Chapter 4: The Processor

ITSC 3181 Introduction to Computer Architecture
https://passlab.github.io/ITSC3181/

Department of Computer Science
Yonghong Yan
yyan7@uncc.edu
https://passlab.github.io/yanyh/

Chapter 4: The Processor

☛ **Lecture**
- **4.1 Introduction**
- **4.2 Logic Design Conventions**
- **4.3 Building a Datapath**

- **Lecture**
  - **4.4 A Simple Implementation Scheme**
- **Lecture**
  - **4.5 An Overview of Pipelining**
- **Lecture (Pipeline implementation), will not be covered!**
  - 4.6 Pipelined Datapath and Control
  - 4.7 Data Hazards: Forwarding versus Stalling
  - 4.8 Control Hazards
  - ~~4.9 Exceptions~~
  - ~~4.13 Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations~~
- **Lecture (Advanced pipeline techniques and real-world CPU examples)**
  - **4.10 Parallelism via Instructions**
  - **4.11 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines**
  - 4.12 Going Faster: Instruction-Level Parallelism and Matrix Multiply
  - ~~4.14 Fallacies and Pitfalls~~
  - **4.15 Concluding Remarks**

Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- We will examine two CPU implementations
  - A simplified version
  - A more realistic and pipelined version
- **Simple subset, shows the most aspects**
  - **Memory reference: ld/lw, sd/sw**
  - **Arithmetic-logical: add, sub, and, and or**
  - **Condition branch: beq (branch if equal)**

Instruction Set Architecture: The Interface Between Hardware and Software

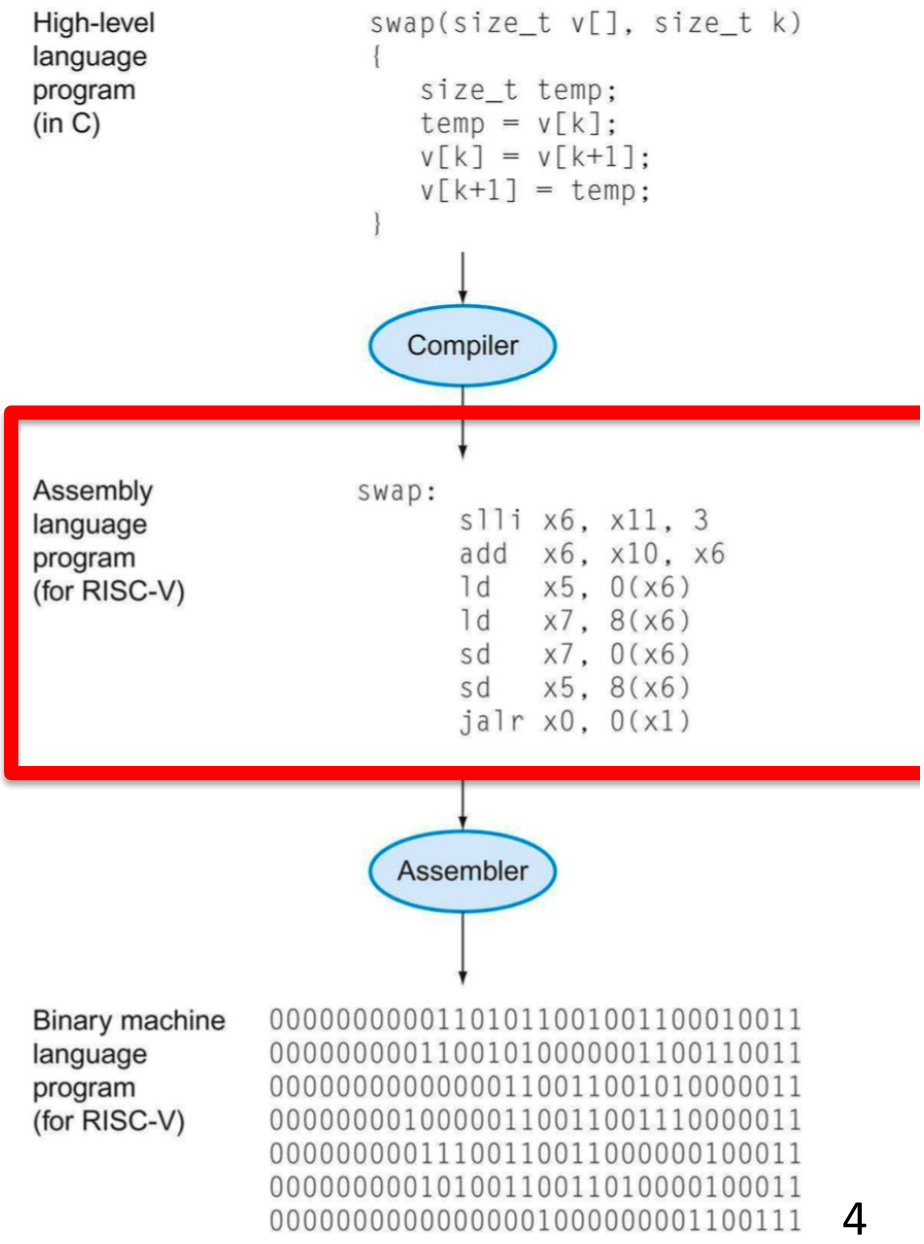
software

hardware

- The words of a computer language are called *instructions*, and its vocabulary/dictionary is called an *instruction set*
  - lowest software interface, assembly level, to the users or to the compiler writer

Instruction Set Architecture – A type of machine

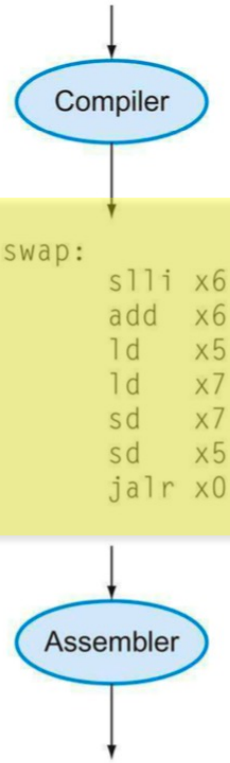
A language represents a race



RISC-V and X86_64 Assembly Example

High-level language program (in C)

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



Assembly language program (for RISC-V)

```
swap:
        slli x6, x11, 3
        add  x6, x10, x6
        ld   x5, 0(x6)
        ld   x7, 8(x6)
        sd   x7, 0(x6)
        sd   x5, 8(x6)
        jalr x0, 0(x1)
```

Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000011001001010000011
00000001000001100110011110000011
00000000111001100110000000100011
00000000101001100110100001000011
00000000000000001000000001100111
```

```
MacBook-Pro-8:exercises yanyh$ gcc -c swap.c
MacBook-Pro-8:exercises yanyh$ objdump -D swap.o

swap.o: file format Mach-O 64-bit x86-64

Disassembly of section __TEXT,__text:
_swap:
       0:       55      pushq   %rbp
       1:       48 89 e5        movq    %rsp, %rbp
       4:       48 89 7d f8     movq    %rdi, -8(%rbp)
       8:       89 75 f4        movl    %esi, -12(%rbp)
       b:       48 8b 7d f8     movq    -8(%rbp), %rdi
       f:       48 63 45 f4     movslq  -12(%rbp), %rax
      13:       8b 34 87        movl    (%rdi,%rax,4), %esi
      16:       89 75 f0        movl    %esi, -16(%rbp)
      19:       48 8b 45 f8     movq    -8(%rbp), %rax
      1d:       8b 75 f4        movl    -12(%rbp), %esi
      20:       83 c6 01        addl    $1, %esi
      23:       48 63 fe        movslq  %esi, %rdi
      26:       8b 34 b8        movl    (%rax,%rdi,4), %esi
      29:       48 8b 45 f8     movq    -8(%rbp), %rax
      2d:       48 63 7d f4     movslq  -12(%rbp), %rdi
      31:       89 34 b8        movl    %esi, (%rax,%rdi,4)
      34:       8b 75 f0        movl    -16(%rbp), %esi
      37:       48 8b 45 f8     movq    -8(%rbp), %rax
      3b:       8b 4d f4        movl    -12(%rbp), %ecx
      3e:       83 c1 01        addl    $1, %ecx
      41:       48 63 f9        movslq  %ecx, %rdi
      44:       89 34 b8        movl    %esi, (%rax,%rdi,4)
      47:       5d      popq    %rbp
      48:       c3      retq
```

Three Classes of Instructions

1. **Arithmetic-logic instructions**
   - add, sub, addi, and, or, shift left|right, etc
2. **Memory load and store instructions**
   - lw and sw: Load/store word
   - ld and sd: Load/store doubleword
3. **Control transfer instructions (changing sequence of instruction execution)**
   - Conditional branch: bne, beq
   - Unconditional jump: j (
   - Procedure call and return: jal and jr

x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

Arithmetic-logic and load/store

- Arithmetic-logic instructions
  - Three operands, could be either register or immediate (for source operands only)
    - add x10, x5, x6; sub x5, x4, x7; and x10, x5, x7
    - addi x10, x5, 10;
- Load and store (L/S) instructions: Load data from memory to register and store data from register to memory
  - Remember the way of specifying memory address (base+offset)
  - ```
    ld  x9, 64(x22)     // load doubleword
    sd  x9, 96(x22)     // store doubleword
    ```
- With these two classes instructions, you can implement the following high-level code, and different ways of combining them
  - `f = (g + h) - (i + j);`
  - `A[12] = h + A[8];`
  - For L/S: **Left-value (of =) to Store**, **Right-value of (=) to Load**
    ```

Load and Store Operations

```
Format:  ld  rd, offset(rs1)
Example: ld  x9, 64(x22)  // load doubleword to x9
```

- ld: load a doubleword from a memory location whose address is specified as rs1+offset (base+offset, x22+64) into register rd (x9)
  - Base should be stored in an register, **offset MUST be a constant number**
  - Address is specified similar to array element, e.g. A[8], for ld, the address is offset(base), e.g. 64(x22)

```
Format:  sd  rs2, offset(rs1)
Example: sd  x9, 96(x22)    // store a doubleword
```

- sd: store a doubleword from register rs2 (x9 in the example) to a memory location whose address is specified as rs1+offset(base+offset, x22+96). **Offset MUST be a constant number.**

- **Load and store are the ONLY two instructions that access memory**
- lw: load a word from memory location to a register
- sw: store a word from a register to a memory location

Memory Operand Example

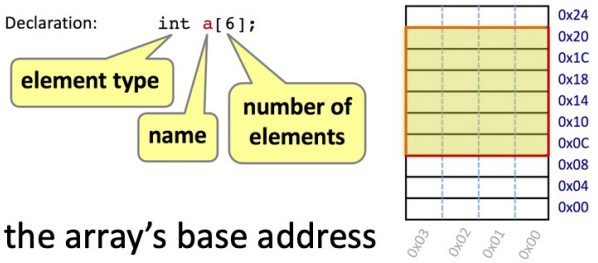
- C code:

```
double A[N]; //double size is 8 bytes
A[12] = h + A[8];
```

  - **h in x21, base address of A in x22**

- Compiled RISC-V code:
  - Index 8 requires offset of 64
  - **A[8] right-val, A[12]: left-val**

- `int a[6];`



- a is the name of the array's base address
  - 0x0C

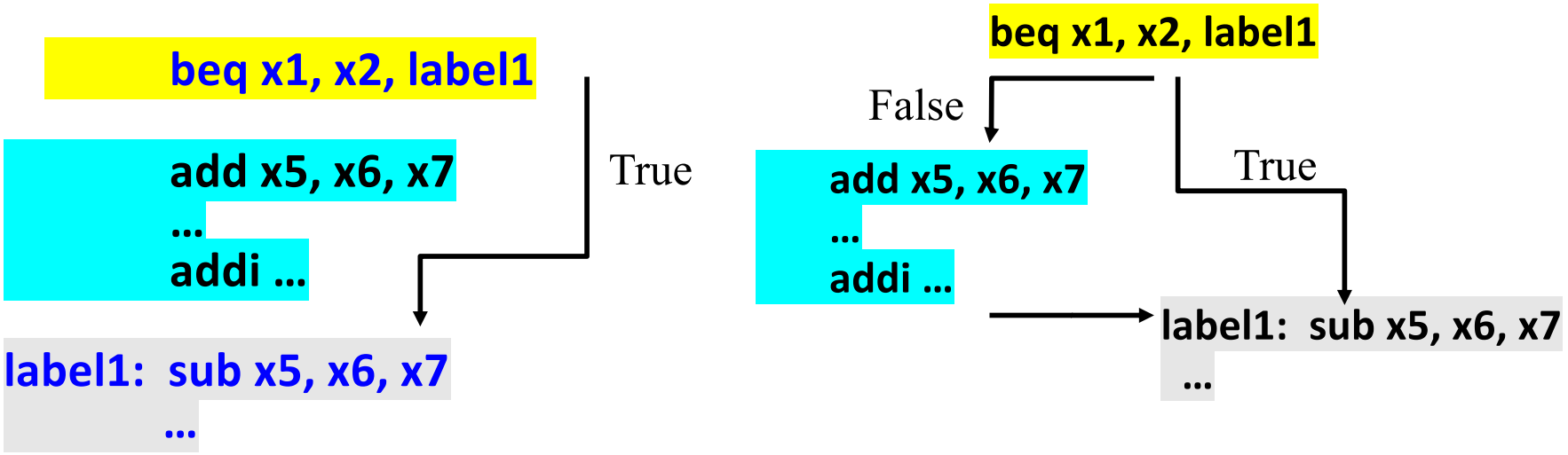
&a[i]: (char*)a + i * sizeof(int)

```
ld  x9, 64(x22)     // load doubleword
add x9, x21, x9
sd  x9, 96(x22)     // store doubleword
```

Conditional Branch

Branch to the labeled instruction if a condition is true, otherwise continue

- `beq rs1, rs2, L1`
  - if (rs1 == rs2, i.e. true) branch to instruction labeled L1 (branch target);
  - else continue the following instruction



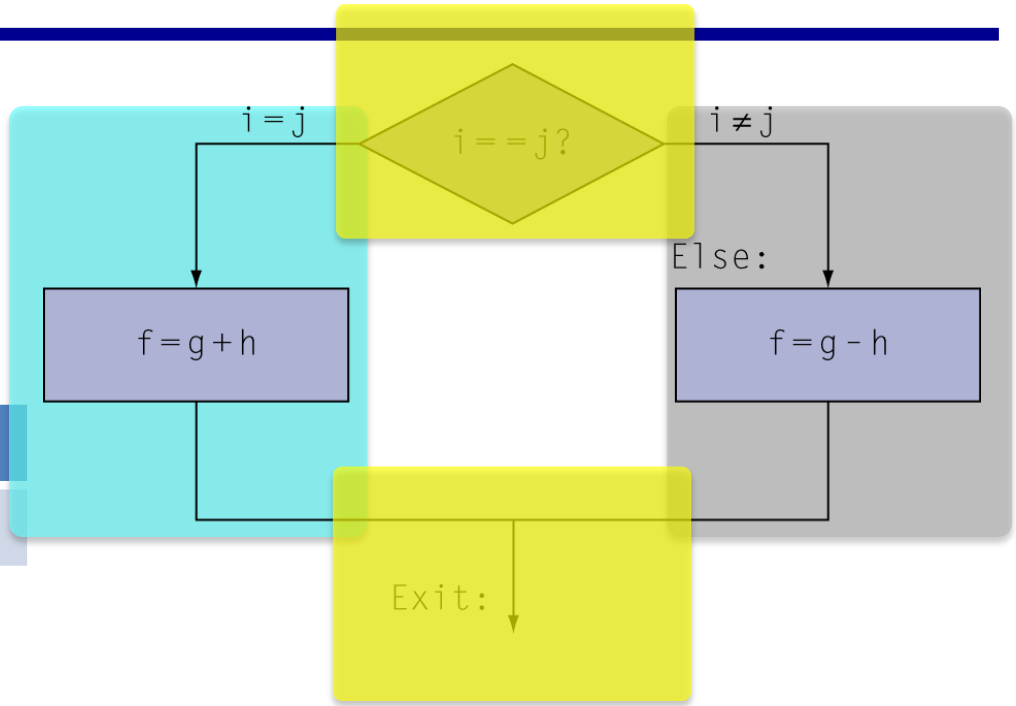
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1 (branch target);
  - else continue the following instruction
- J: unconditional jump (not an instruction)
  - `beq x0, x0, L1`

Translating If Statements 1/2

- C code:

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

Variable	f	g	h	i	j
Register	x19	x20	x21	x22	x23



- Compiled RISC-V code:

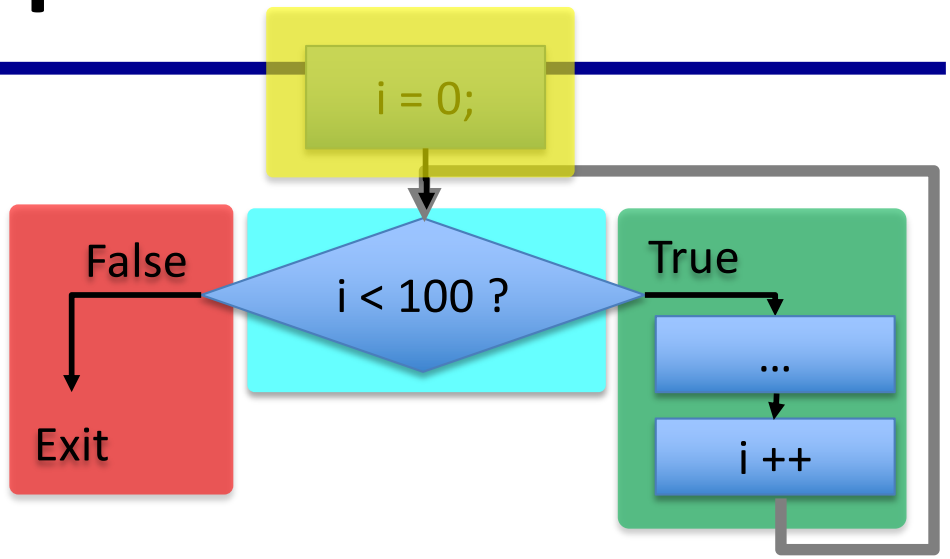
```
      bne x22, x23, Else //branch if not equal
      add x19, x20, x21  //Then path
      beq x0, x0, Exit //unconditional
Else: sub x19, x20, x21 //Else path
Exit: …
```

1. **Using bne (reverse of if (==)) to branch to the Else path b.c. we want the code following the bne to be the code of the Then path**
2. **We need "beq x0 x0 Exit", an unconditional jump, to let Then path terminate since CPU executes instruction in the sequence if not branching.**

Translating Loop Statement

```
for (i=0; i<100; i++) { … }

while (i<100) { …; i++; }
```



- Do the loop structure first
  - Init condition
  - Loop condition (using reverse relationship for branch instr)
  - True path (the loop body)
  - Loop back
  - False path (break the loop)

```
Loop: beq/bge x22, x23, Exit
      …  # loop body

      addi, x22, x22, 1
      beq x0, x0, loop
Exit:
```

- Then translate the loop body

1. **Using bge for (<) to branch to the false/exit path, which breaks the loop**
2. **The instruction(s) following bge are for the true path, which are for the loop body.**
3. **beq to jumping back to the beginning of the loop**

Translating Loop Statement: for loop

- C code:

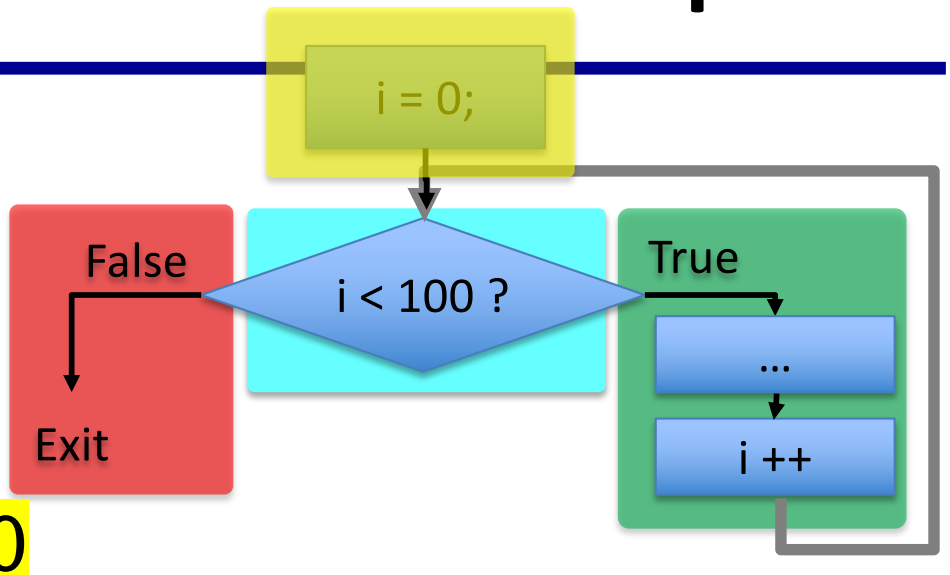
  for (i=0; i<100; i++) …

  – i in x22

- RISC-V code:

```
        addi x22, x0, 0
        li x23, 100

Loop:   bge x22, x23, Exit //beq works
        … …
        addi x22, x22, 1 //true,the loop body,i++
        beq  x0,  x0,  Loop
Exit: …
```



1. **Using bge for (<) to branch to the false/exit path, which breaks the loop**
2. **The instruction(s) following bge are for the true path, which are for the loop body.**
3. **beq to jumping back to the beginning of the loop**

Instruction and Data (1/2)

- **Are all numbers stored as binary format in memory**
  - **It is up to the CPU on how to interpret and do with them**

2s-Complement Signed Integers

Bit 31 is sign bit
- 1 for negative numbers
- 0 for non-negative numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$

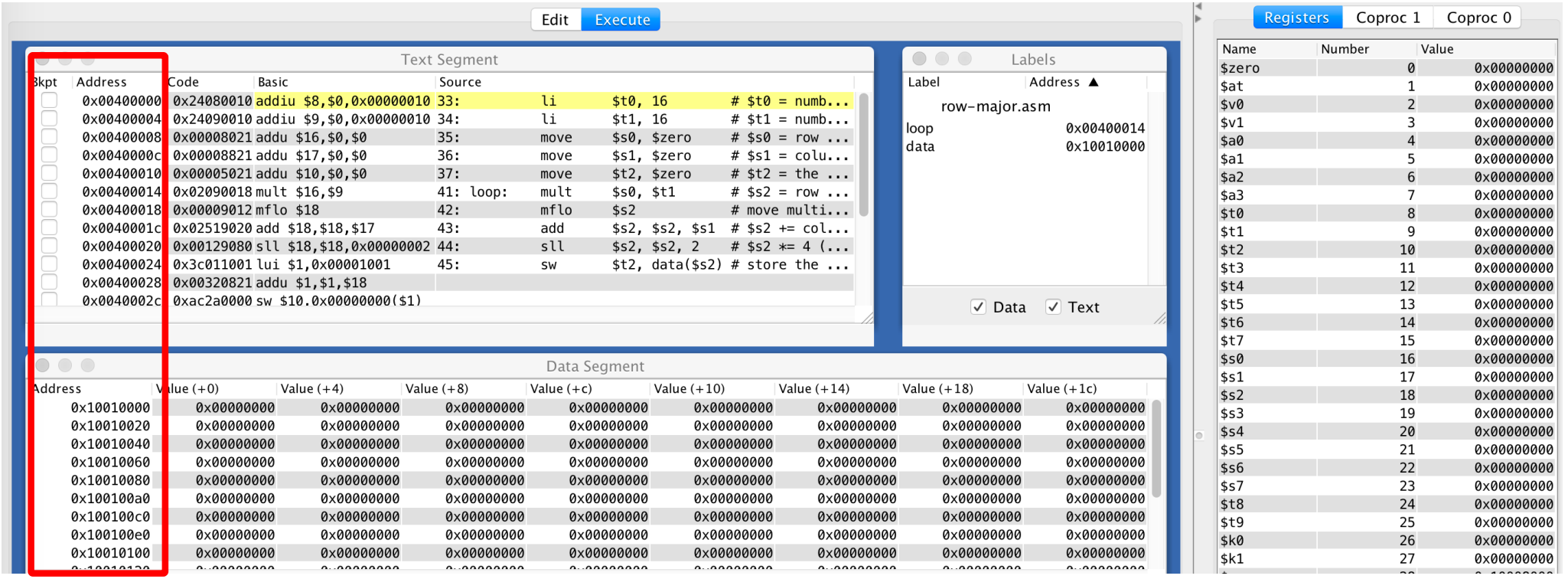
							P	96	`	112	p
							Q	97	a	113	q
							R	98	b	114	r
							S	99	c	115	s
							T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters.

