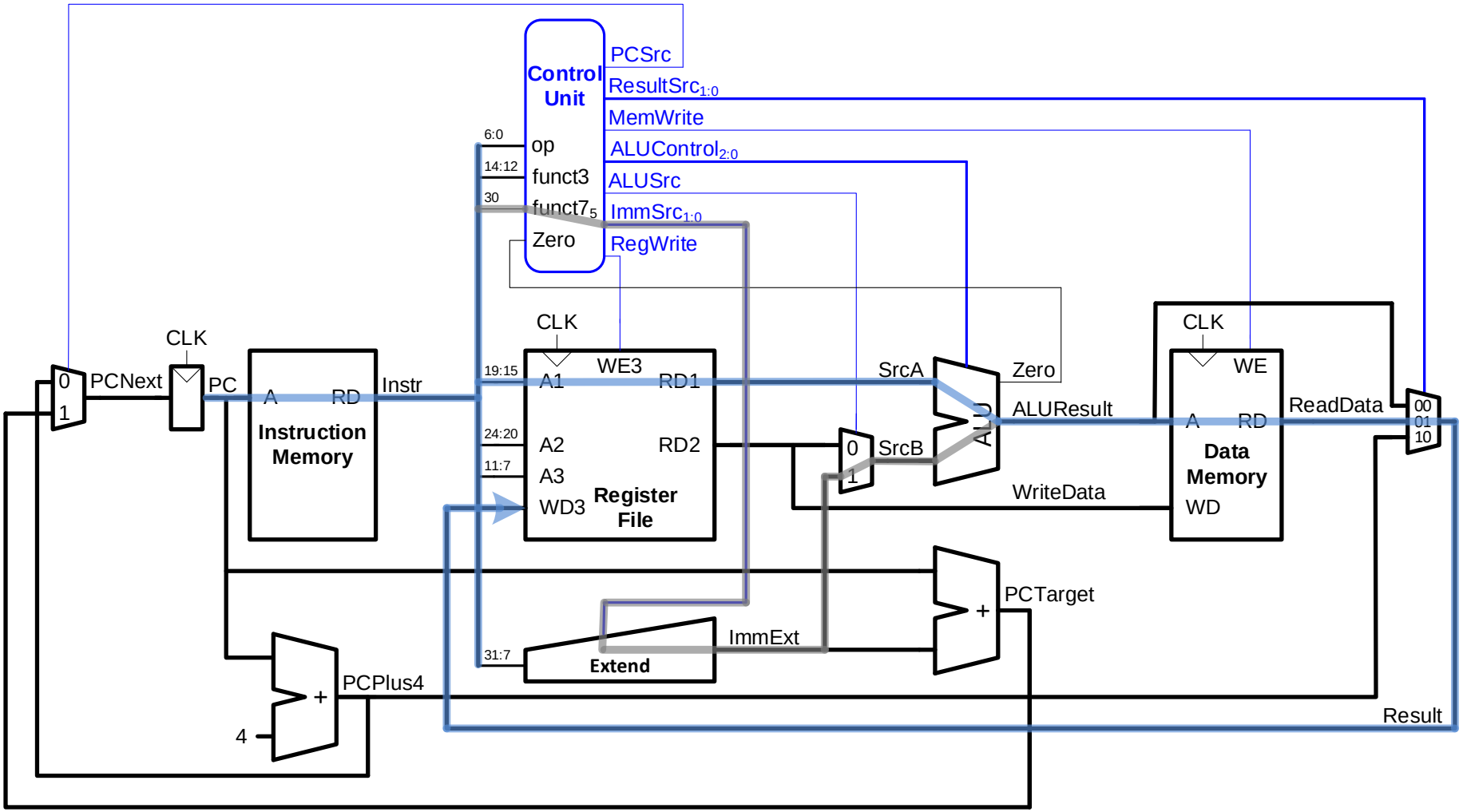
Processor Performance

Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

Single-Cycle Processor Performance



T_C limited by critical path (`lw`)

Single-Cycle Processor Performance

- **Single-cycle critical path**:

$$T_{c\_single} = t_{pcq\_PC} + t_{\text{mem}} + \max[t_{RF\text{read}}, t_{dec} + t_{ext} + t_{\text{mux}}] + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

- **Typically, limiting paths are:**
  - memory, ALU, register file
  - *So,* $T_{c\_single} = t_{pcq\_PC} + t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$

$$= t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}}$$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{\text{AND-OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\text{read}}$	100
Register file setup	$t_{RF\text{setup}}$	60

$$T_{c\_single} = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}}$$
$$=$$

Single-Cycle Performance Example

Program with 100 billion instructions:

Execution Time = # instructions x CPI x T_C