- **Each instruction is encoded as 32-bit numbers**

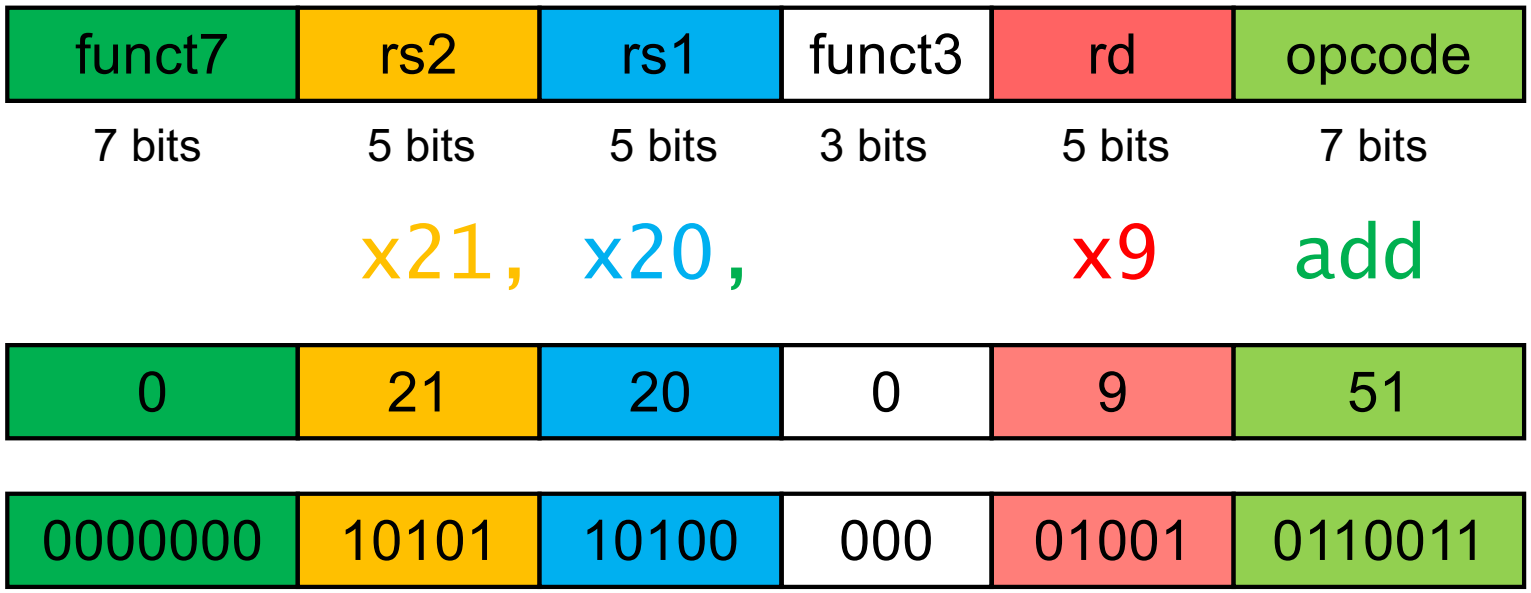
Instruction and Data (2/2)

- Are all numbers stored as binary format in memory
  - It is up to the CPU on how to interpret and do with them
- Each byte/word has its memory address



R-Format Encoding for Arithmetic-Logic Instructions

`add x9,x20,x21` (add rd, rs1, rs2)

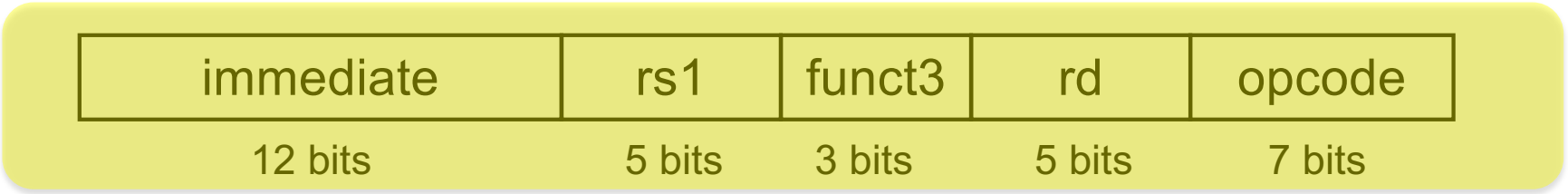


$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

5 bits for rd, rs1 and rs2 because we have 32 registers, thus only needs 5 bit to address a register

RISC-V I-Format Encoding for Instructions That Has Immediate as one of the Operand

- **I-Format: The second source operand is an Immediate**, the first source operand is register, destination operand is register.
- **Immediate arithmetic/logic, and load instructions (NOT store instruction)**
  - `addi x22, x22, 4; Format: addi rd, rs1, #immediate`
  - `ld  x9, 64(x22); Format: ld|lw, rd, #immediate(rs1)`
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended



  - **NOT for store: because destination for store is the memory location (not a register), thus no rd for store.**

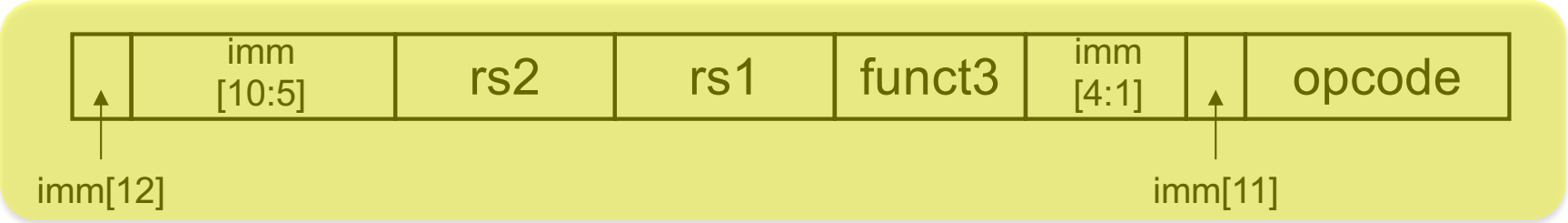
RISC-V S-Format Encoding for Just Store

- S-Format: instructions that use two source register operands and NO destination operand register (rd), **only store instruction**
- **`Format: sd|sw, rs2, #immediate(rs1)`**

- Different immediate format for store instructions
  - `sd  x9, 96(x22);`
  - rs1: base address register number (x22)
  - rs2: source operand register number (x9), which provide the value to be stored to memory
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place as for R- or I-Format

SB-Format Encoding for Branch Instructions

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB-Format instructions: beq x8, x9, 4



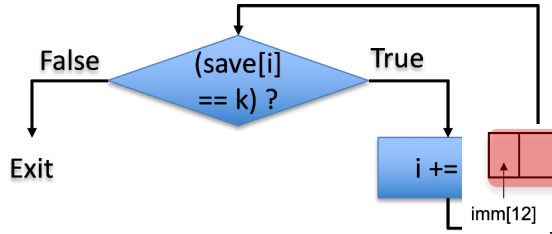
- PC-relative addressing
  - Branch target address is encoded as the offset off the the address of the branch instruction itself
  - Target address = PC (Branch address) + immediate × 2

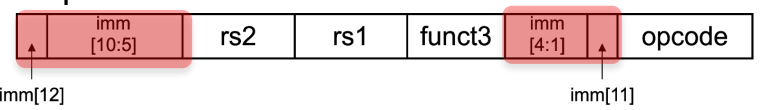
Branch Target Address is Encoded as offset off the branch address

- C code:

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

  – i in x22, k in x24
  – address of save in x25





- RISC-V code: (save[i] is to be read/loaded)

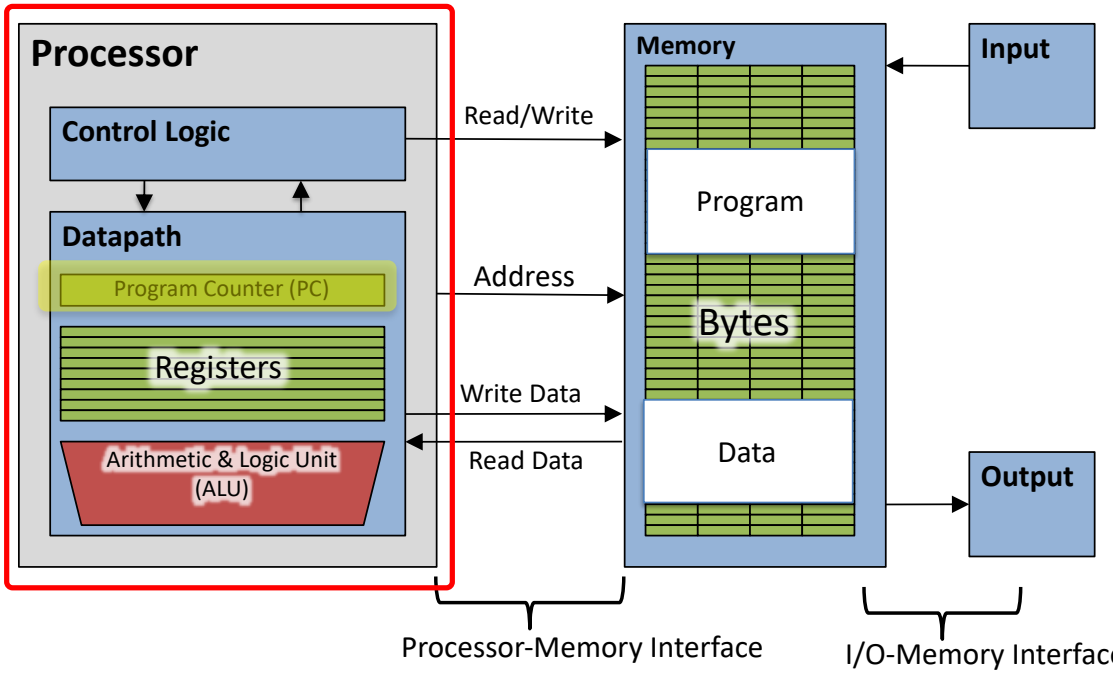
```
Loop: slli x10, x22, 3   //x10 has i*8
      add  x10, x10, x25 //base+offset
      ld   x9,  0(x10)//newbase in x10
      bne  x9,  x24, Exit //false
      addi x22, x22, 1 //true,the loop body,i=i+1
      beq  x0,  x0,  Loop
Exit: …
```

Address	Instruction					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

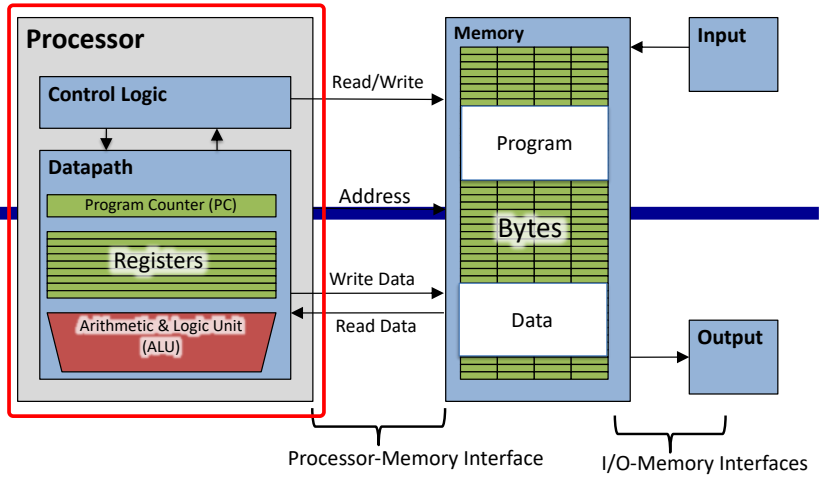
- The Exit offset of the bne is encoded as 6 (..0110)
  – Offset is 6*2 = 12 bytes, i.e. 3 instr forward
  – Exit's address = bne's address (8012) + 12 = 80024 (Exit)
- The Loop offset of the beq is encoded as -10 (..110110)
  – Offset is -10*2 = -20 bytes, i.e. 5 instr backward
  – Loop's address = beq's address (80020) + -20 = 80000 (Loop)

Components of a Computer

- Program instructions and data are all stored in memory
  - Instruction need to be loaded from memory in order to be executed
    - Processor does this automatically, thus no instruction needed
    - **Program counter (PC): a register that stores the address of the execution the process is executing**
  - Data need to be loaded from memory to register in order to be processed: load and store instructions

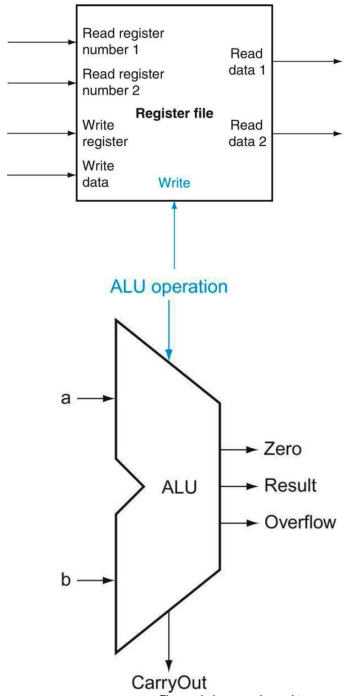


Instruction Execution



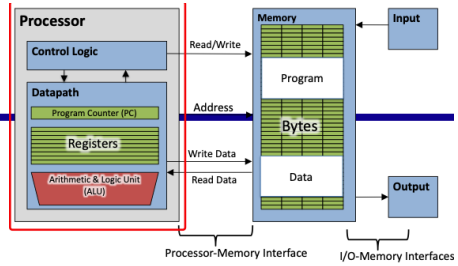
0x0FFE1230: add x6, x12, x13
0x0FFE1234: lw x6, 24(x12)
0x0FFE1238: sw x13, 24(x12)
0x0FFE123C: beq x12, x13, offset

1. Processor fetches an instruction word from instruction memory
   - PC → register to store address to access instruction memory to fetch instruction
2. The instruction word is decoded to know the source operands (register numbers), and then registers are read to have source operand values ready
   - Register numbers → register file, read registers (rs1 and rs2)
     - x12 and x13 for add; x12 for lw, x12 and x13 for sw, x12 and x13 for beq
3. **Use ALU to calculate**
   - **Depending on instruction class**
     - **Arithmetic result: [x12] + [x13]**
     - **Memory address for load/store: 24+[x12], add operation**
     - **Branch condition: x12 ?= x13 ➔[x12]-[x13] and check result is 0 or not**
     - **Branch target address: PC ← target address or PC + 4: pc = [pc]+offset*2 if branch is taken**
4. **LW|SW: access data memory:**
   - **load from mem[32+[x12]]**
   - **Store [x13] to mem[32+[x12]]**
5. **Write result to register**
   - **Arithmetic (add): write result back to the register file x6**
   - **Load: write to register x6**



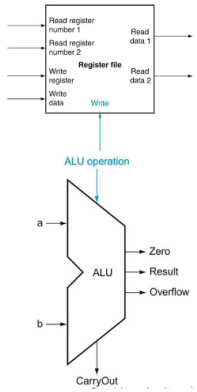
CPU Overview

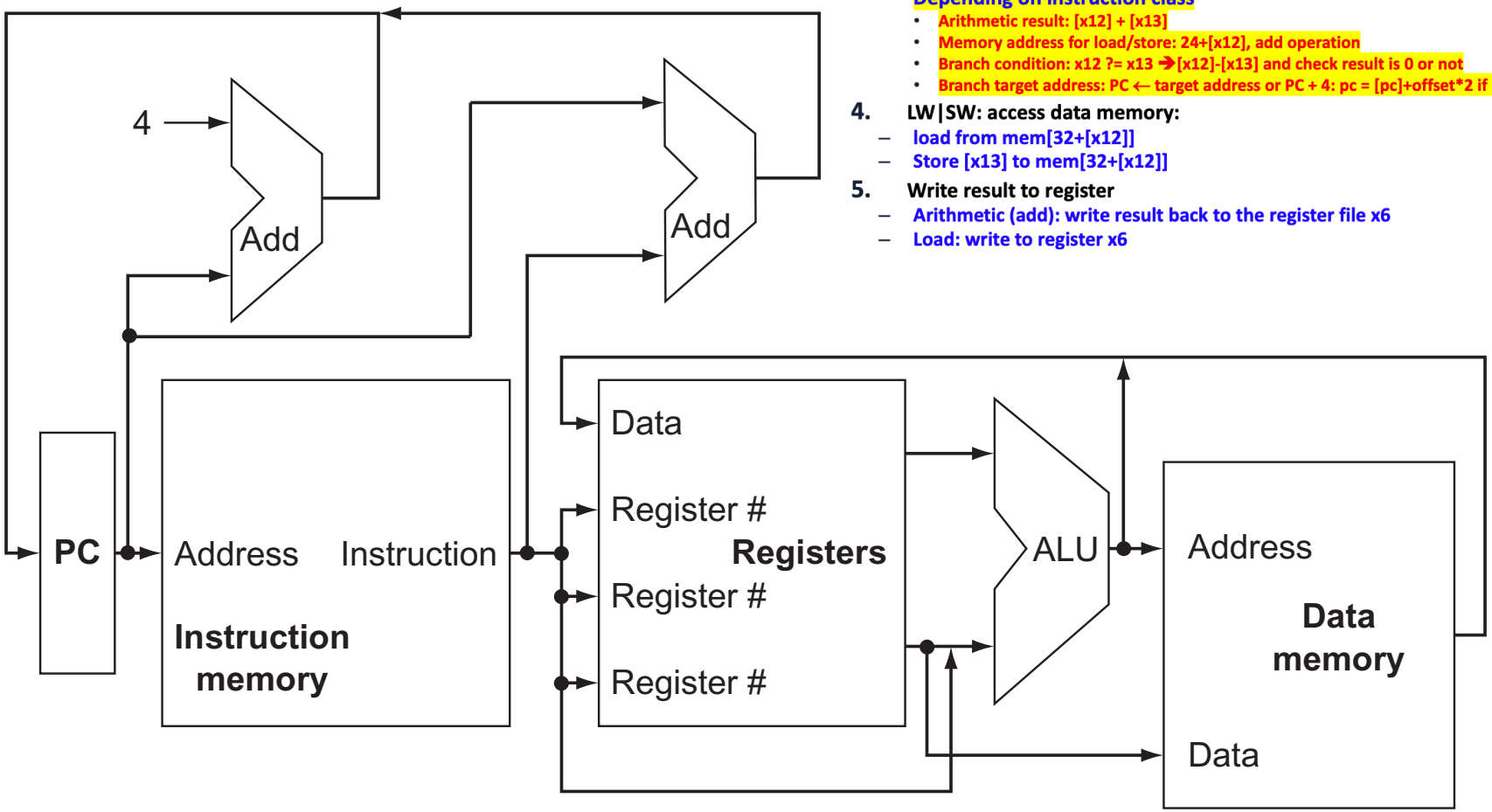
Instruction Execution



0x0FFE1230: add x6, x12, x13
0x0FFE1234: lw x6, 24(x12)
0x0FFE1238: sw x13, 24(x12)
0x0FFE123C: beq x12, x13, offset

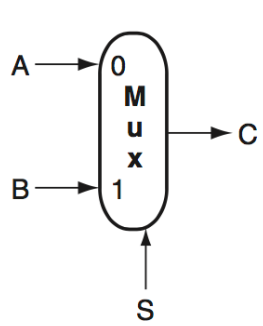
1. Processor fetches an instruction word from instruction memory
   - PC → register to store address to access instruction memory to fetch instruction
2. The instruction word is decoded to know the source operands (register numbers), and then registers are read to have source operand values ready
   - Register numbers → register file, read registers (rs1 and rs2)
     - x12 and x13 for add; x12 for lw, x12 and x13 for sw, x12 and x13 for beq
3. **Use ALU to calculate**
   - **Depending on instruction class**
     - **Arithmetic result: [x12] + [x13]**
     - **Memory address for load/store: 24+[x12], add operation**
     - **Branch condition: x12 ?= x13 ➔[x12]-[x13] and check result is 0 or not**
     - **Branch target address: PC ← target address or PC + 4: pc = [pc]+offset*2 if branch is taken**
4. **LW|SW: access data memory:**
   - **load from mem[32+[x12]]**
   - **Store [x13] to mem[32+[x12]]**
5. **Write result to register**
   - **Arithmetic (add): write result back to the register file x6**
   - **Load: write to register x6**

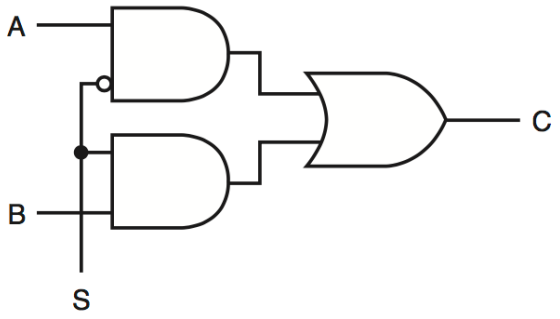




Multiplexers

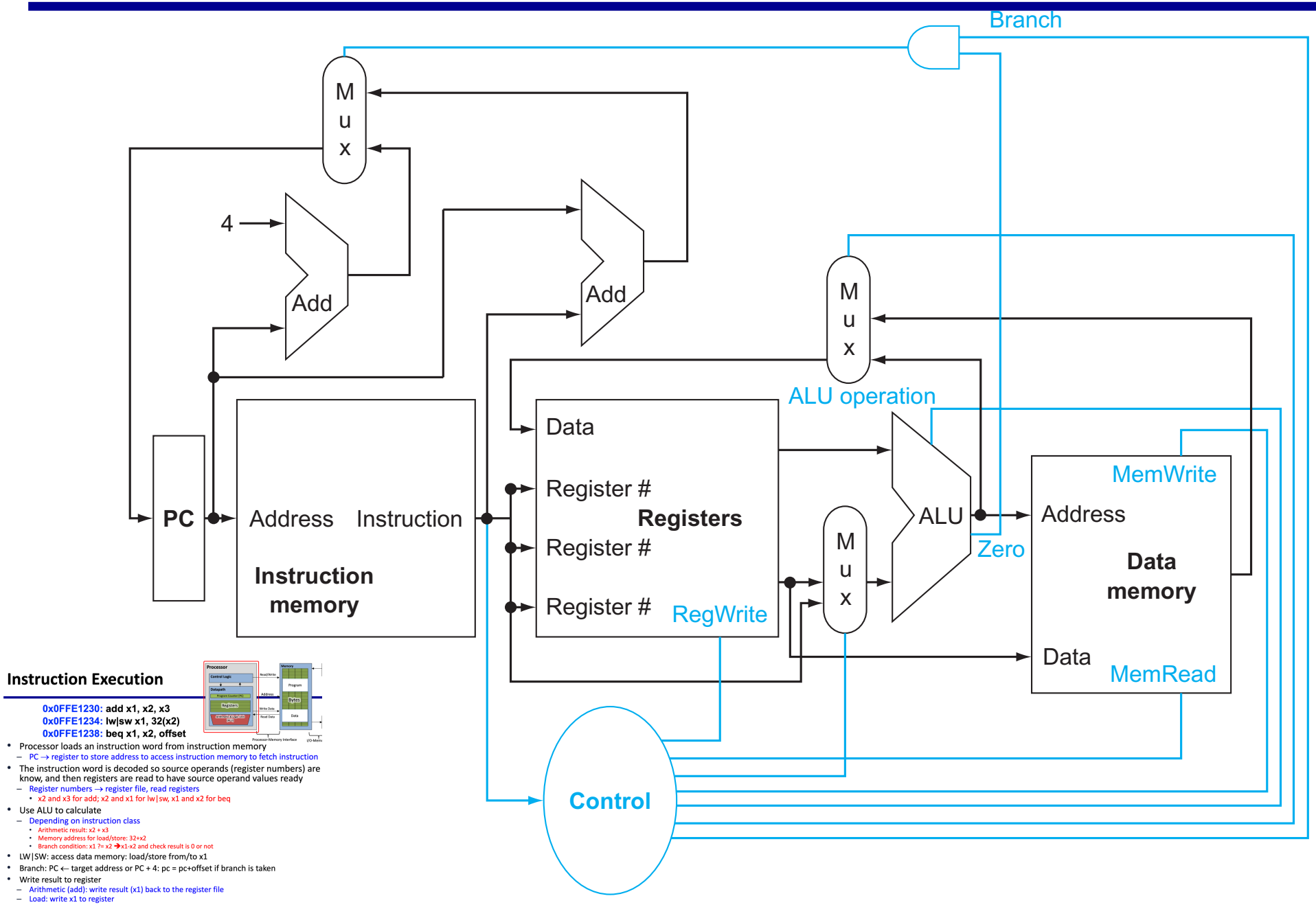
- Can't just join wires together
  - Use multiplexers





Textbook A.3

Full CPU with Data and Control Path (Wires)



Instruction Execution

0x0FFE1230: add x1, x2, x3
0x0FFE1234: lw|sw x1, 32(x2)
0x0FFE1238: beq x1, x2, offset

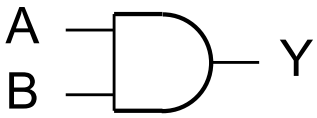
- Processor loads an instruction word from instruction memory
  - PC → register to store address to access instruction memory to fetch instruction
- The instruction word is decoded so source operands (register numbers) are know, and then registers are read to have source operand values ready
  - Register numbers → register file, read registers
    - x2 and x3 for add; x2 and x1 for lw|sw, x1 and x2 for beq
- Use ALU to calculate
  - Depending on instruction class
    - Arithmetic result: x2 + x3
    - Memory address for load/store: 32+x2
    - Branch condition: x1 ?= x2 ➔ x1-x2 and check result is 0 or not
- LW|SW: access data memory: load/store from/to x1
- Branch: PC ← target address or PC + 4: pc = pc+offset if branch is taken
- Write result to register
  - Arithmetic (add): write result (x1) back to the register file
  - Load: write x1 to register

Logic Design Basics

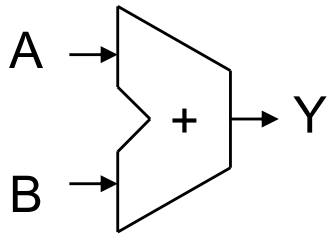
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

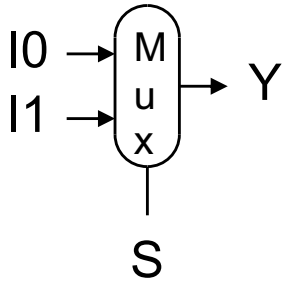
- AND-gate
  - Y = A & B

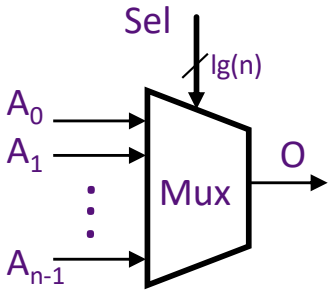


- Adder
  - Y = A + B

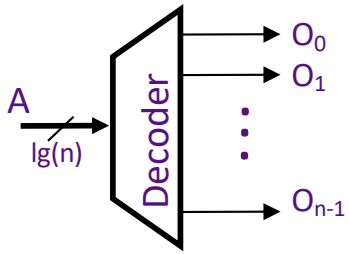


- Multiplexer
  - Y = S ? I1 : I0

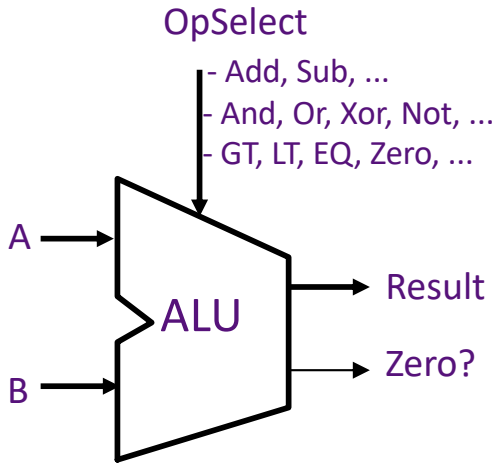




- Decoder

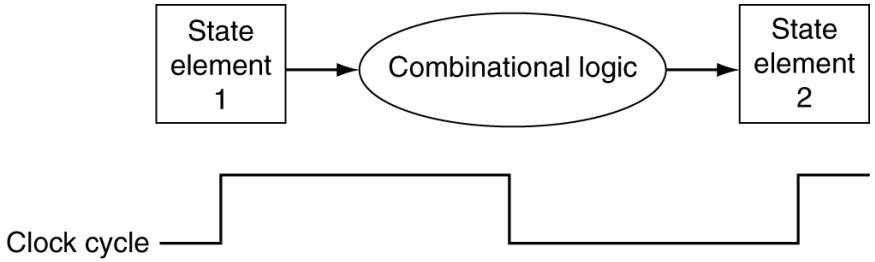


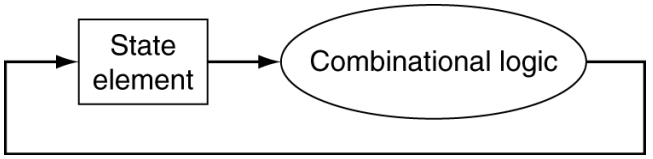
- Arithmetic/Logic Unit
  - Y = F(A, B)



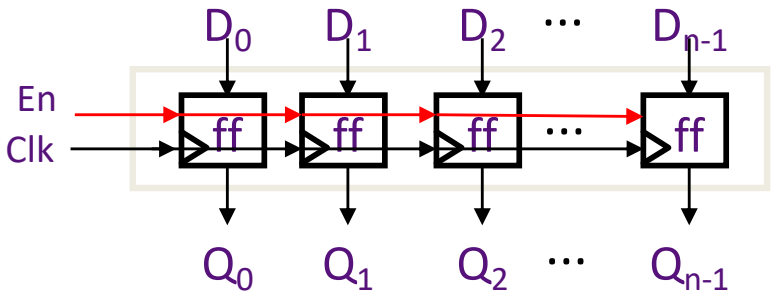
Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

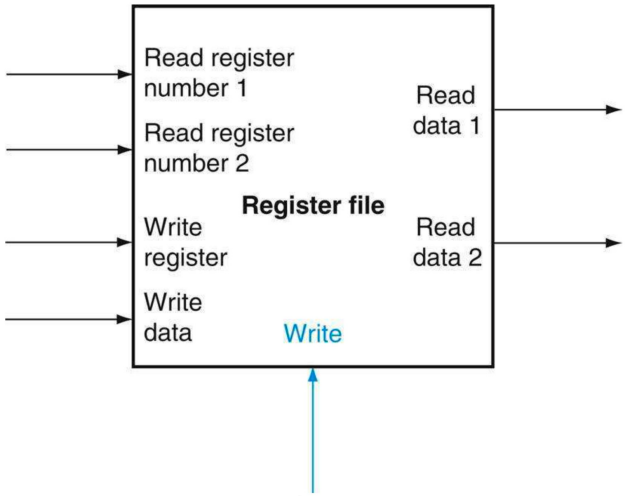




Register Files

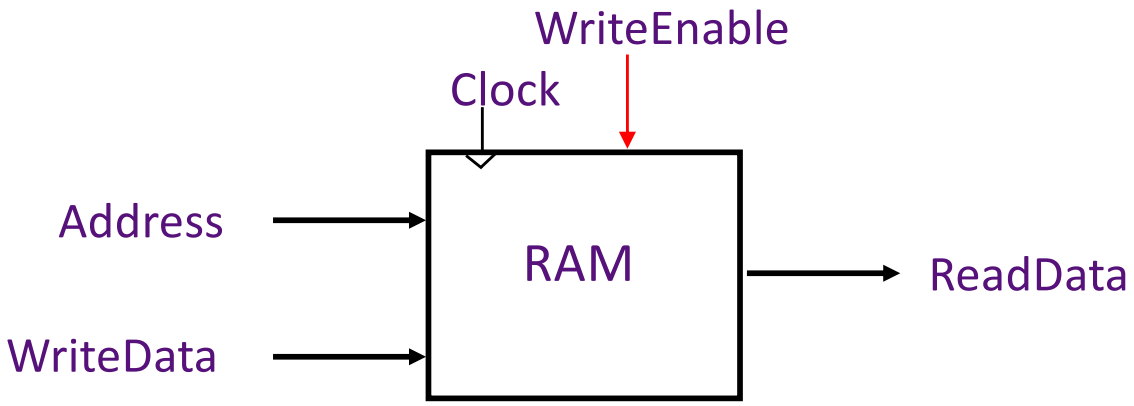


- Reads are combinational
  - Can read in any cycle and for multiple reads
  - Only 2 register source operands needed



- Calculate the number of input/output wires of the register file
  - Read register 1 , Read register 2 and Write register each needs 5 bits (5 wires) since we have 32 32-bit registers
  - Read Data 1, Read Data 2 and Write Data each has 32 bits
  - Write needs one wire (one bit each).

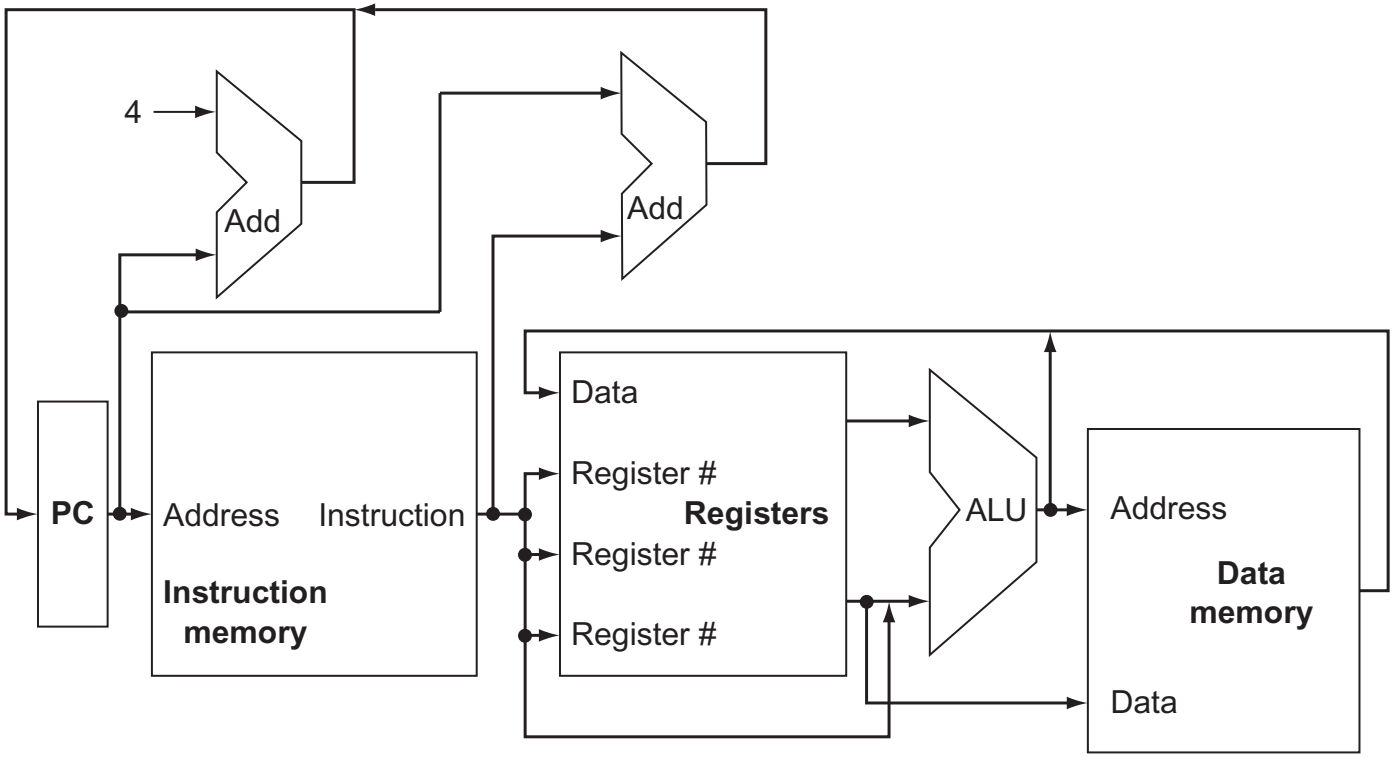
A Simple Memory Model



- Reads and writes are always completed in one cycle
- Read can be done any time (i.e. combinational)
- Write is performed at the rising clock edge
  - if it is enabled
- The number of wires for RAM (Random Access Memory)
  - Address has 32 bits
  - WriteData and ReadData each has 32 bits
  - WE and Clock each needs one wire

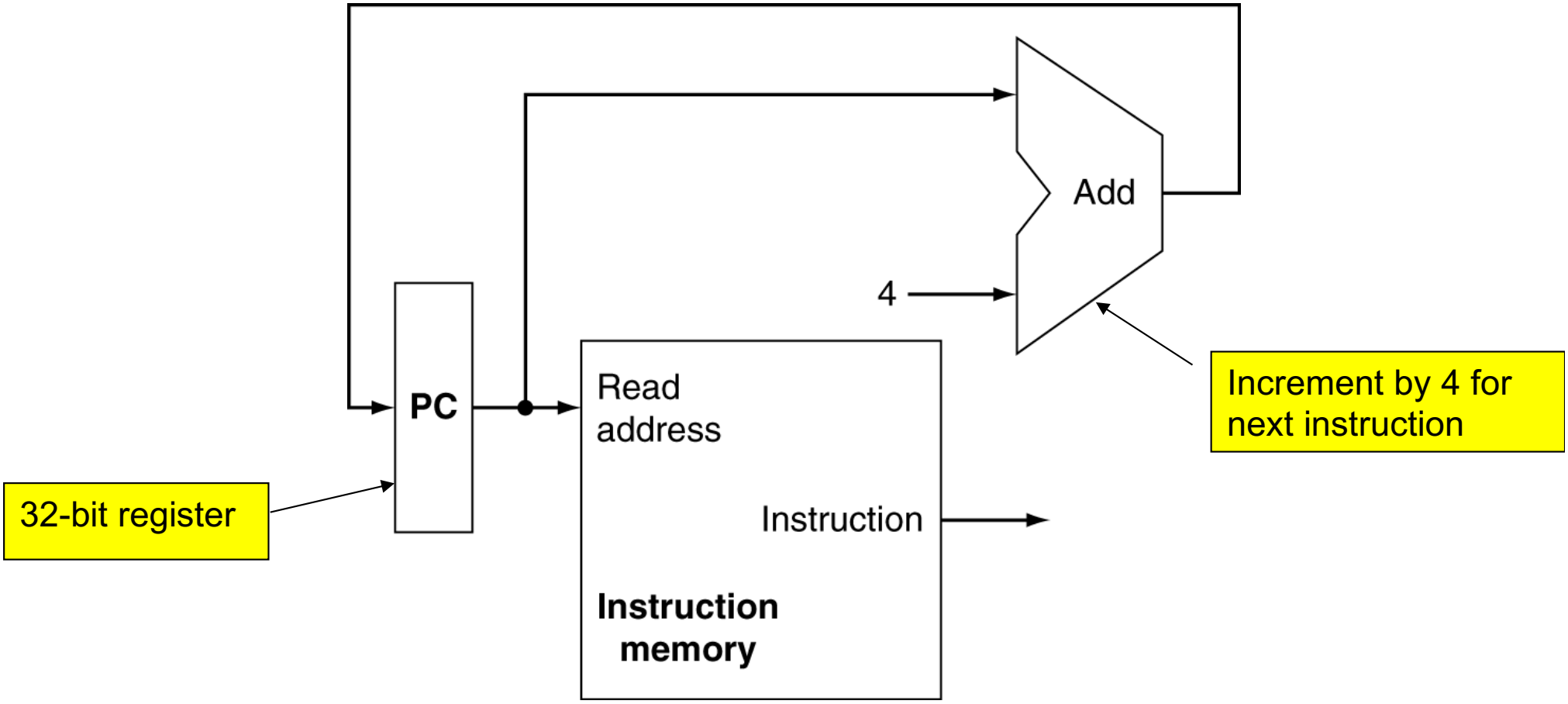
Building a Datapath

- Datapath
  - Elements/wires that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a RISC-V datapath incrementally
  - Refining the overview design



Instruction Fetch

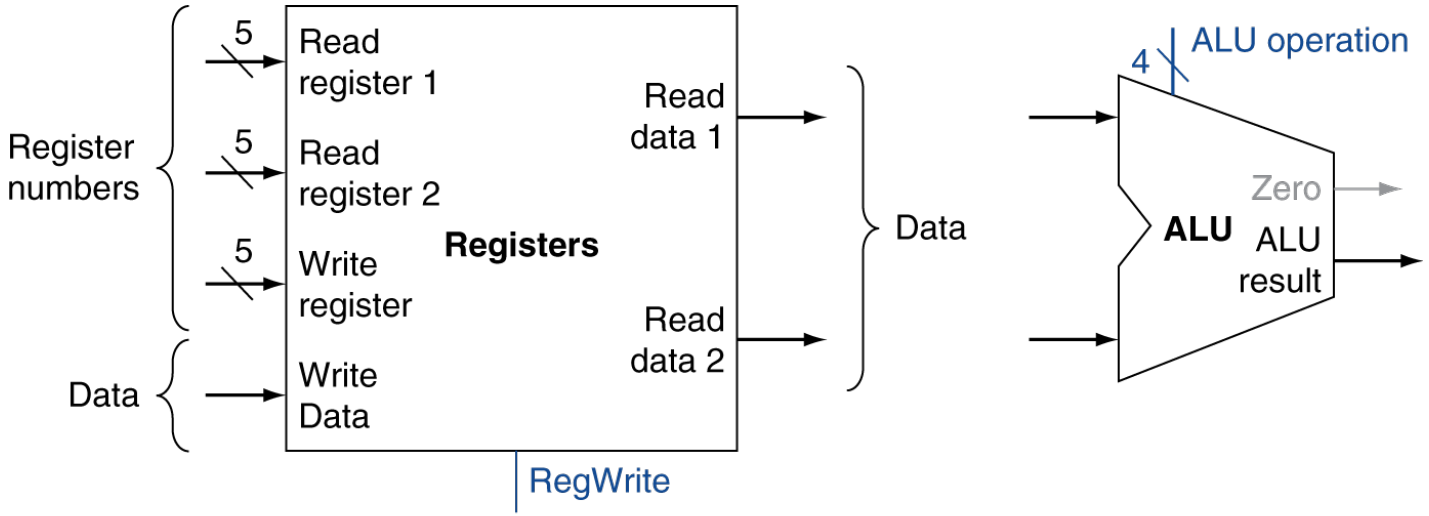
```
0x0FFE1230:  add x6,   x12, x13
0x0FFE1234:  lw    x6,   24(x12)
0x0FFE1238:  sw    x13, 24(x12)
0x0FFE123C: beq  x12, x13, offset
```



R-Format Instructions

- Read two register operands
  - x12 and x13
- Perform arithmetic/logical ops
  - [x12] + [x13], ALU operation is +
- Write register result
  - x6 ← [x12] + [x13], RegWrite is on

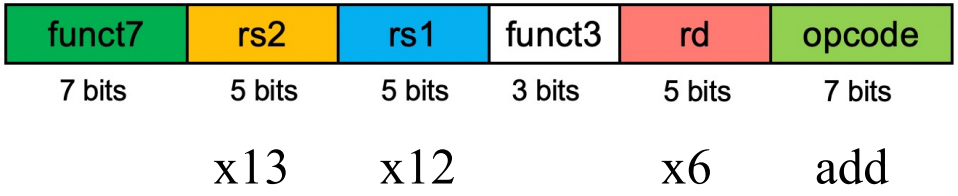
```
0x0FFE1230: add x6,   x12, x13
0x0FFE1234: lw  x6,   24(x12)
0x0FFE1238: sw  x13, 24(x12)
0x0FFE123C: beq x12, x13, offset
```



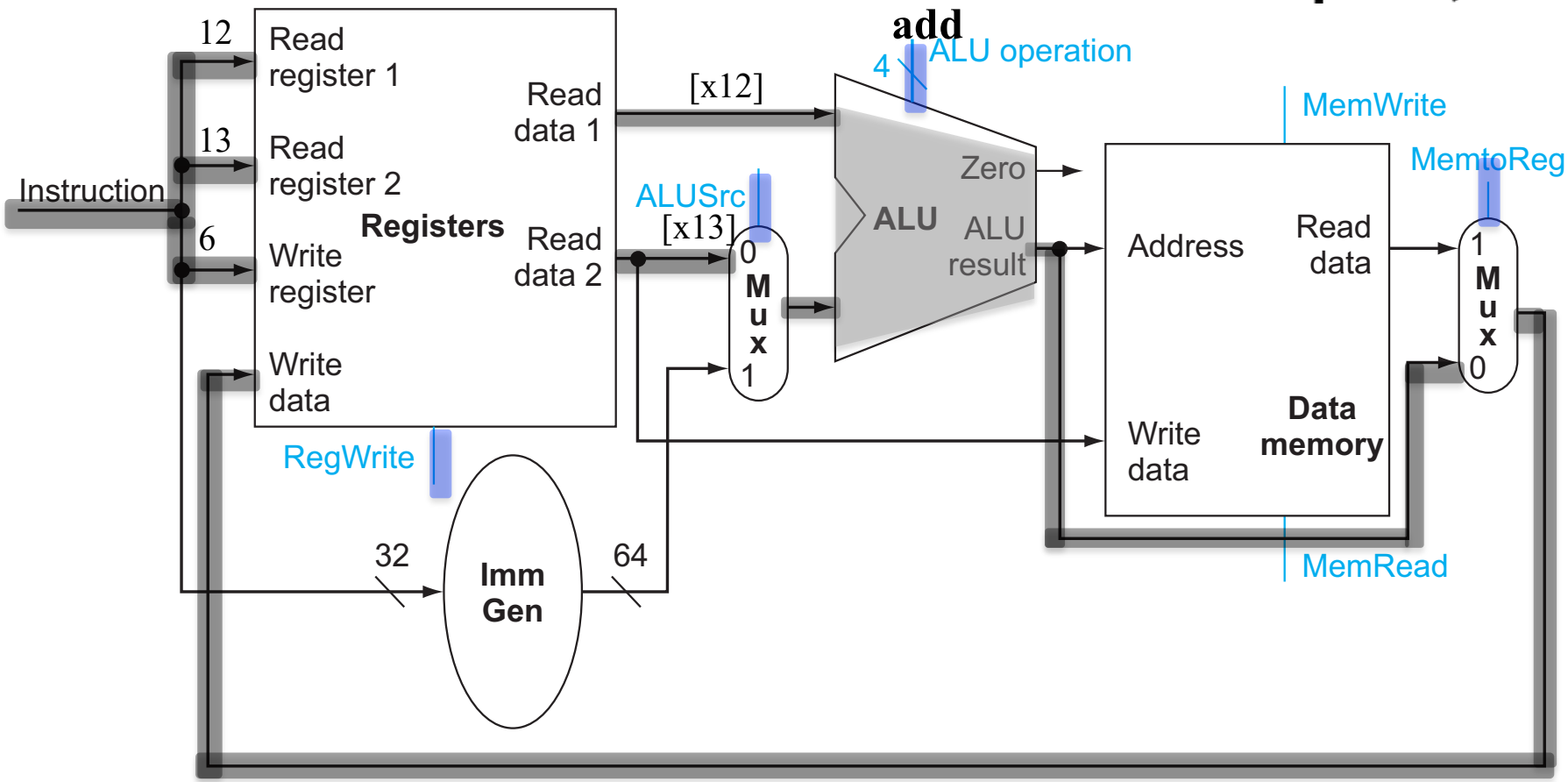
a. Registers

b. ALU

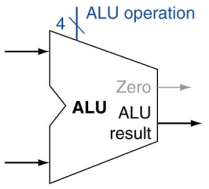
R-Type Datapath




```
0x0FFE1230: add x6,   x12, x13
0x0FFE1234: lw  x6,   24(x12)
0x0FFE1238: sw  x13, 24(x12)
0x0FFE123C: beq x12, x13, offse
```



Load/Store Instructions



- Fetch source register operands
  - Load: x12
  - Store: x12 and x13
- Calculate address using 12-bit signed offset
  - 24 + [x12]
  - Use ALU, but sign-extend offset
  - ALU operation is +
- Load: Read memory and update register
  - x6 ← MEM(24+[x12])
  - MemRead signal is on
- Store: Write register value to memory
  - [x3] → MEM(24+[x12])
  - MemWrite is on

```
0x0FFE1230:  add x6,   x12, x13
0x0FFE1234:  lw   x6,   24(x12)
0x0FFE1238:  sw   x13, 24(x12)
0x0FFE123C:  beq  x12, x13, offs
```

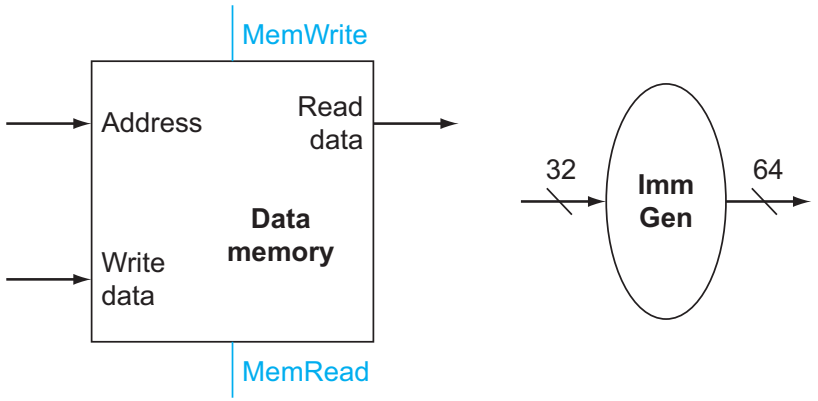


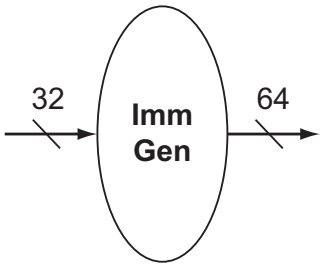
Figure 4.8. Imm Gen: generate 32- or 64-bit immediate value (depending on whether we design 32-bit or 64-bit machine) from an instruction word.

- Select the 12-bit from the instruction word and sign-extended to 32- or 64-bit.
- **Used for for I-, S- and SB-format (I-format ALU, load, store, and beq)**

Immediate Generator

- Figure 4.8. Imm Gen: generate 32- or 64-bit immediate value (depending on whether we design 32-bit or 64-bit machine) from an instruction word.
  - Select the 12-bit from the instruction word and sign-extended to 32- or 64-bit.
  - **Used for for I-, S- and SB-format (I-format ALU, load, store, and beq)**

- **Elaboration on Imm generation:**
  - **Last paragraph of 4.3, page 251**



b. Immediate generation unit

Load Datapath

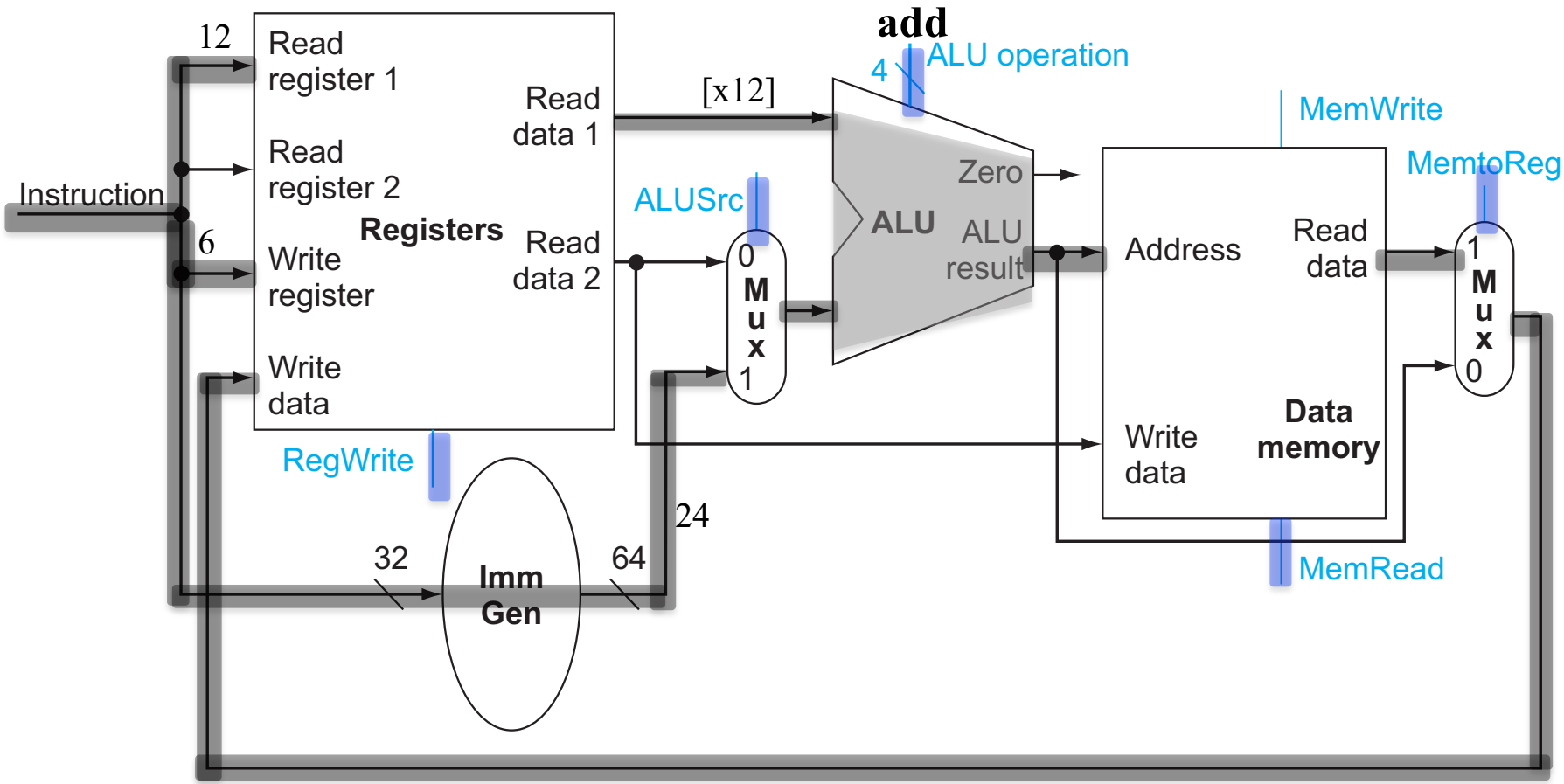
immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits
24	x12		x6	load

0x0FFE1230: add x6, x12, x13

0x0FFE1234: lw x6, 24(x12)

0x0FFE1238: sw x13, 24(x12)

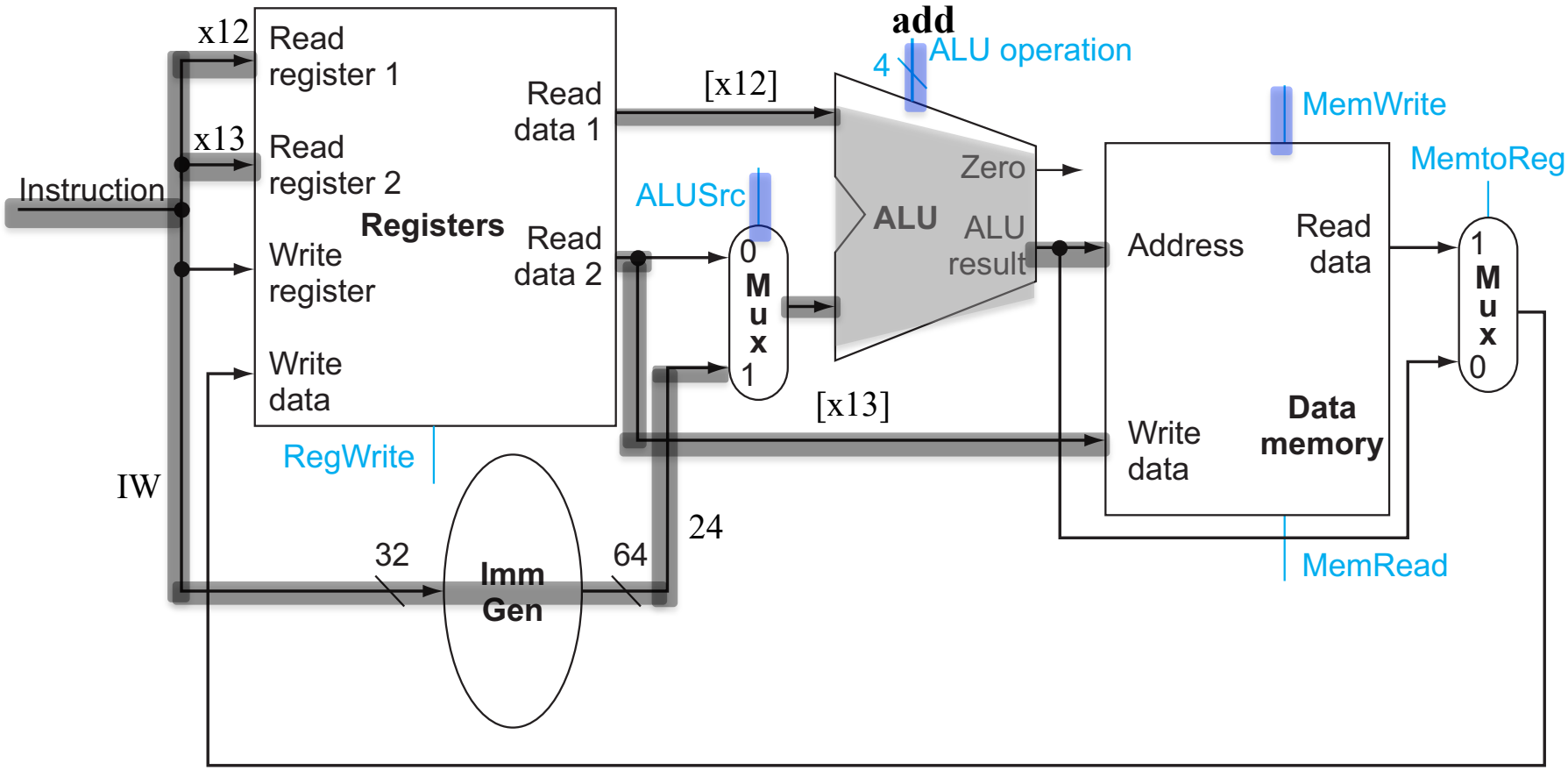
0x0FFE123C: beq x12, x13, offset



Store Datapath

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
32	x1	x2			store

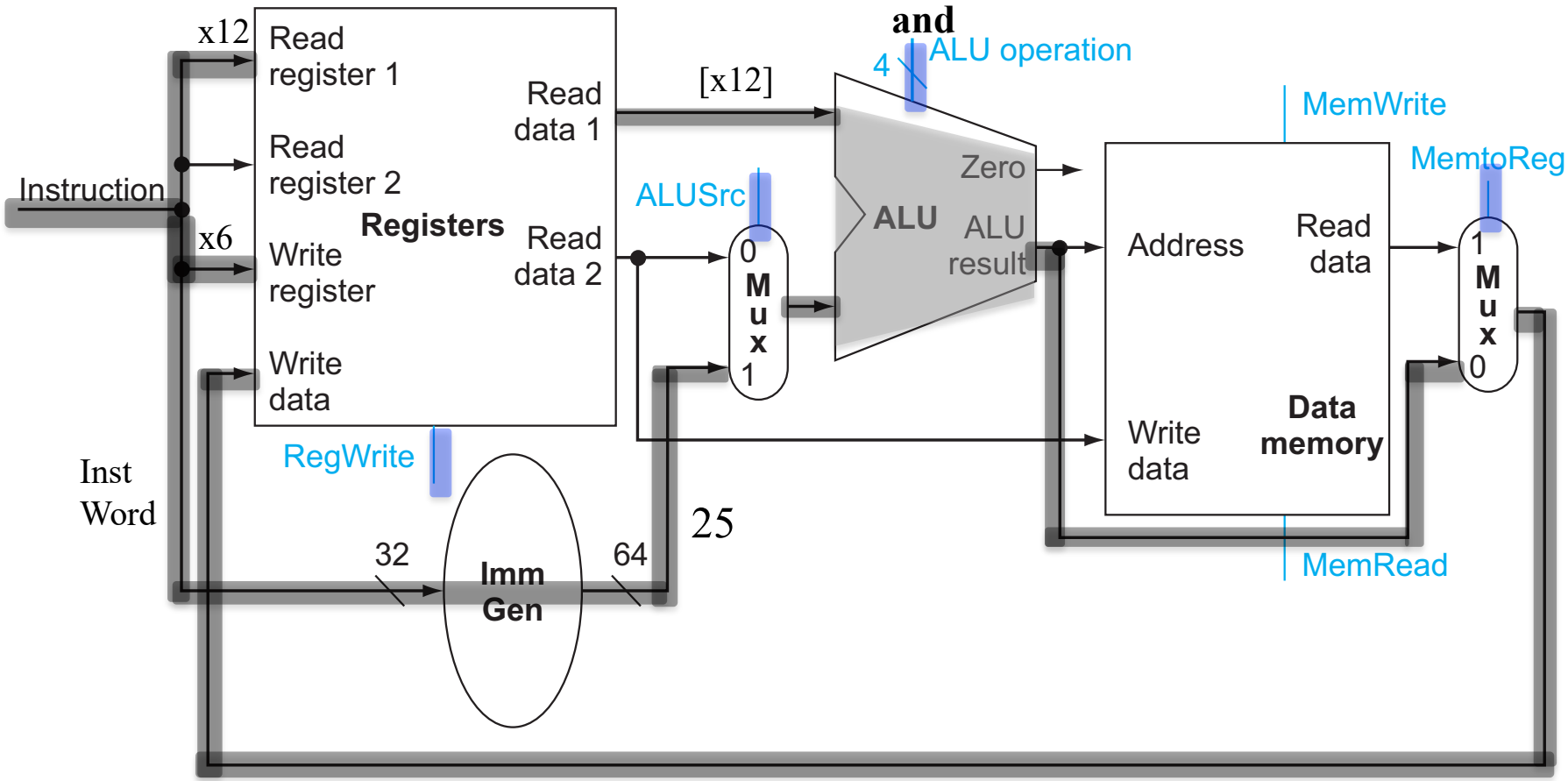
0x0FFE1230: add x6, x12, x13
0x0FFE1234: lw x6, 24(x12)
0x0FFE1238: sw x13, 24(x12)
0x0FFE123C: beq x12, x13, offset



Andi Datapath

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits
25	x12		x6	andi

Andi x6, x12, 25

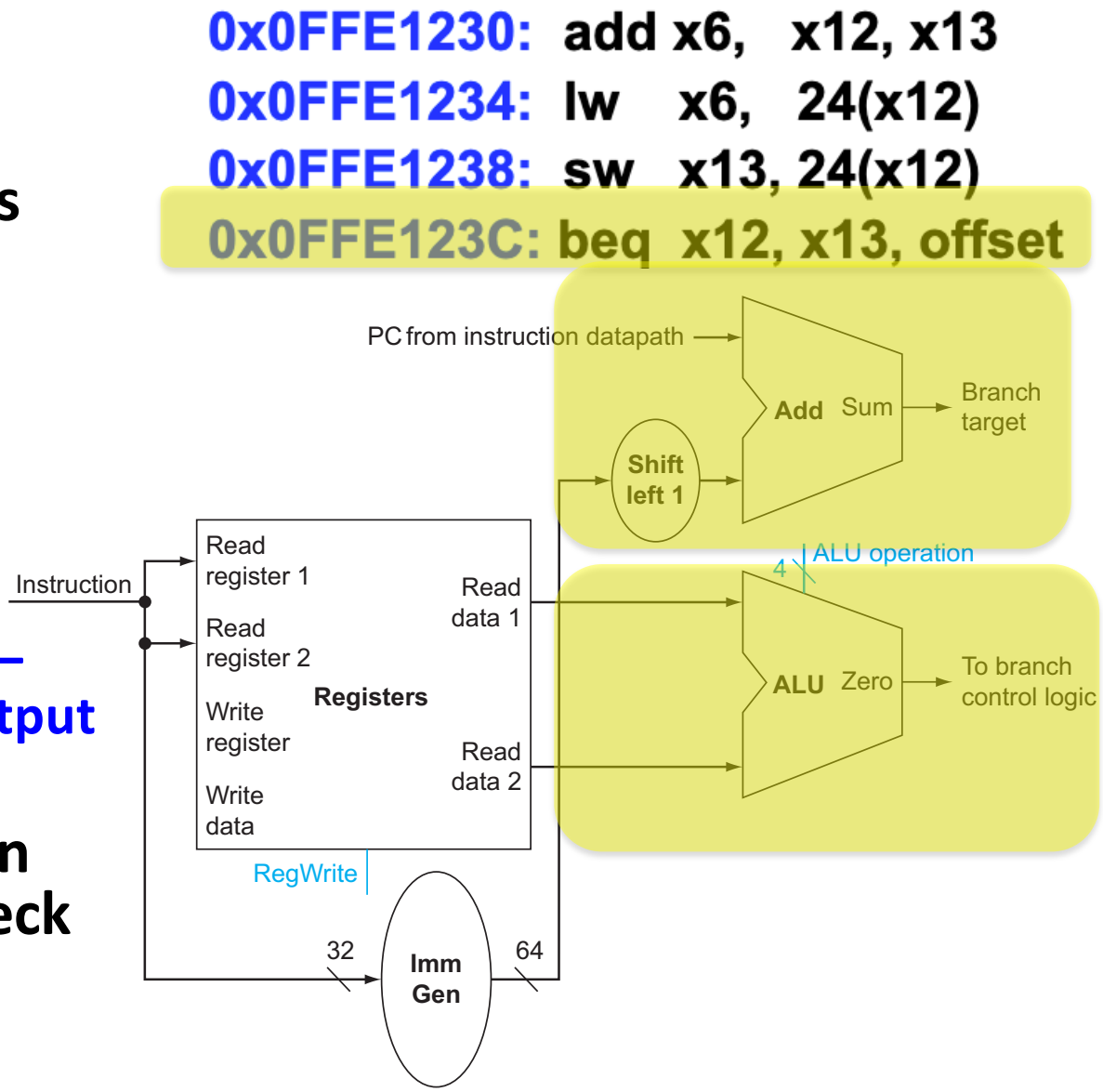


Branch Instructions

- Read register operands
  - x12 and x13

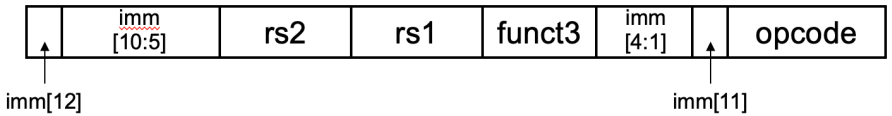
1. **Calculate target address (pc + offset * 2)**
   - **Shift left 1 places**
     - **Offset * 2**
   - **Add to PC**
2. **Branch condition: Compare operands**
   - **Use ALU, subtract ([x12] – [x13]) and check Zero output**

- **Target address calculation and branch condition check can be performed at the same time**



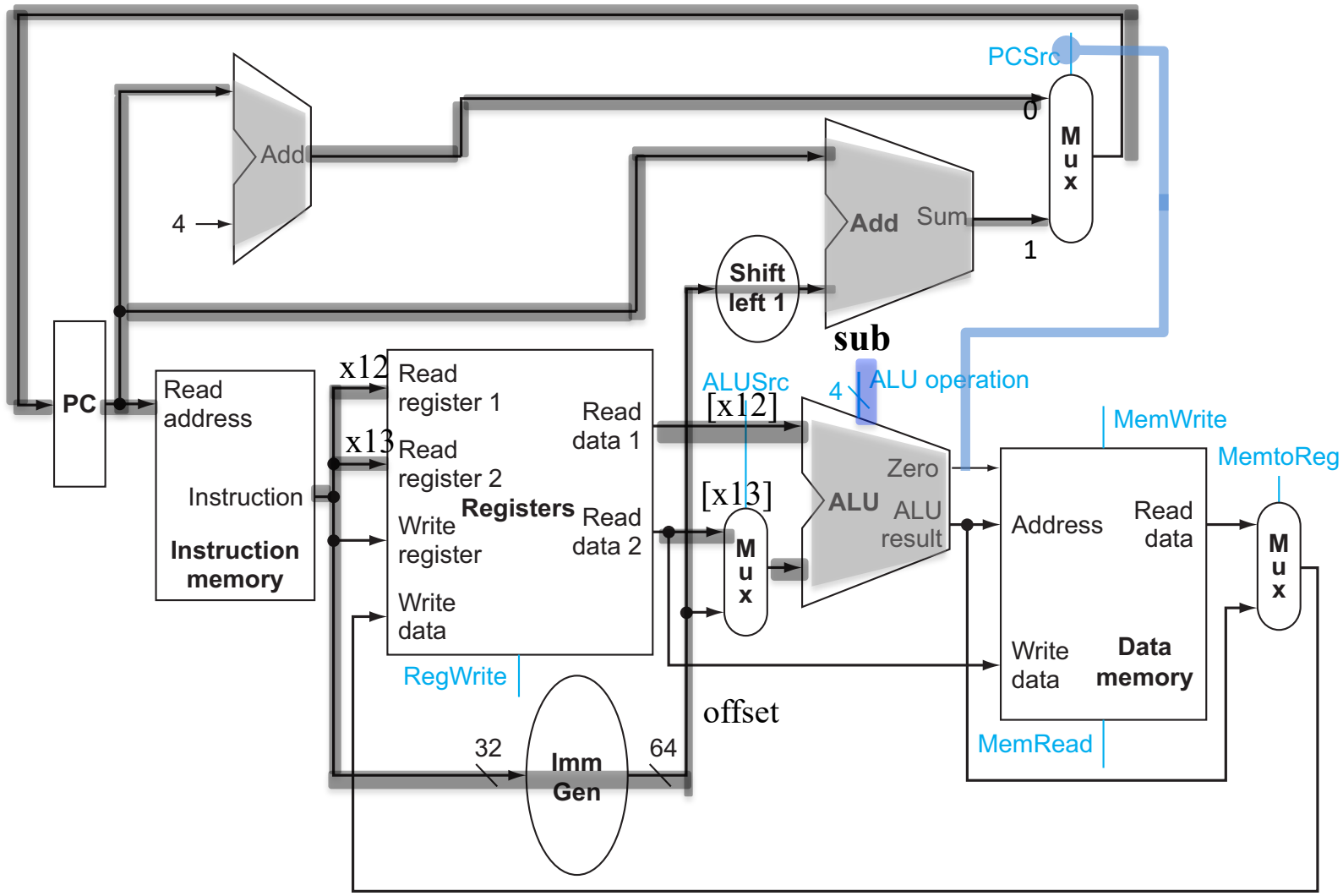
Beq Datapath

```
0x0FFE1230: add x6,   x12, x13
0x0FFE1234: lw   x6,   24(x12)
0x0FFE1238: sw   x13, 24(x12)
0x0FFE123C: beq  x12, x13, offset
```



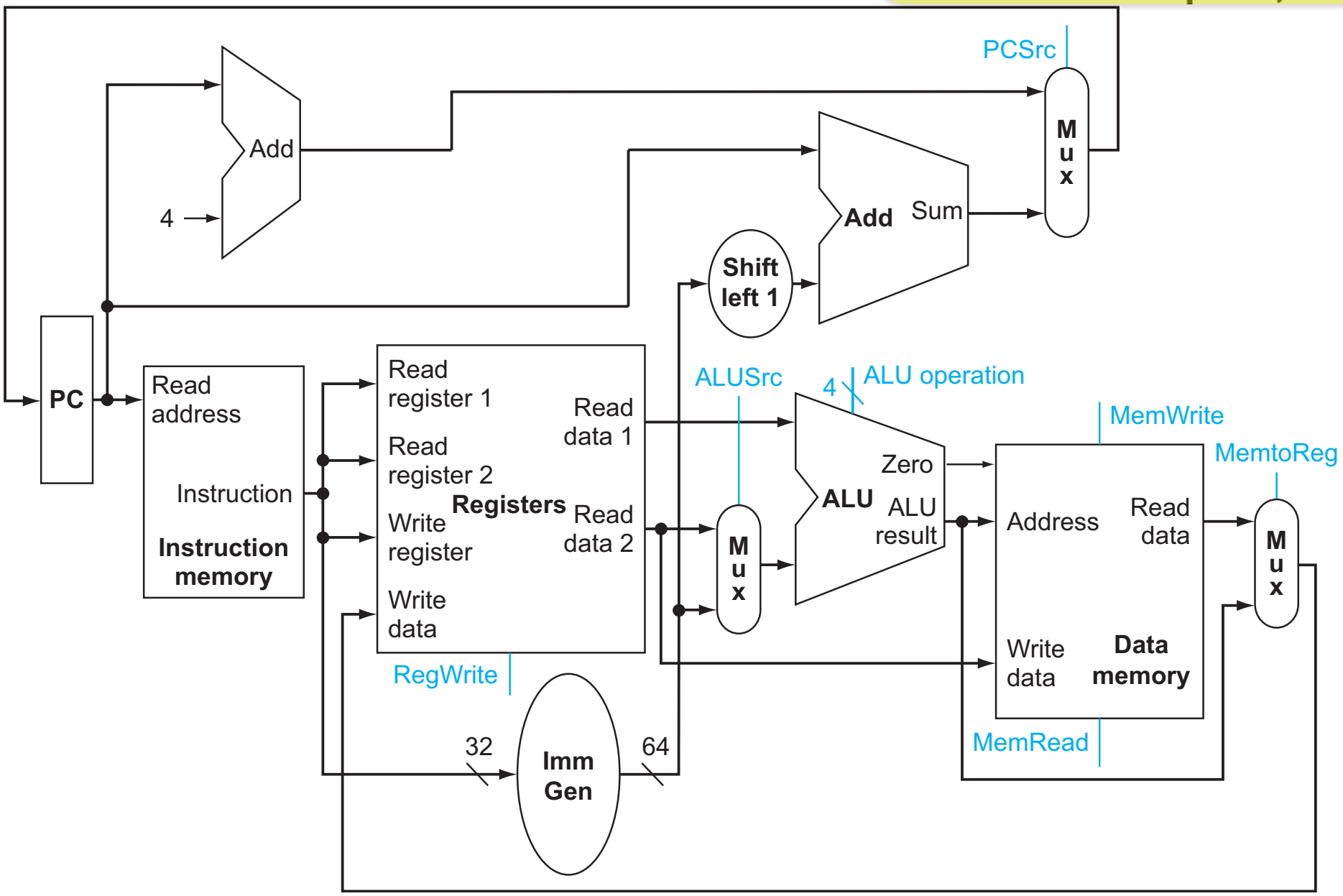
offset x13 x12 beq

Branch target address is PC + Imm * 2 (Slide #20)



Full Datapath

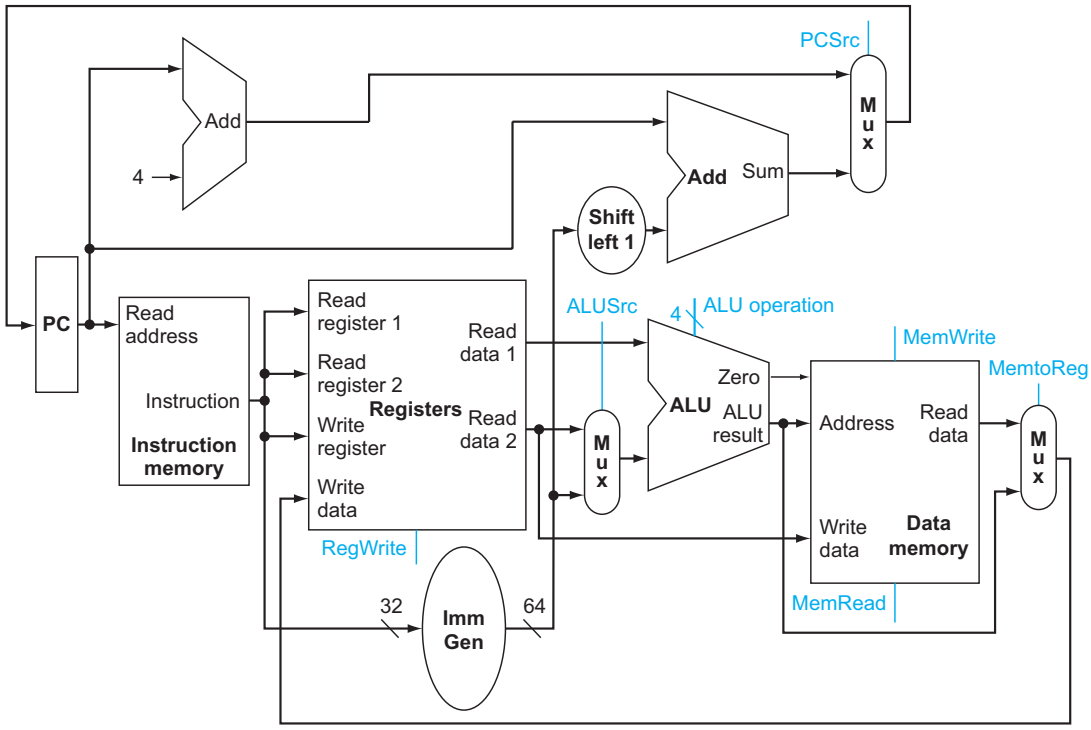
```
0x0FFE1230: add x6,   x12, x13
0x0FFE1234: lw   x6,   24(x12)
0x0FFE1238: sw   x13, 24(x12)
0x0FFE123C: beq  x12, x13, offse
```



Full Datapath - Study Goals

```
0x0FFE1230: add x6,   x12, x13
0x0FFE1234: lw   x6,   24(x12)
0x0FFE1238: sw   x13, 24(x12)
0x0FFE123C: beq  x12, x13, offset
```

- Know what each component does
  - PC, two adders, IM, Registers, Mux, ALU, DM, Imm-Gen
- Know what each line does and their width
  - Data path
  - Control path
- Given an instruction, mark the lines that the inst uses
  - Add/andi, etc
  - lw and sw
  - Beq
- Given a control signal, know what instructions assert it
  - RegWrite, MemRead, MemWrite, ALUSrc, etc



Chapter 4: The Processor

- **Lecture**
  - **4.1 Introduction**
  - **4.2 Logic Design Conventions**
  - **4.3 Building a Datapath**

☛ **Lecture**
  - **4.4 A Simple Implementation Scheme**

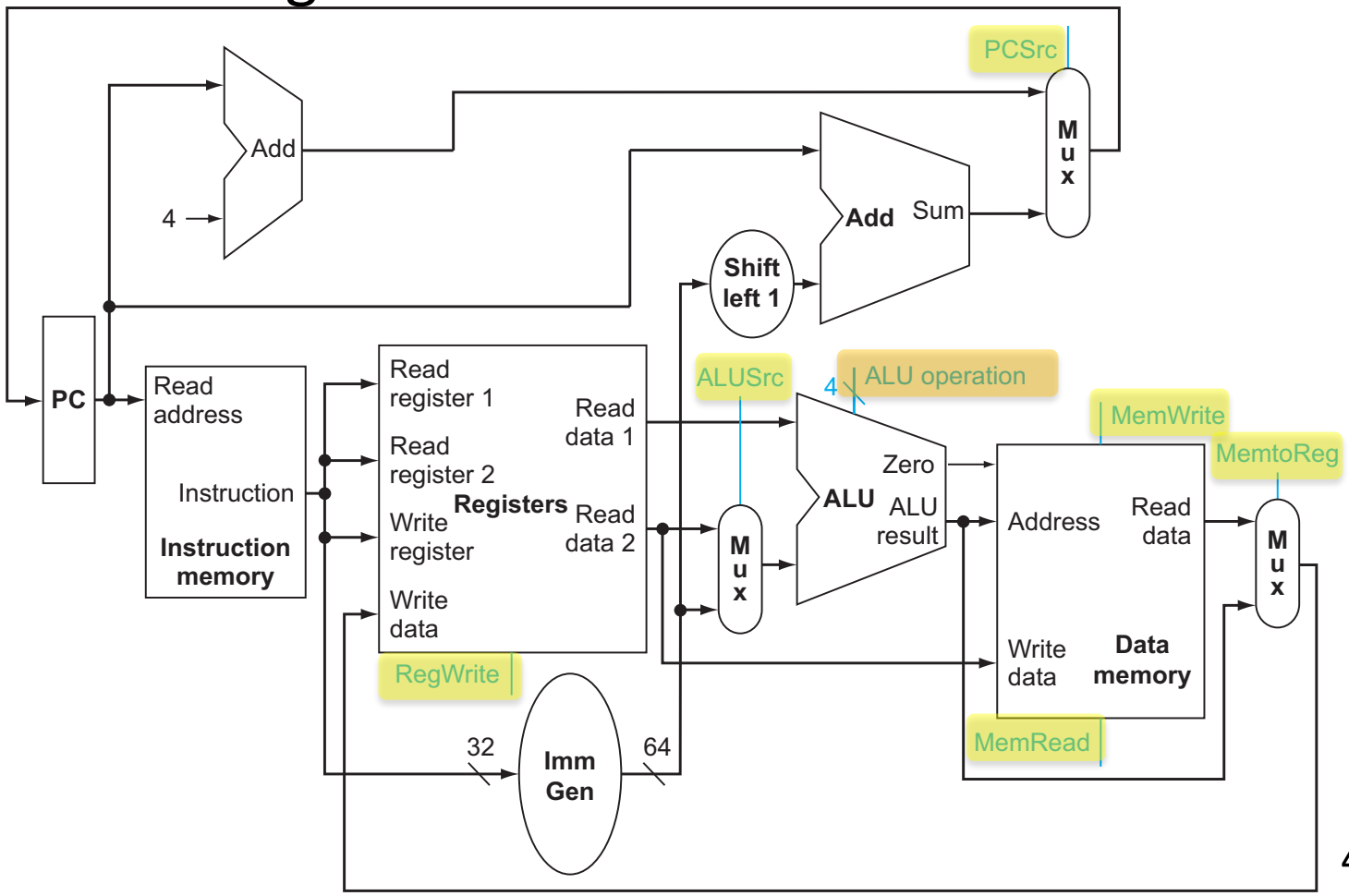
- **Lecture**
  - **4.5 An Overview of Pipelining**

- **Lecture (Pipeline implementation), will not be covered!**
  - 4.6 Pipelined Datapath and Control
  - 4.7 Data Hazards: Forwarding versus Stalling
  - 4.8 Control Hazards
  - ~~4.9 Exceptions~~
  - ~~4.13 Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations~~

- **Lecture (Advanced pipeline techniques and real-world CPU examples)**
  - **4.10 Parallelism via Instructions**
  - **4.11 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines**
  - 4.12 Going Faster: Instruction-Level Parallelism and Matrix Multiply
  - ~~4.14 Fallacies and Pitfalls~~
  - **4.15 Concluding Remarks**

How Those Control Signals are Set Correctly?

- 1 4-bit control: ALU operation
- 6 1-bit control: PCSrc, ALUSrc, RegWrite, MemRead, MemWrite, MemtoReg

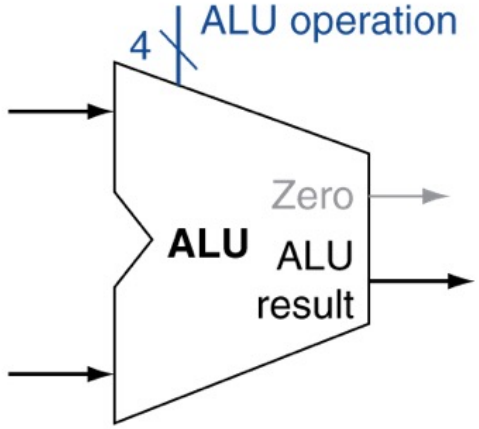


ALU Operation Control

```
0x0FFE1230:  add x6,   x12, x13
0x0FFE1234:  lw    x6,   24(x12)
0x0FFE1238:  sw    x13, 24(x12)
0x0FFE123C: beq  x12, x13, offset
```

- ALU used for
  - Load/Store: Func = add
  - Branch: Func = subtract
  - R-type: Func depends on funct field

ALU operation control	Function
0000	**AND**
0001	**OR**
0010	**add**
0110	**subtract**
0111	set-on-less-than
1100	NOR



- How to generate those control signals
  - **Based on the opcode, func3 and func7 fields of an instruction word**
  - Encoding Review:

R-Format Instruction Encoding (AL Instructions)

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

Arithmetic instructions

RV32I Base Instruction Set

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

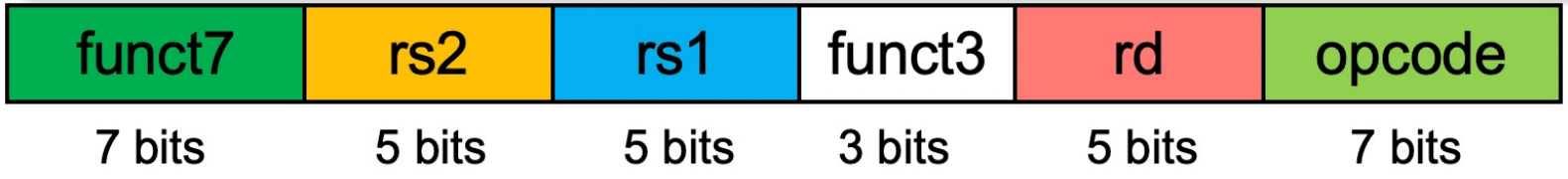
RV64I Base Instruction Set (in addition to RV32I)

0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

Logic instructions

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU

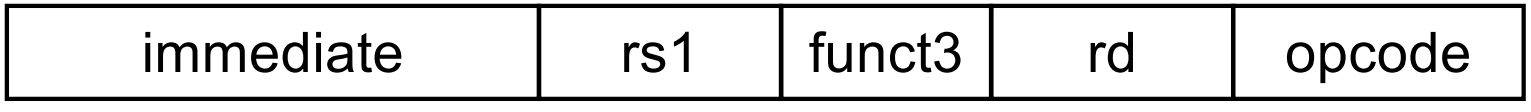


I-Format Instruction Encoding (AL and Load)

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

Immediate arithmetic/logic

load instructions



imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

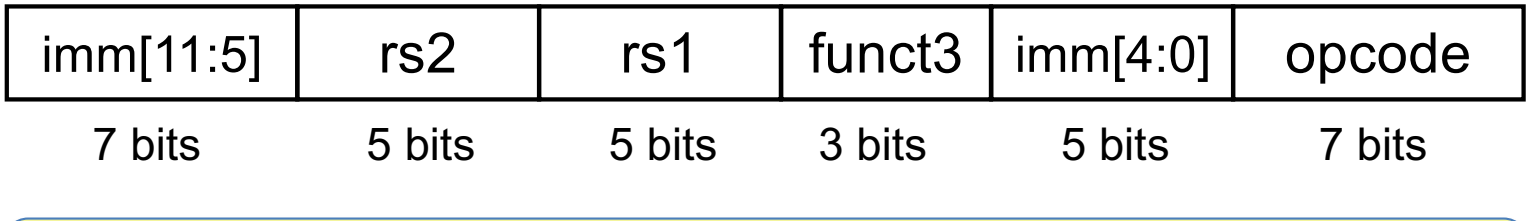
imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI
0000000 shamt	rs1	001	rd	0010011	SLLI
0000000 shamt	rs1	101	rd	0010011	SRLI
0100000 shamt	rs1	101	rd	0010011	SRAI

imm[11:0]	rs1	110	rd	0000011	LWU
imm[11:0]	rs1	011	rd	0000011	LD

imm[11:0]	rs1	000	rd	0011011	ADDIW

S-Format Instruction Encoding (Store)

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116



imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD

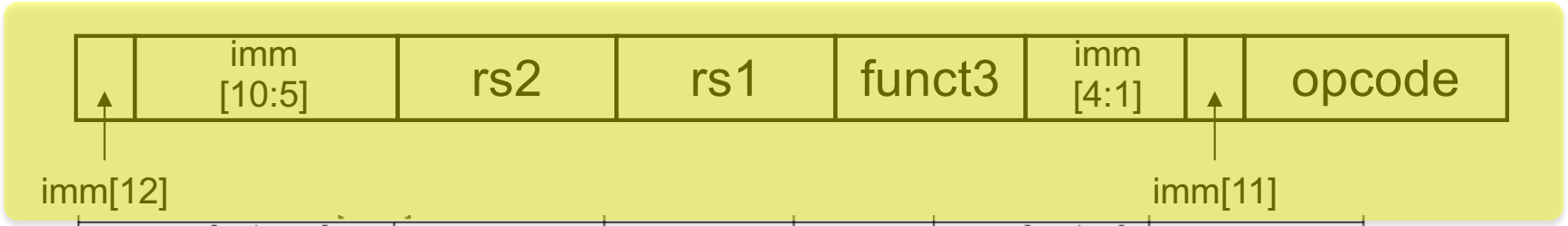
Store instructions

- Same opcode
- Func3 are different for different sizes of data
  - Byte, half-word, word, doubleword

SB-Format Encoding for Branch Instr (e.g. beq)

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

- Branch instructions specify
  - Opcode, two registers, target address
  - Most branch targets are near branch, Forward or backward
- SB-Format instructions: beq x8, x9, 4



imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

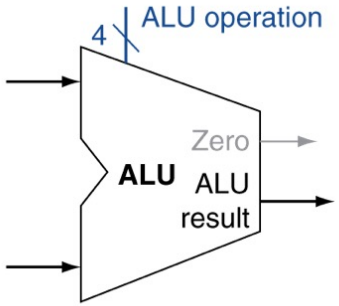
  - Same opcode, func3 are different for different branch instr
- PC-relative addressing
  - Branch target address is encoded as the offset off the the address of the branch instruction itself
  - Target address = PC (Branch address) + immediate × 2

Observation

- Opcode are the same for each basic function category
  - R-format 32-bit AL
  - R-format 64-bit AL
  - I-format AL
  - Load (I-Format)
  - Store (S-Format)
  - Branch (SB-Format)

- Func3 and func7 are different for different operations with each categories
  - To determine the ALU action for the instructions
    - Add, sub, AND, or, etc.

Four Formats of Instruction



- ALU Control input = ALUOp + ALU action
  - **2-bit ALUOp determined by opcode**
  - **ALU action determinded by bit[30, 14-12] of func3/func7**

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

Instruction opcode	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

The Truth Table for ALU Operation

- Control signals derived from instruction opcode/func3/func7
  - Nothing to do with operands (register or immediate)

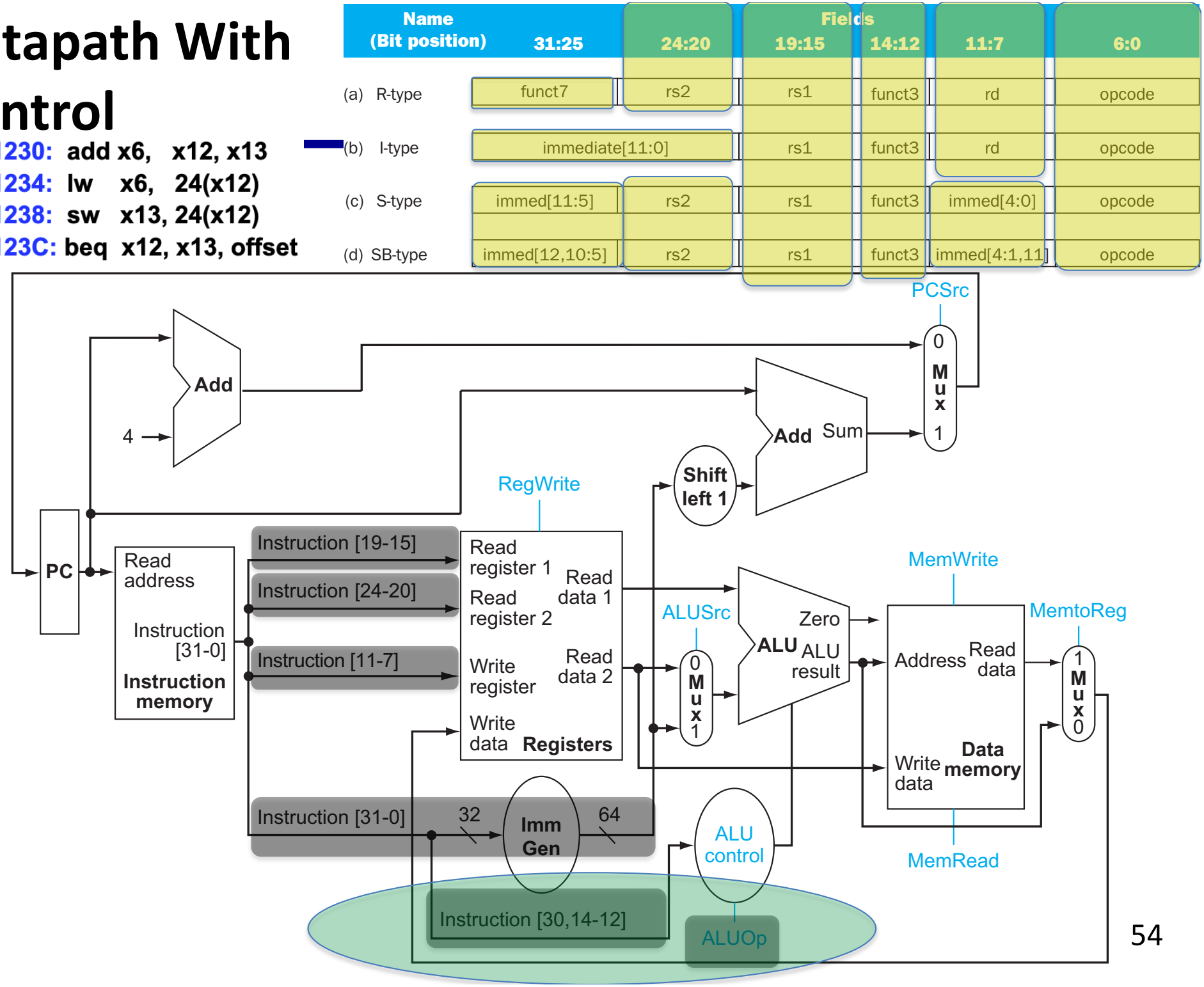
Instruction opcode	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

- The design of those logics can be done with PLA

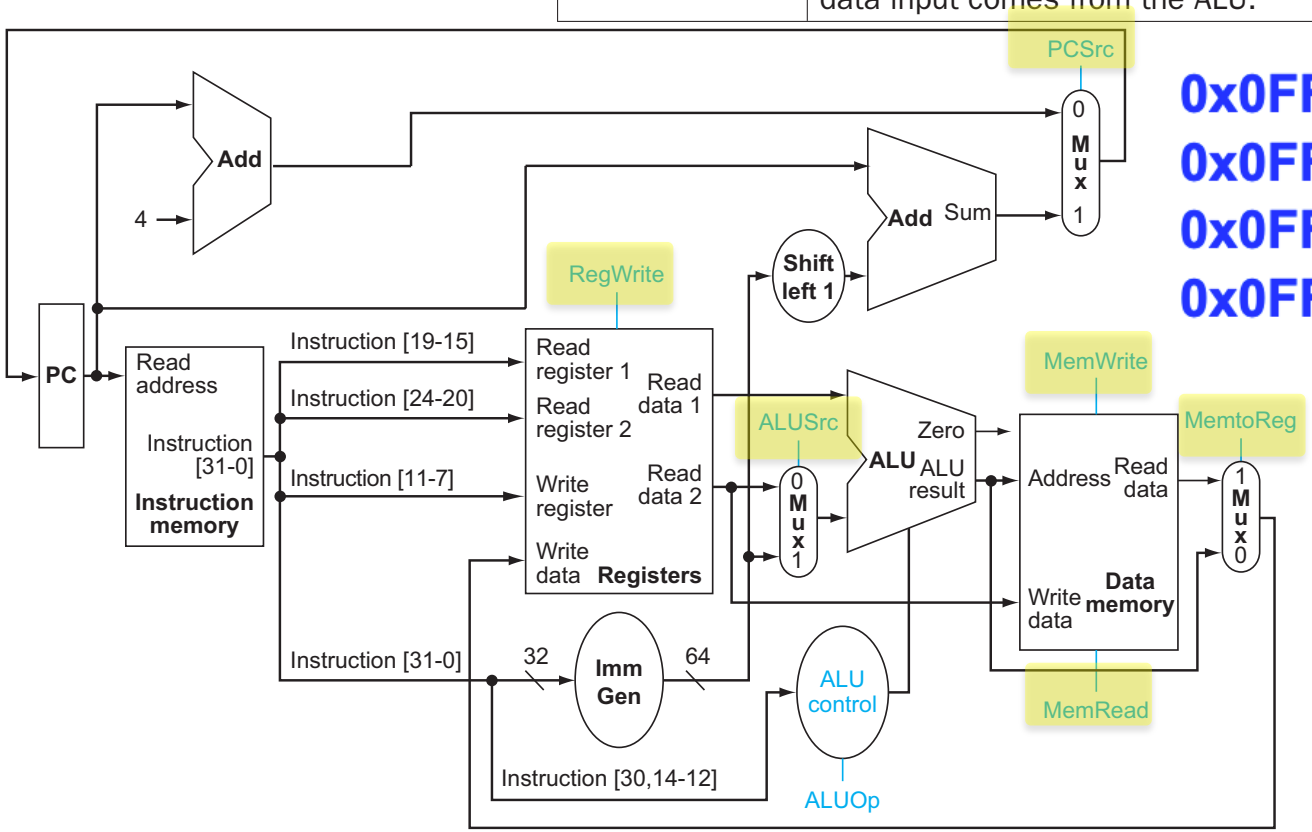
Datapath With Control

0x0FFE1230: add x6, x12, x13
0x0FFE1234: lw x6, 24(x12)
0x0FFE1238: sw x13, 24(x12)
0x0FFE123C: beq x12, x13, offset



Six Control Signals

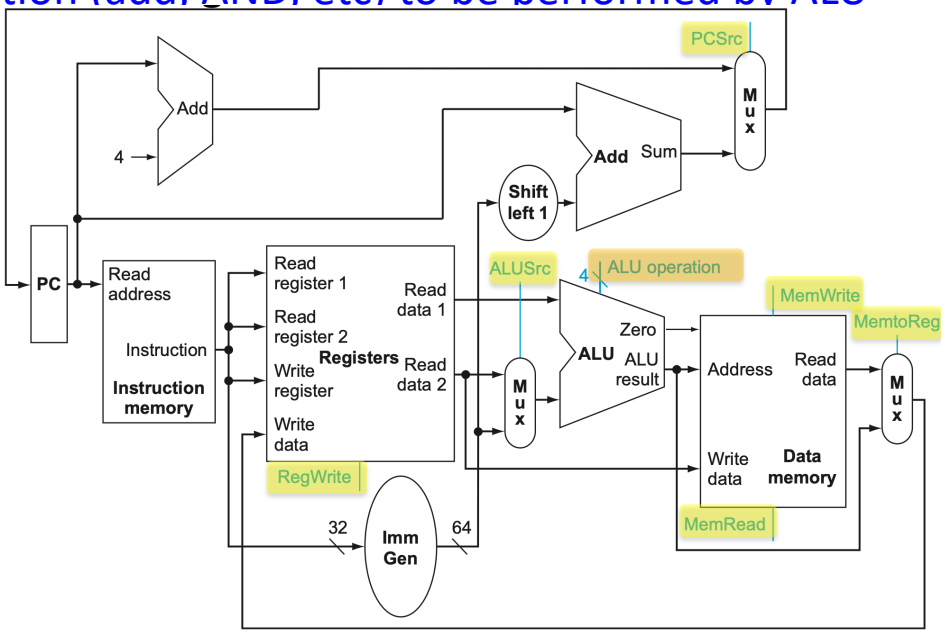
Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.



0x0FFE1230: add x6, x12, x13
0x0FFE1234: lw x6, 24(x12)
0x0FFE1238: sw x13, 24(x12)
0x0FFE123C: beq x12, x13, offset

Control Signals

- 6 1-bit control:
  - PCSrc: Mux input to select PC+4 or PC+offset, for beq instruction to select next instruction
  - ALUSrc: Mux input to select input from rs2 or immediate, for R/I-type ALU and load instr
  - RegWrite: enable signal to enable write to register, for ALU, and load instr (write to register)
  - MemRead: enable signal to enable read from memory, for load instr
  - MemWrite: enable signal to enable write to memory, for store instr
  - MemtoReg: Mux input to select input to write to register from memory or ALU, for ALU and load instr
- 1 4-bit control: ALU operation
  - 2-bit ALUOp: for enabling certain input (Ainvert, Binvert, etc) of the ALU
  - 2-bit ALU Action: AL operation (add, AND, etc) to be performed by ALU

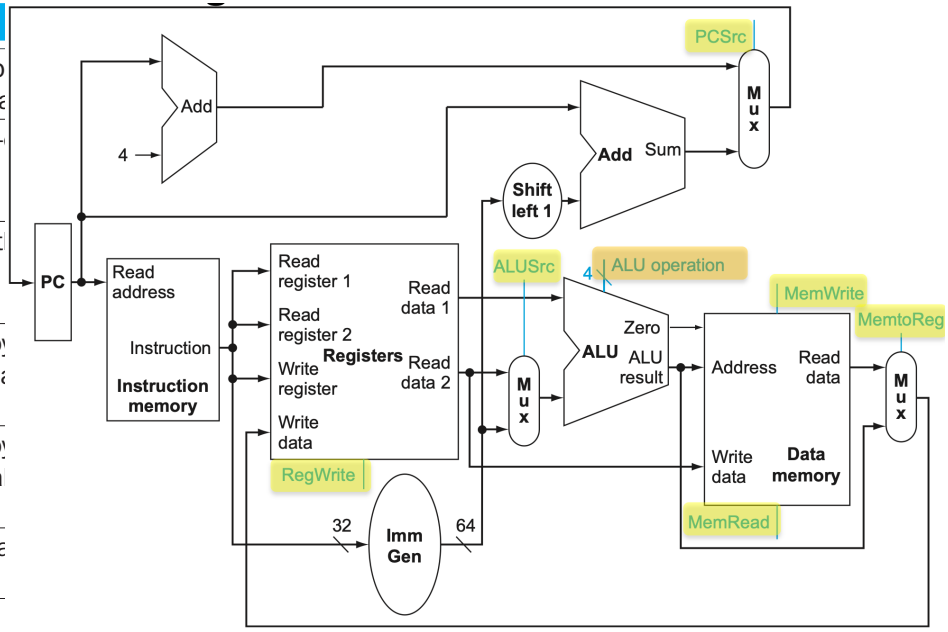


Setting of the 2-Bit ALUOp and the 6 1-bit Controls

- **Are completely determined by the instruction opcode**
  - Check Figure 4.18 of the description

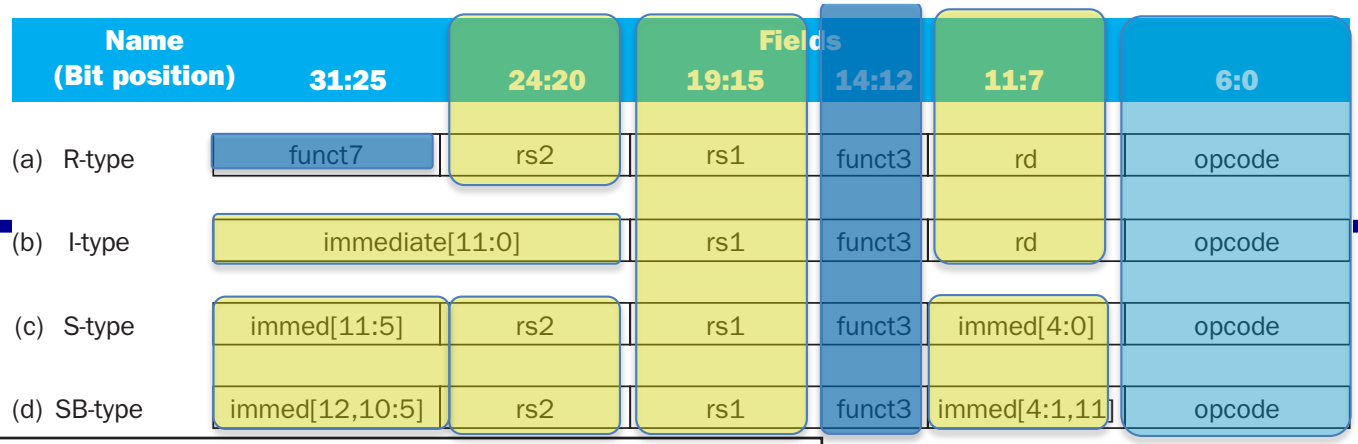
Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

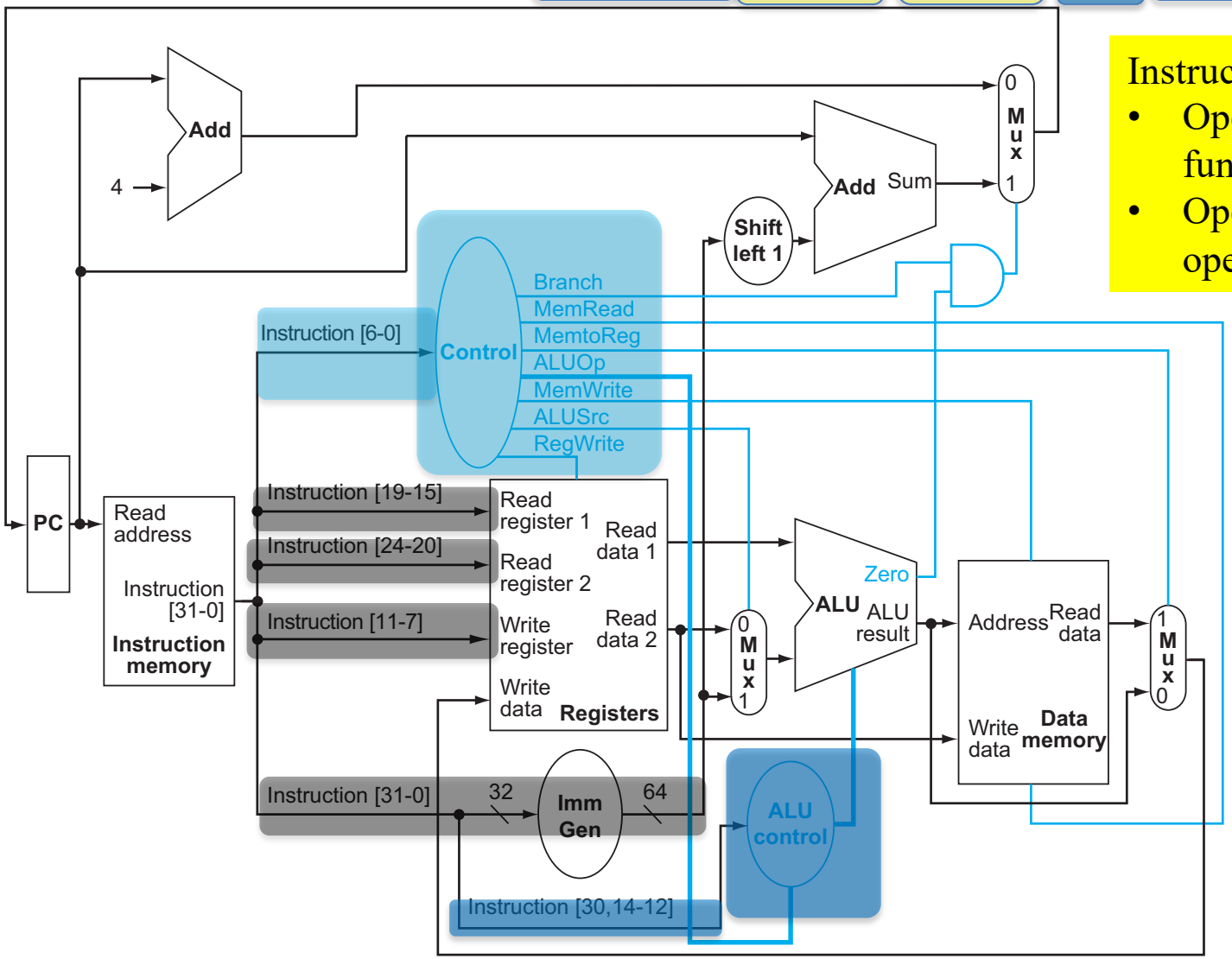
Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register inp written with the value on the Write da
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign- 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of t that computes the branch target.
MemRead	None.	Data memory contents designated by address input are put on the Read da output.
MemWrite	None.	Data memory contents designated by address input are replaced by the va the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write da comes from the data memory.



Datapath With Control

0x0FFE1230: add x1, x2, x3
0x0FFE1234: lw|sw x1, 32(x2)
0x0FFE1238: beq x1, x2, offset





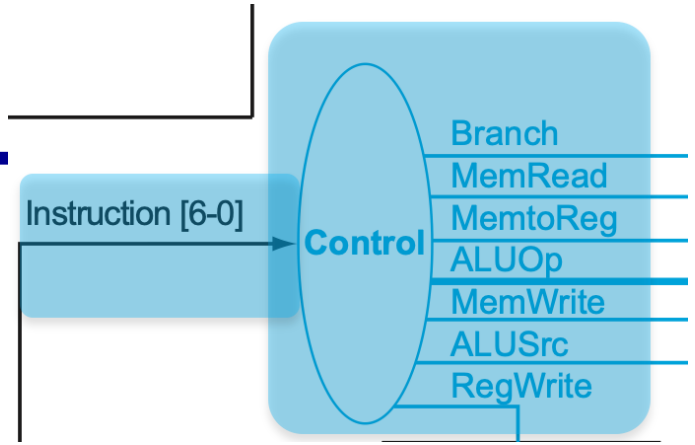
Instruction Word:

- Operation: opcode, func3 and func7) ➔ control logic
- Operands: register or immediate operands ➔ data path

The "Control" logic derives the 2-bit ALUOp and other six 1-bit control, solely based on Instruction[6-0] bits, which is the opcode of an instruction word.

"ALU Control" logic derives the 2-bit ALU action based on Instruction[30,14-12] bits (func3 and func7) and then combines the 2-bit ALU action with the 2-bit ALUOp to create the 4-bit ALU input control.

Truth Table for the Control Logic



- The "Control" logic derives the 2-bit ALUOp and other six 1-bit controls, solely based on Instruction[6-0] bits, which is the opcode of an instruction word.

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

R-, I-, S and SB-Instruction

```
0x0FFE1230:  add x6,   x12, x13
0x0FFE1234:  lw    x6,   24(x12)
0x0FFE1238:  sw    x13, 24(x12)
0x0FFE123C: beq  x12, x13, offset
```

- Study and test goals:
  - Understand the data and control path
  - Given an instruction and the processor diagram, specify the values in EACH datapath and control path
    - Exercise in HW4 and test questions
  - Datapath flow
    - Slide 38-42
  - Control path
    - Slide 46-60
  - Exercise today

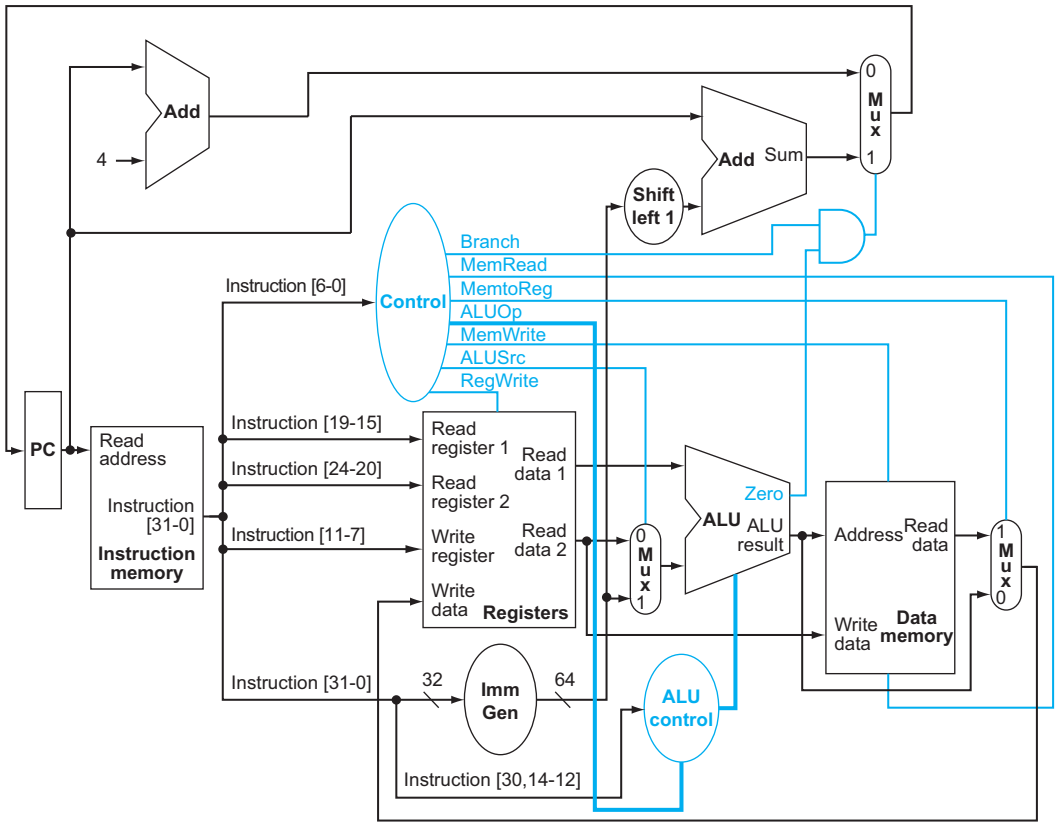
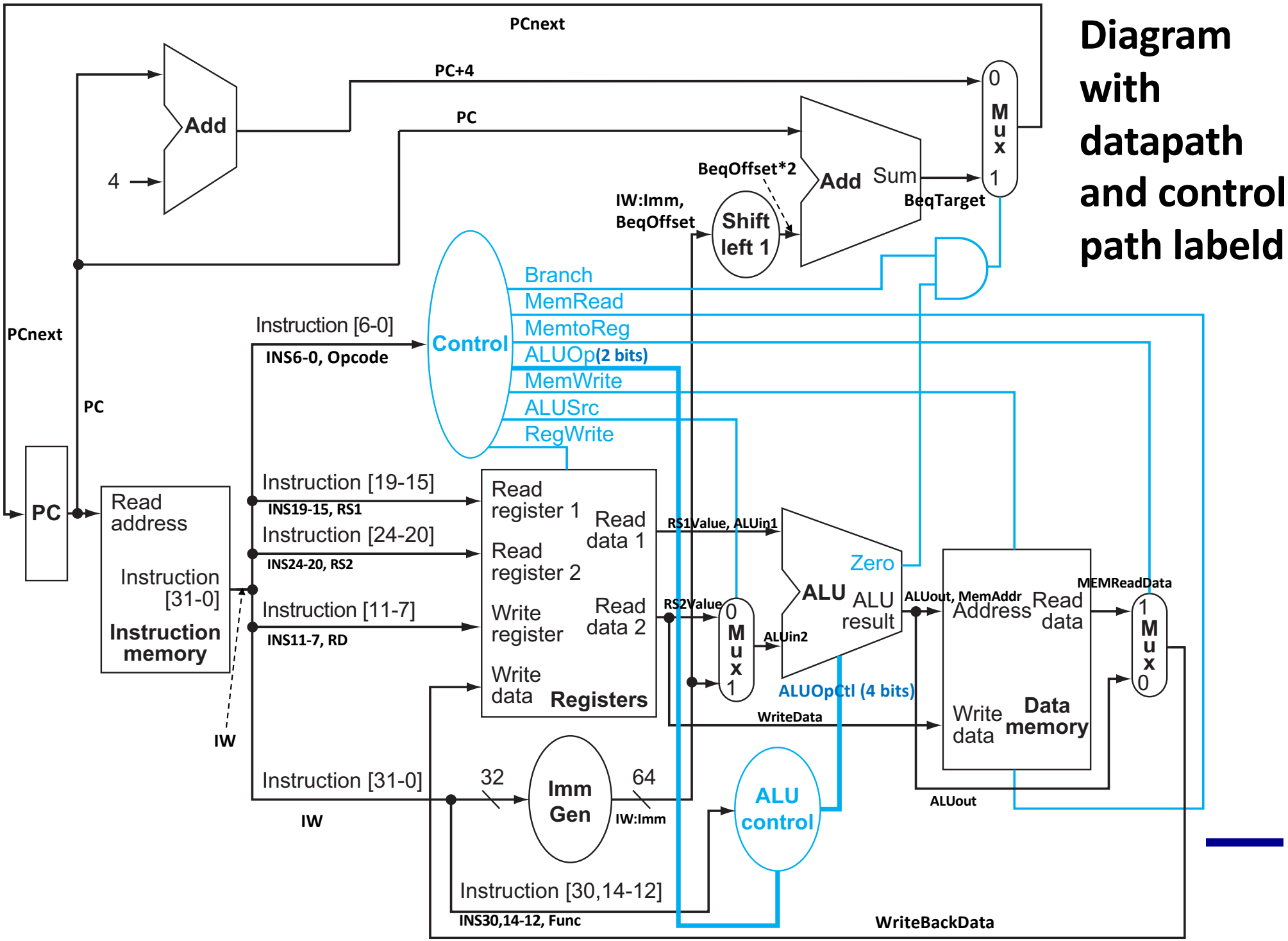


Diagram with datapath and control path labeld

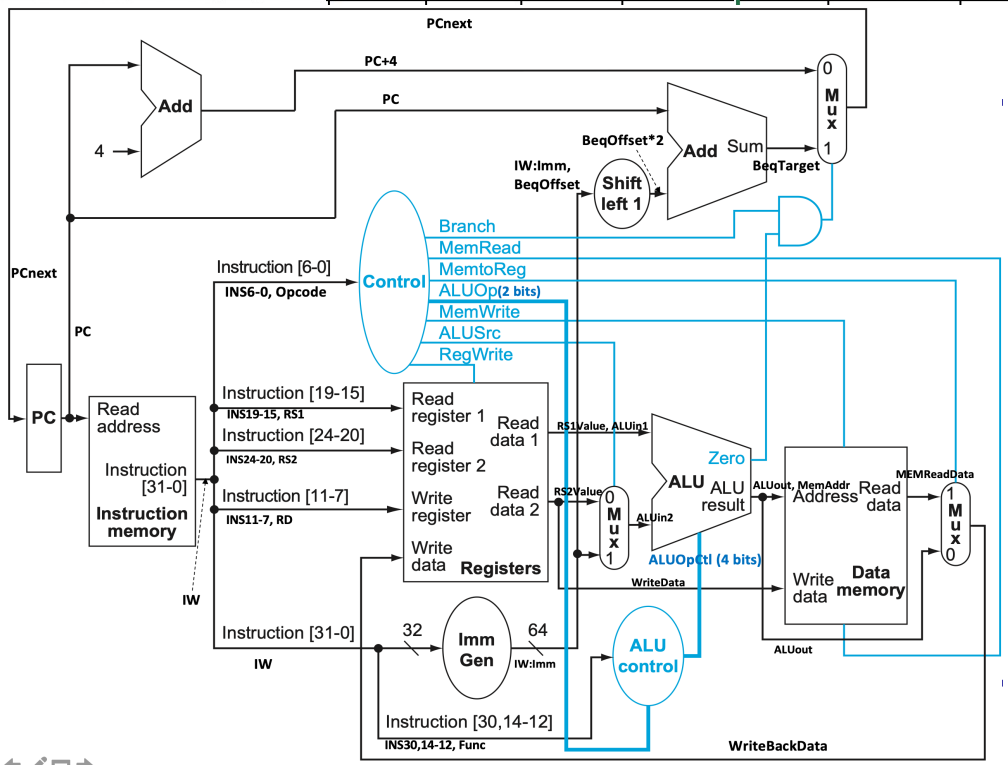


Specifying Values for Each Datapath

Answer sheet: https://passlab.github.io/ITSC3181/HW4/Homework_4_AnswerSheet.xlsx

							Datapath								
Instructions				Instruction Fetch (IF)			Instruction Decoding and Register Fetch (ID/F)								
Address	Instr Word	Instruction	Format (R-	PC	PC+4	IW (Instruct	INS6-0, Opcode	INS19-15, RS1	INS24-20, RS2	INS11-7, RD	INS30,14-12, Func	RS1Value, ALUin1	RS2Value	IW:Imm, BeqOffset	AL
4194304	0x003100b3	ADD x1,x2,x3	R	4194304	4194308	0000 0000 0	011 0011	2	3	1	0, 000	4096	2	x	
4194308	0x02012083	LW x1,32(x2)	load	4194308	4194312	0000 0010 0	000 0011	2	x	1	0, 001	4096	x	32	
4194312	0x02112023	SW x1,32(x2)	store	4194312	4194316	0000 0010 0	010 0011	2	1	x	0, 001	4096	1024	32	
4194316	0x00208a63	beq x1,x2,10	beq	4194316	4194320	0000 0000 0	110 0011	1	2	x	0, 000	1024	4096	10	
4194320	0x00100193	ADDI x3,x0,1	I	4194320	4194324	0000 0000 0	001 0011	0	x	3	0, 000	0	x	1	

							Datapath								
Instructions				Instruction Fetch (IF)			Execution and Branch Target Calculation (EXE)					Memory Read/Write Access (MEM)			Write Back(WB)
Address	Instr Word	Instruction	Format (R-	PC	PC+4	IW (Instruct	ALUin2	ALUOut	BeqOffset*2	BeqTarget	PCNext	MemAddress	WriteData	MEMReadData	WriteBackData
4194304	0x003100b3	ADD x1,x2,x3	R	4194304	4194308	0000 0000 0	2	4098	x	x	4194308	x	x	x	4098
4194308	0x02012083	LW x1,32(x2)	load	4194308	4194312	0000 0010 0	32	4128	x	x	4194312	4128	x	8	8
4194312	0x02112023	SW x1,32(x2)	store	4194312	4194316	0000 0010 0	32	4128	x	x	4194316	4128	1024	x	x
4194316	0x00208a63	beq x1,x2,10	beq	4194316	4194320	0000 0000 0	4096	-3072	20	4194336	4194320	x	x	x	x
4194320	0x00100193	ADDI x3,x0,1	I	4194320	4194324	0000 0000 0	1	1	x	x	4E+07	x	x	x	1



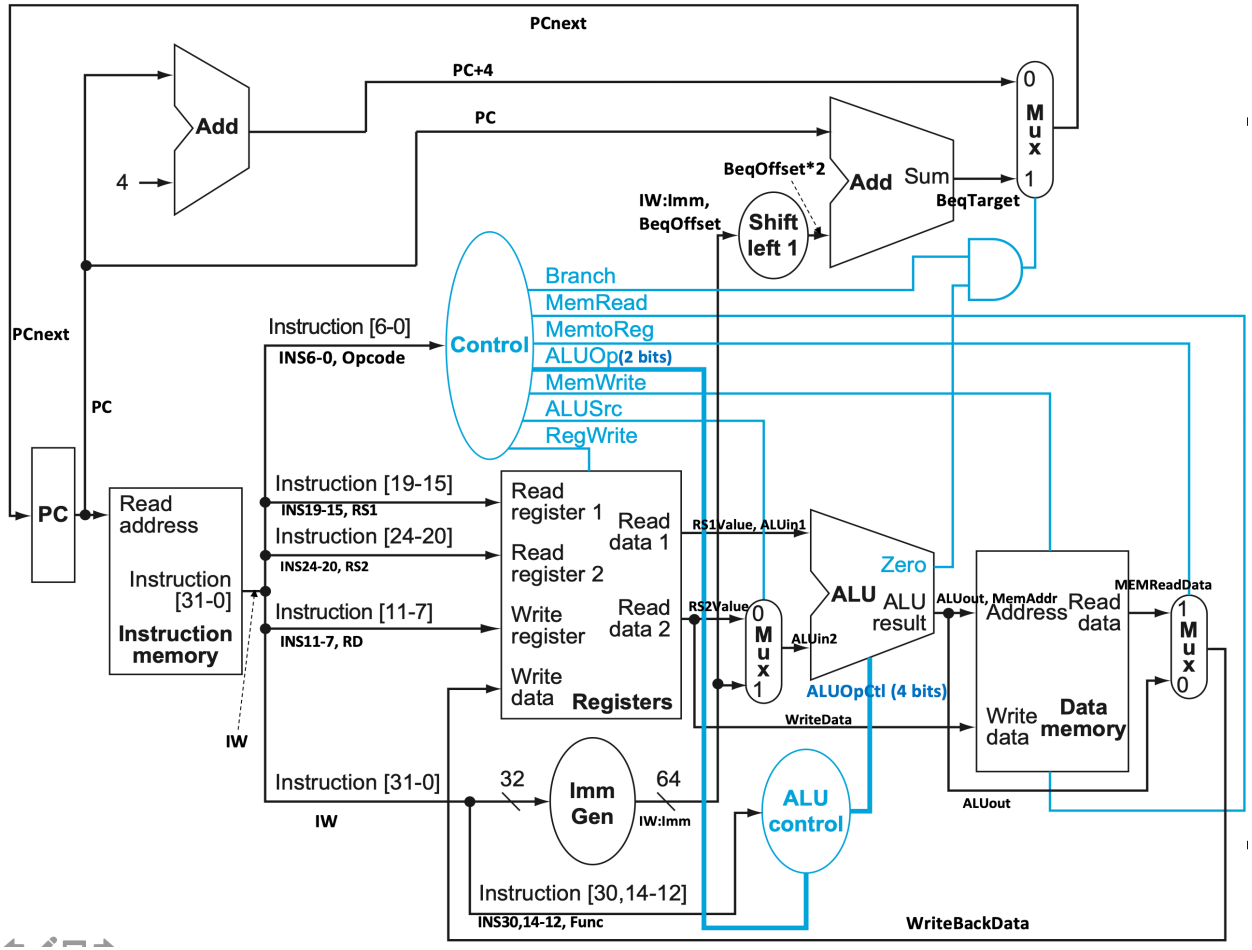
Register Num	Value
0	0
1	1024
2	4096
3	2
4	254
5	4100
6	0
7	1
8	2
9	3
10	1028
11	4
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	

Mem Address	Value
...	
1024	
1028	3
1032	
1036	
1040	
...	
4096	0
4100	1
4104	2
4108	3
4112	4
4116	5
4120	6
4124	7
4128	8
4132	9
...	

Specifying Values for Each Control

Answer sheet: https://passlab.github.io/ITSC3181/HW4/Homework_4_AnswerSheet.xlsx

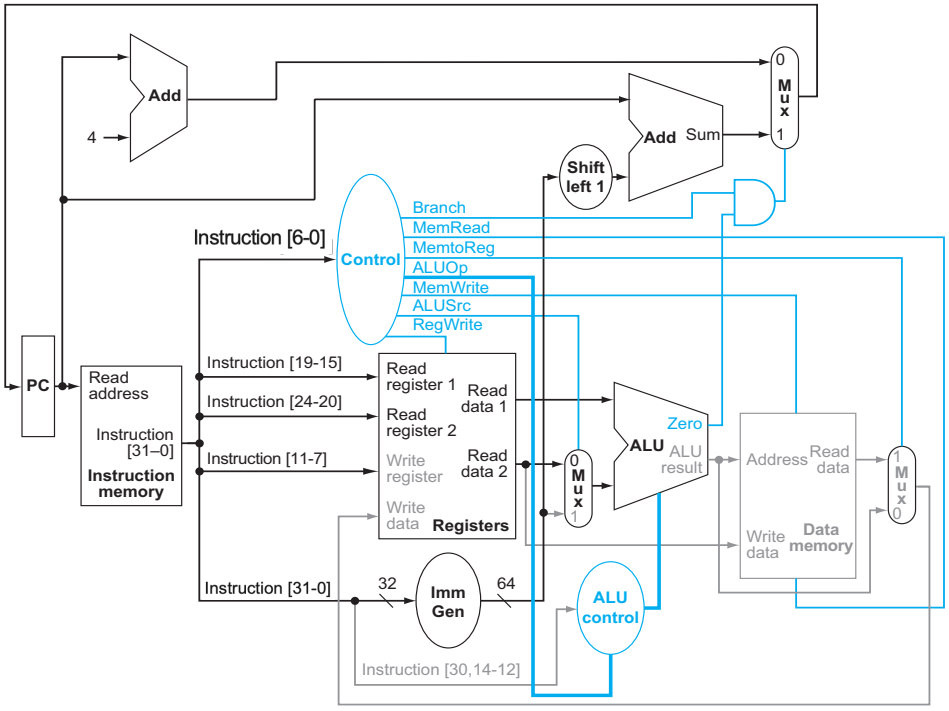
				Control Signal										
Address	Instr Word	Instruction	Format (R-	Branch	MemRead	MemtoReg	ALUOp(2 bits)	MemWrite	ALUSrc	RegWrite	ALU Action	ALUOpCtl (4 bits)	Zero
4194304	0x003100b3	ADD x1,x2,x3	R	0	0	0	10	0	0	1	add	0010	0
4194308	0x02012083	LW x1,32(x2)	load	0	1	1	00	0	1	1	add	0010	0
4194312	0x02112023	SW x1,32(x2)	store	0	0	0	00	1	1	0	add	0010	0
4194316	0x00208a63	beq x1,x2,10	beq	1	0	0	01	0	0	0	sub	0110	0
4194320	0x00100193	ADDI x3,x0,1	I	0	0	0	10	0	1	1	add	0010	0
4194324	0x10007213	ANDI x4,x0,256	I										



Homework 4

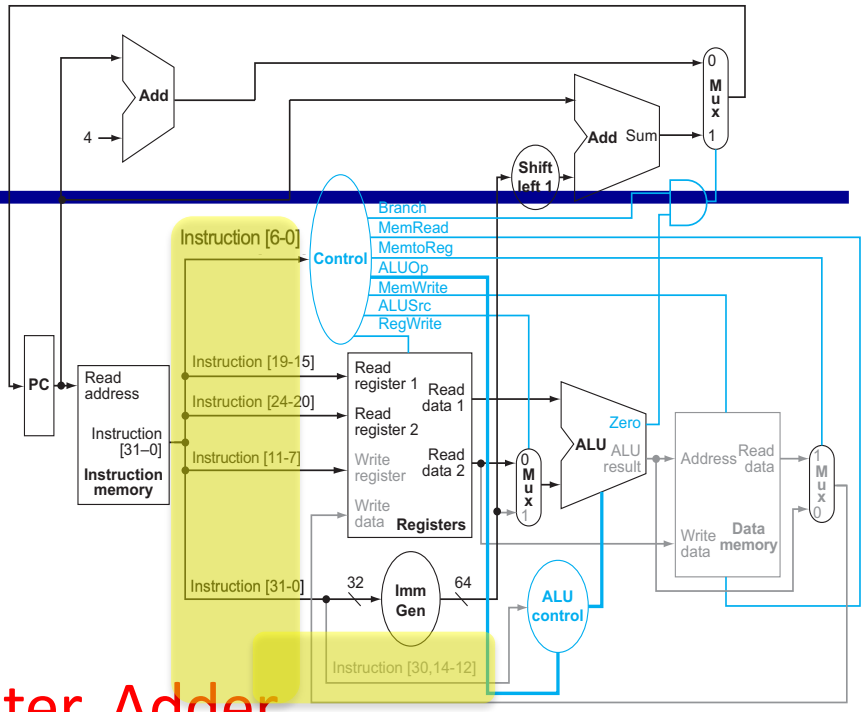
https://passlab.github.io/ITSC3181/HW4/Homework_4.pdf

- Homework 4
  - Work out the datapath and control for I-type AL instruction
    - Addi, ANDi
  - Fill in the sheet for the datapath and control value for other instructions
  - Pipeline execution diagram (following sections)



Lab 11 and 12

https://passlab.github.io/ITSC3181/notes/Lab_11_12_SingleCycleCPU.pdf



- Create the processor diagram using Digital
  - Close to realistic design, but not need to make it work
  - We have most components:
    - Instr/Data Mem, ALU, Mux, Register, Adder
  - We need to add
    - PC: a 32-bit register
    - Control, ALU control, Imm Gen, and Shift left 1:
      - Create fake logics that have the required input and outputs and use them
      - Make sure the bitwidth of the input and output are correctly set
    - Decoder: to split 32-bit instruction word into instruction[6-0], instruction[19-15], instruction[24-20], instruction[11-7], instruction[30], and instruction[14-12],

Performance Issues

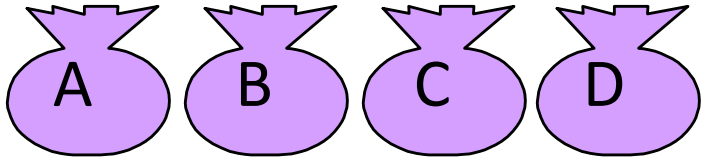
- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Chapter 4: The Processor

- **Lecture**
  - **4.1 Introduction**
  - **4.2 Logic Design Conventions**
  - **4.3 Building a Datapath**
- **Lecture**
  - **4.4 A Simple Implementation Scheme**
- ☛ **Lecture**
  - **4.5 An Overview of Pipelining**
- **Lecture (Pipeline implementation), will not be covered!**
  - 4.6 Pipelined Datapath and Control
  - 4.7 Data Hazards: Forwarding versus Stalling
  - 4.8 Control Hazards
  - ~~4.9 Exceptions~~
  - ~~4.13 Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations~~
- **Lecture (Advanced pipeline techniques and real-world CPU examples)**
  - **4.10 Parallelism via Instructions**
  - **4.11 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines**
  - 4.12 Going Faster: Instruction-Level Parallelism and Matrix Multiply
  - ~~4.14 Fallacies and Pitfalls~~
  - **4.15 Concluding Remarks**

Pipelining Analogy

- Laundry Example

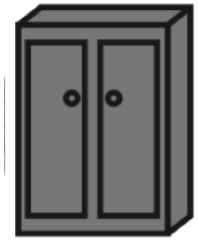


- Ann, Brian, Cathy, Dave
  each have one load of clothes
  to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - ”Folder” takes 30 minutes
  - “Putter” takes 30 minutes

30 minutes

30 minutes

30 minutes

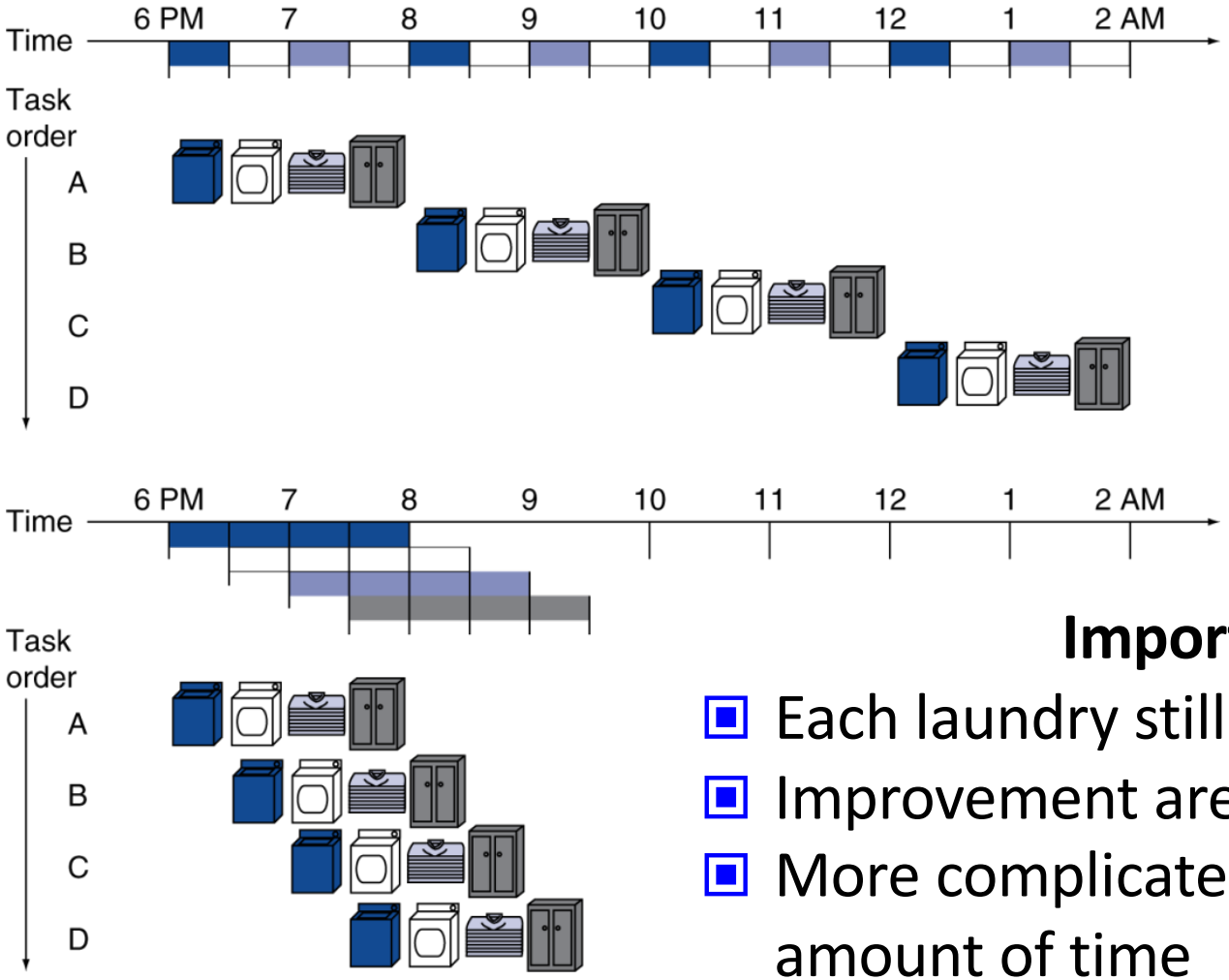


30 minutes

- One load: 120 minutes

Pipelining: Its Natural!

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

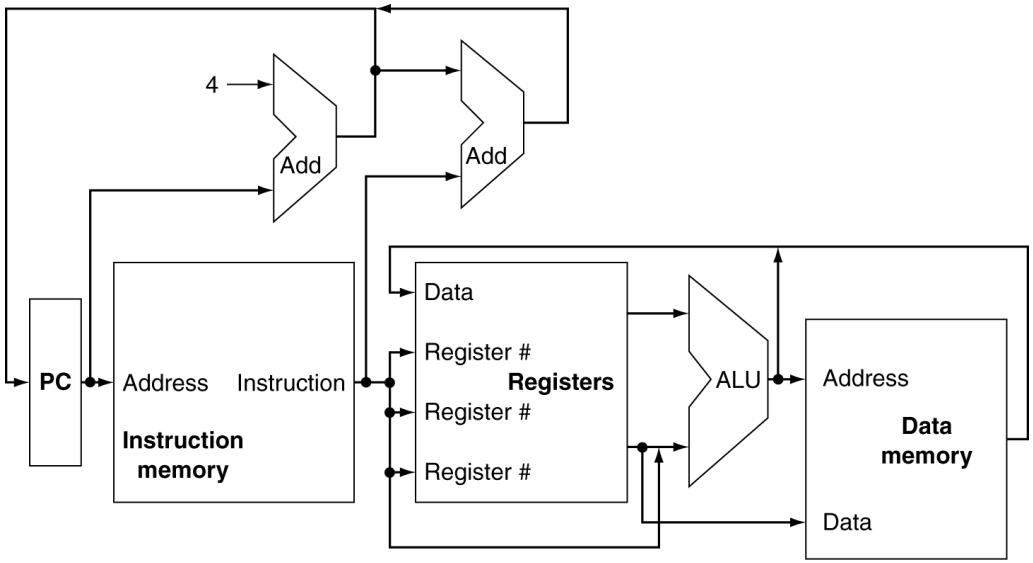
Important to note

- Each laundry still takes 120 minutes.
- Improvement are for 4 load throughput.
- More complicated if stages take different amount of time

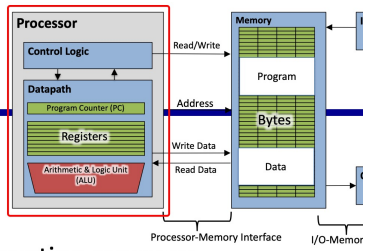
RISC-V Pipeline

Five stages, one step per stage

1. IF: Instruction Fetch from memory
2. ID: Instruction Decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result Back to register

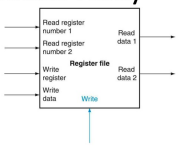


Instruction Execution

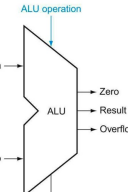


0x0FFE1230: add x1, x2, x3
0x0FFE1234: lw|sw x1, 32(x2)
0x0FFE1238: beq x1, x2, offset

- Processor loads an instruction word from instruction memory
  - PC → register to store address to access instruction memory to fetch instruction
- The instruction word is decoded so source operands (register numbers) are know, and then registers are read to have source operand values ready
  - Register numbers → register file, read registers
    - x2 and x3 for add; x2 and x1 for lw|sw, x1 and x2 for beq

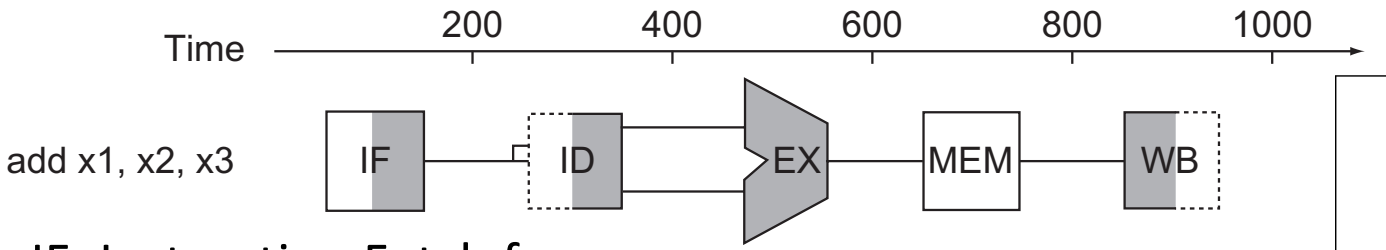


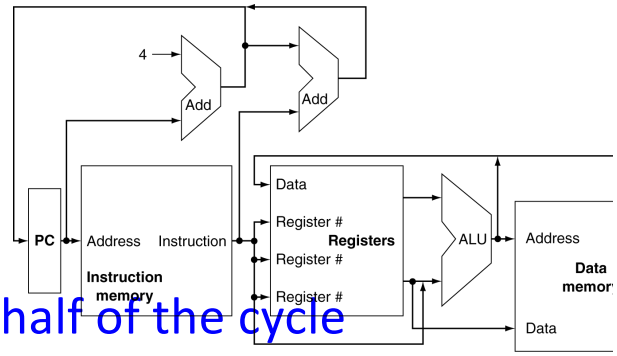
- Use ALU to calculate
  - Depending on instruction class
    - Arithmetic result: x2 + x3
    - Memory address for load/store: 32+x2, add operation
    - Branch condition: x1 ?= x2 ➔x1-x2 and check result is 0 or not



- LW|SW: access data memory: load/store from/to x1
- Branch: PC ← target address or PC + 4: pc = pc+offset*2 if branch is take
- Write result to register
  - Arithmetic (add): write result (x1) back to the register file
  - Load: write x1 to register

Graphical Representation of Instruction Pipeline



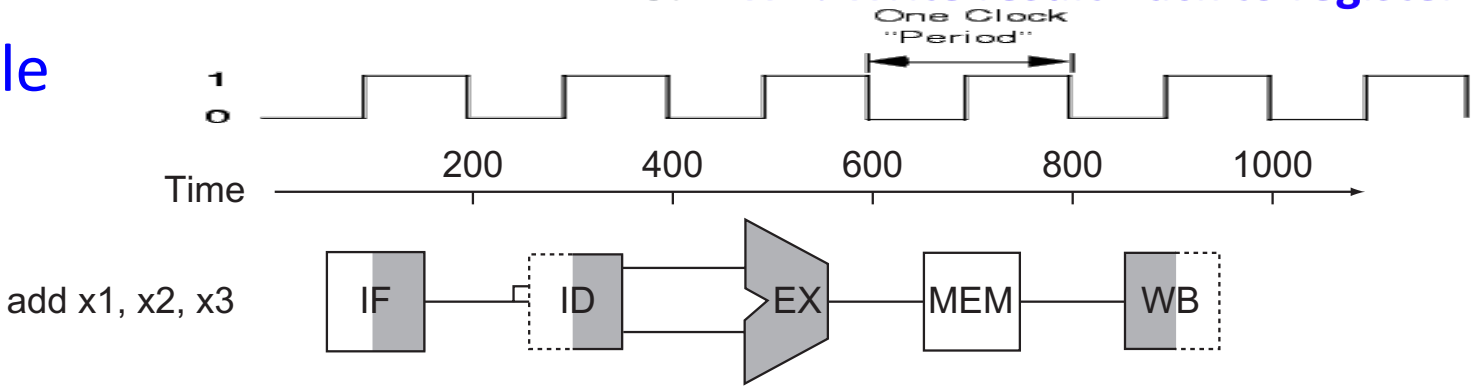


- IF: Instruction Fetch from memory
  - Box representing instruction memory
  - Right-half shade representing usage of IM at the second half of the cycle
- ID: Instruction Decode & register read
  - Box representing register
  - Right-half shade representing usage (read) of Register at the second half of the cycle
- EX: Execute operation or calculate address
  - Shade representing usage
- MEM: Access memory operand (only for load/store)
  - White background representing NOT used by add instruction in this example
- WB: Write result Back to register (only for load and AL instructions)
  - Box representing register
  - Left-half shade representing write to register at the first half of the cycle

Classic 5-Stage Pipeline for a RISC

- In each cycle, hardware initiates a new instruction and executes some part of five different instructions:
  - Simple

1. **IF: Instruction Fetch from memory**
2. **ID: Instruction Decode & register read**
3. **EX: Execute operation or calculate address**
4. **MEM: Access memory operand**
5. **WB: Write result Back to register**



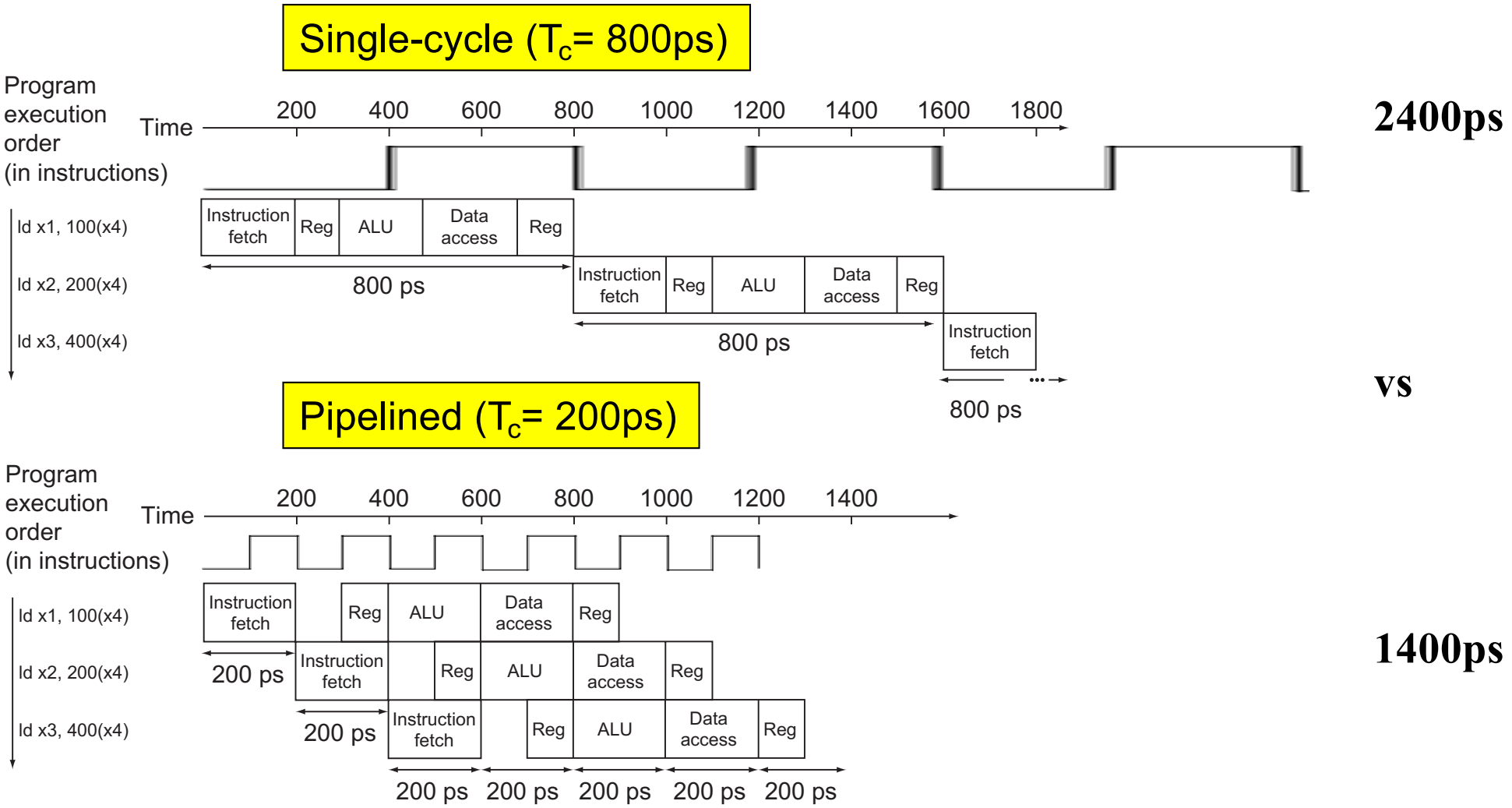
	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	**IF**	**ID**	**EX**	**MEM**	**WB**				
Instruction *i+1*		**IF**	**ID**	**EX**	**MEM**	**WB**			
Instruction *i+2*			**IF**	**ID**	**EX**	**MEM**	**WB**		
Instruction *i+3*				**IF**	**ID**	**EX**	**MEM**	**WB**	
Instruction *i+4*					**IF**	**ID**	**EX**	**MEM**	**WB**

Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (ld)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Pipeline Performance



- For large number of instructions, say 1M, the speedup will be
  - ~= 800ps/200ps = 4

Pipeline Speedup

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i+1*		IF	ID	EX	MEM	WB			
Instruction *i+2*			IF	ID	EX	MEM	WB		
Instruction *i+3*				IF	ID	EX	MEM	WB	
Instruction *i+4*					IF	ID	EX	MEM	WB

- **Execute billions instructions, so throughput is what matters.**
- **Pipelining doesn't help latency of single instruction**
- **Potential speedup = number pipeline stages;**

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

- Unbalanced lengths of pipeline stages reduces speedup;

Pipelining and ISA Design

- RISC ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Hazards

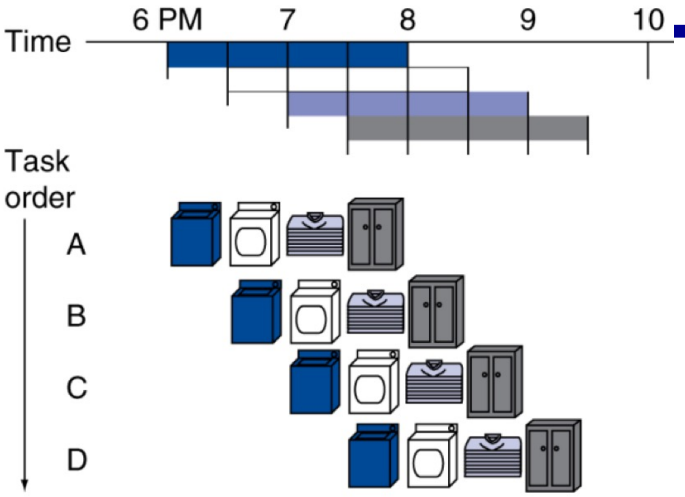
	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i+1*		IF	ID	EX	MEM	WB			
Instruction *i+2*			IF	ID	EX	MEM	WB		
Instruction *i+3*				IF	ID	EX	MEM	WB	
Instruction *i+4*					IF	ID	EX	MEM	WB

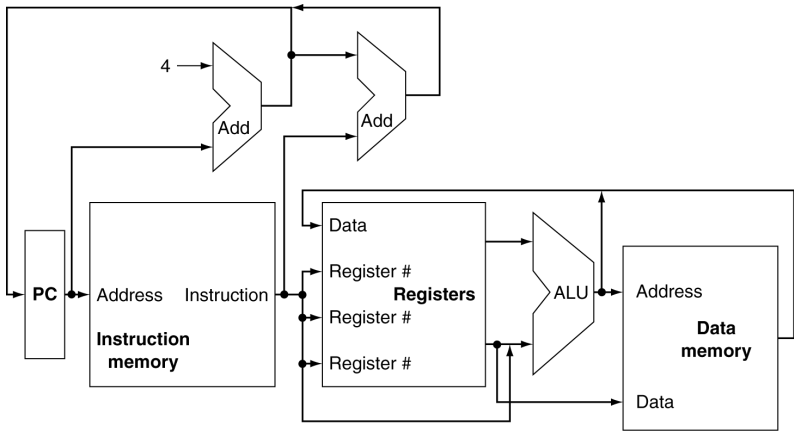
- Situations that prevent starting the next instruction in the next cycle

- **Structure hazards**
  - **A required resource is busy**
- **Data hazard**
  - **Need to wait for previous instruction to complete its data read/write**
- **Control hazard because of branch or jump**
  - **Deciding on control action depends on previous instruction**

Structure Hazards

- Conflict for use of a resource
  - Find a situation in laundry example?
- In RISC-V pipeline if with a single memory → IF and WB conflict
  - Load/store requires mem access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

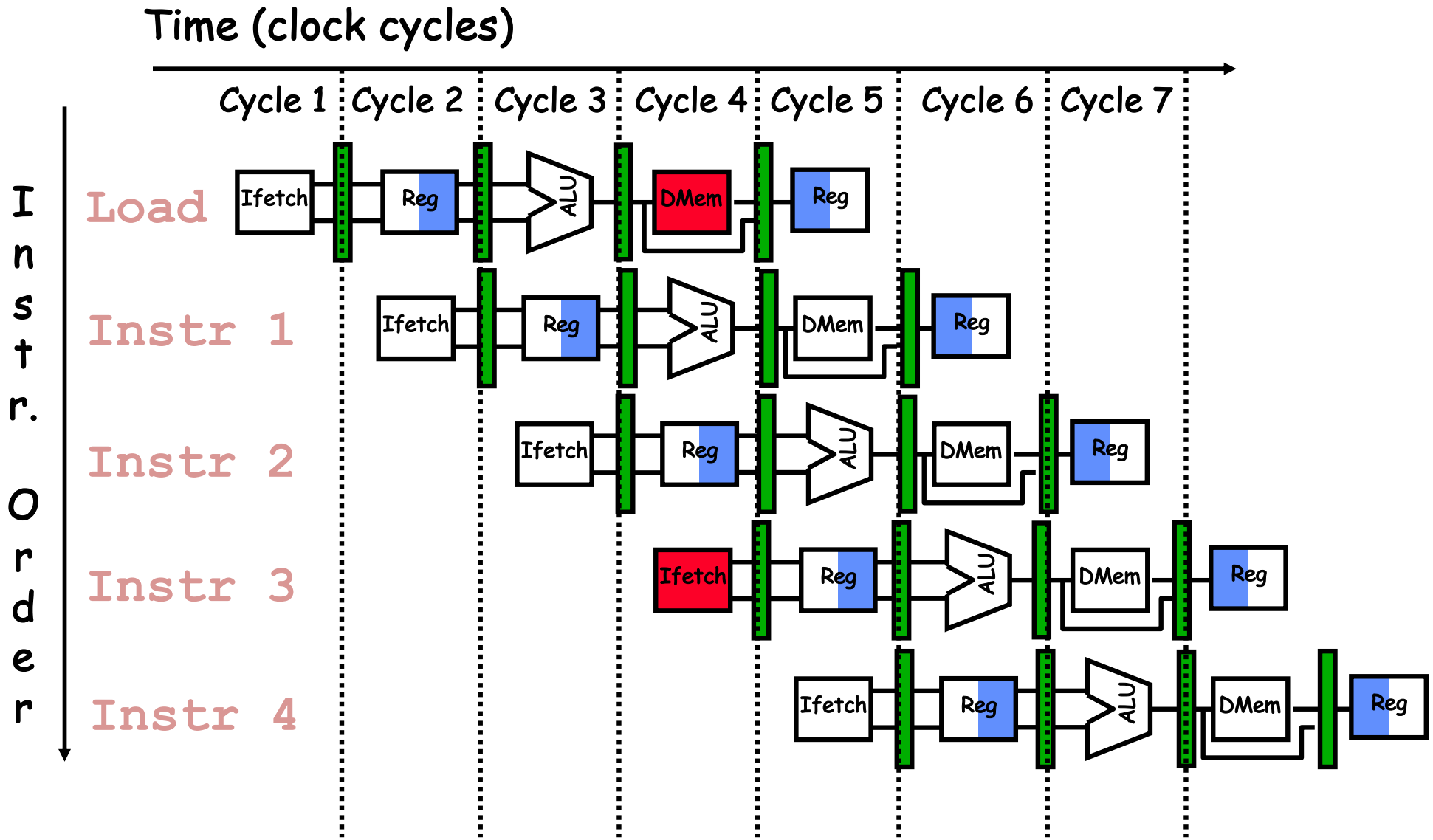




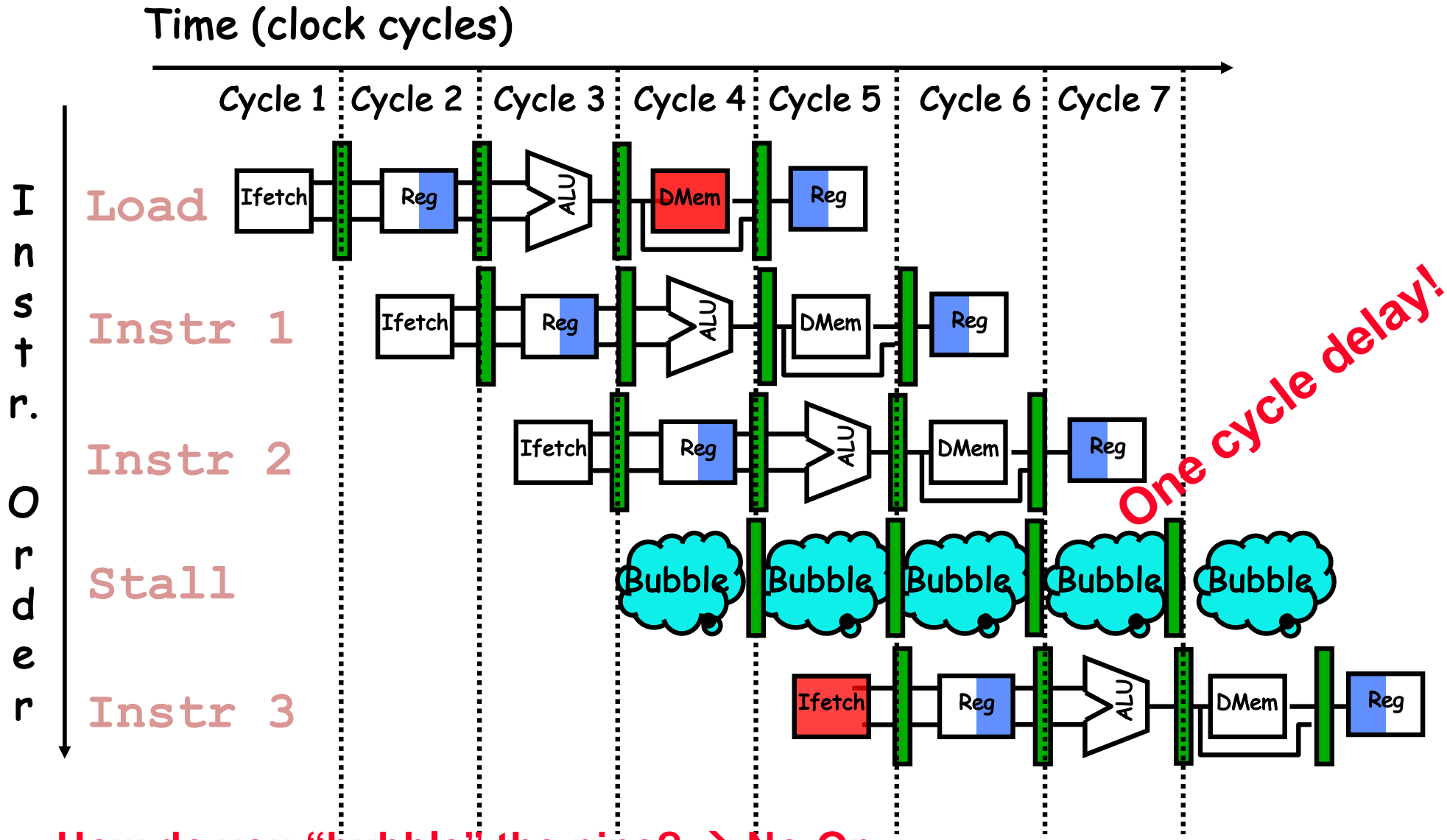
Load or store

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i+1*		IF	ID	EX	MEM	WB			
Instruction *i+2*			IF	ID	EX	MEM	WB		
Instruction *i+3*				IF	ID	EX	MEM	WB	
Instruction *i+3*					IF	ID	EX	MEM	WB

One Memory Port→Structural Hazards



One Memory Port/Structural Hazards



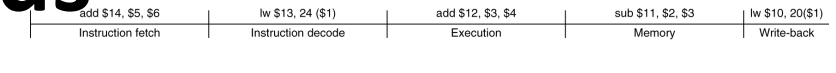
How do you "bubble" the pipe? → No-Op

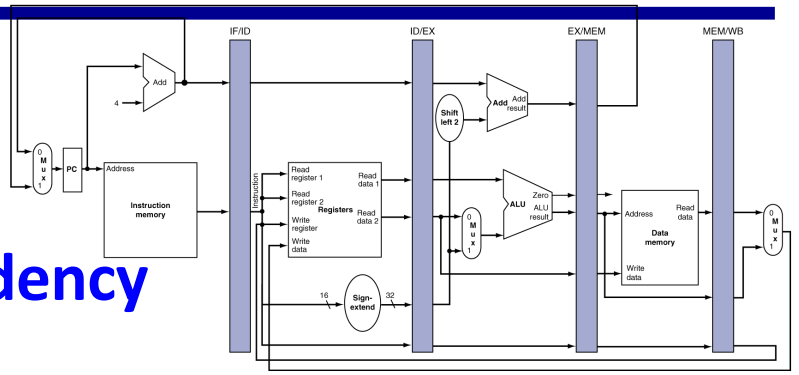
Summary of Structure Hazard

- To address structure hazard, have separate memories for instructions and data
- However, it will increase cost
  - E.g.: pipelining function units or duplicated resources is a high cost;

† If the structure hazard is rare, it may not be worth the cost to avoid it.

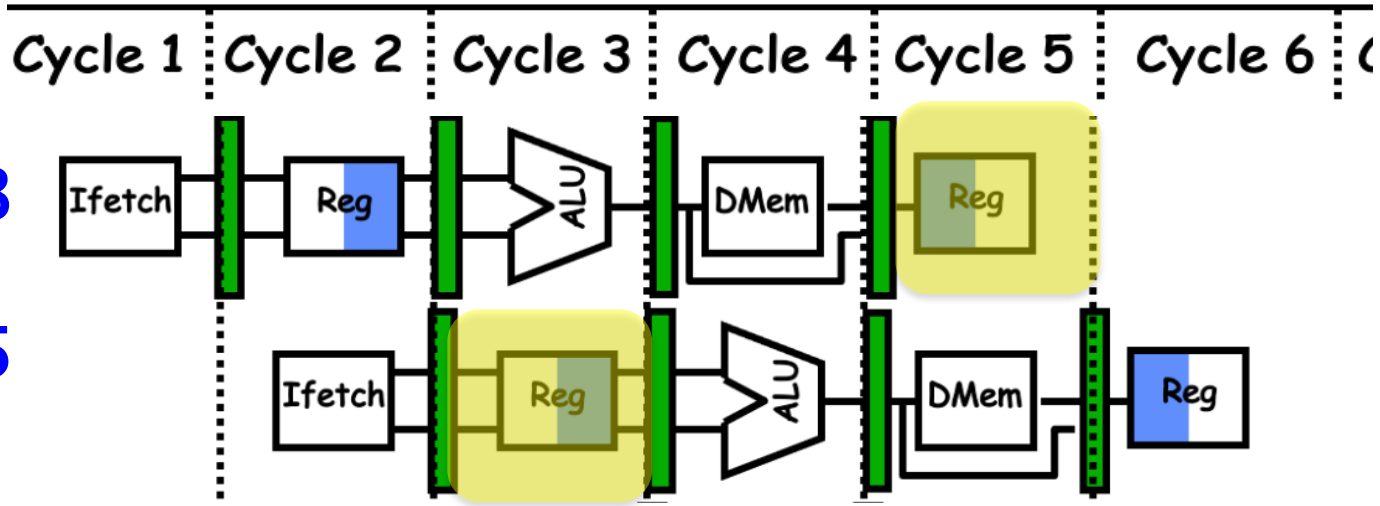
Data Hazards





- An instruction needs data produced by a previous instruction
  - **Read-After-Write (RAW) data dependency**

```
add  x1, x2, x3
sub  x4, x1, x5
```

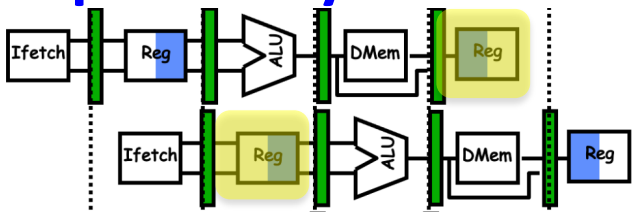


  - **Sub would read old value of x1 at cycle 3**

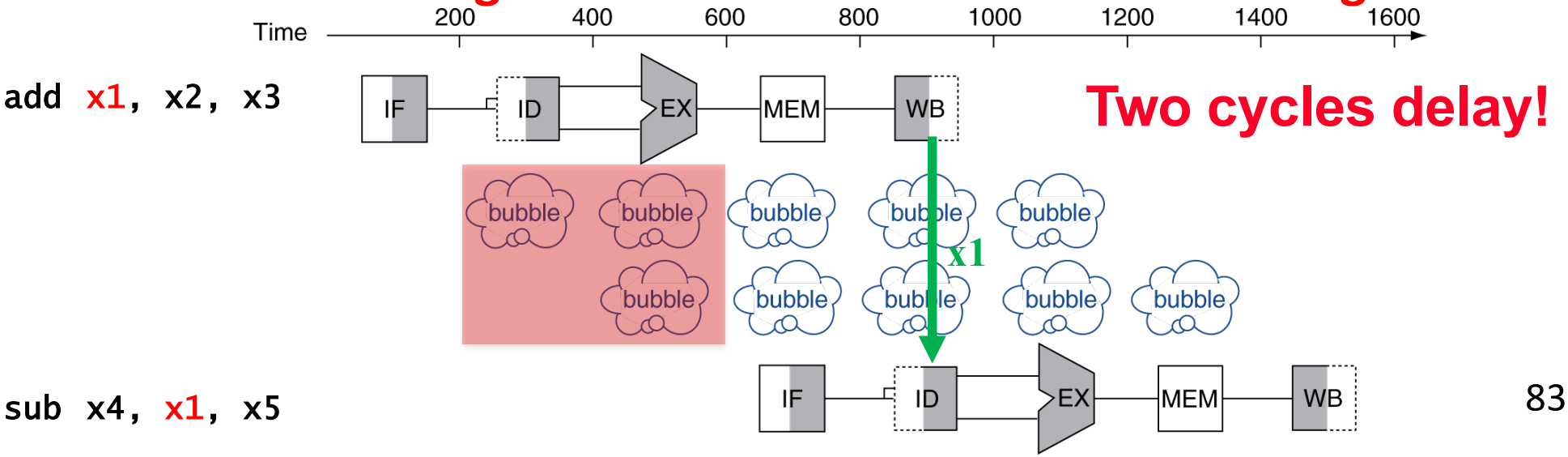
Data Hazards and Solution #1: Interlocking

- An instruction needs data produced by a previous instruction
  - **Read-After-Write (RAW) data dependency**

```
add  x1, x2, x3
sub  x4, x1, x5
```

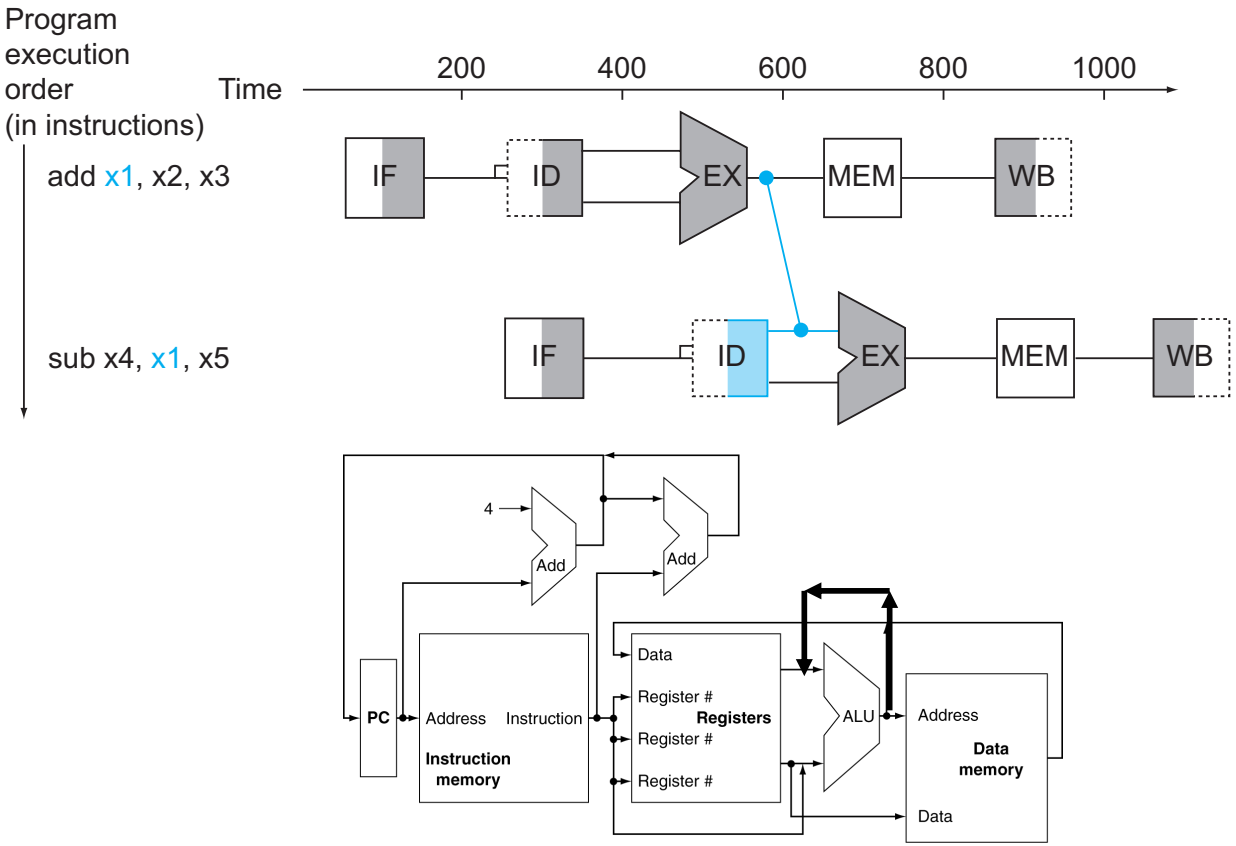


- Interlock: Hardware detect their dependency, and
  - Insert no-op instructions, e.g. “add $0,$0,$0”, as bubble
  - **Waste 400: two instructions in between since sub needs to wait for two stages for add to write the result x1 to register**



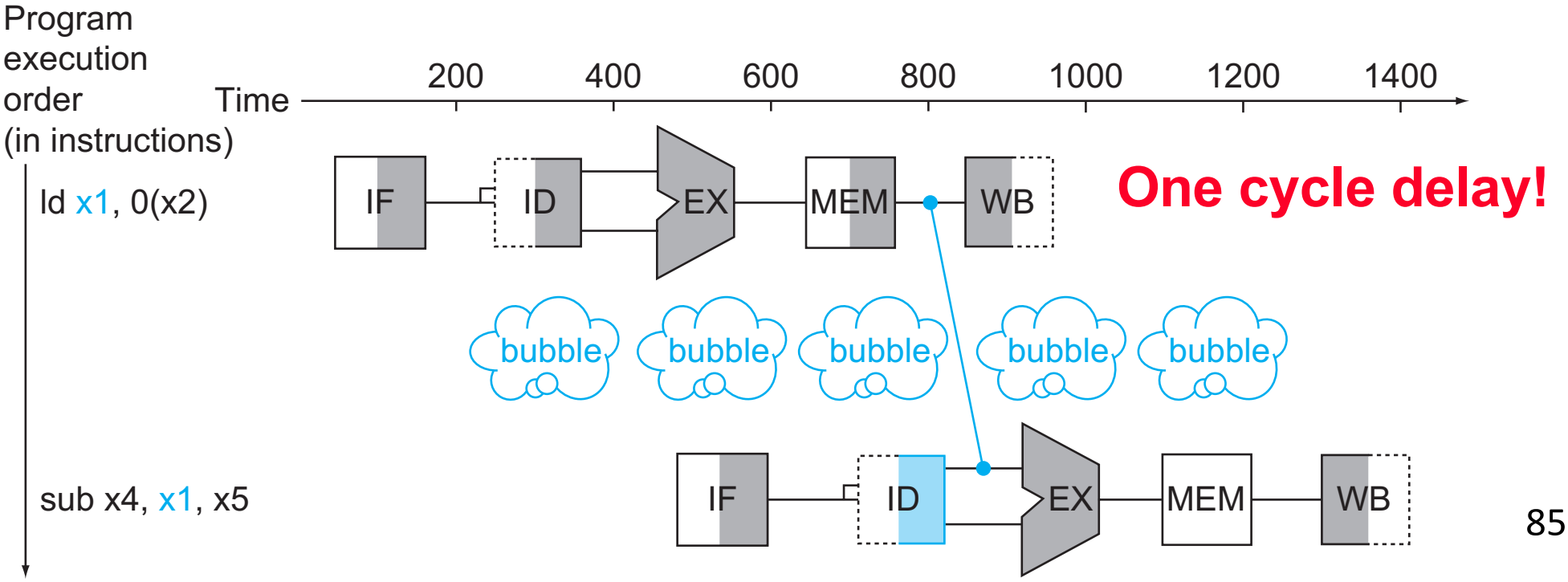
Solution #2: Forwarding (aka Bypassing)

- **Use result right after when it is computed instead of waiting for it to be stored in a register**
  - **add produces the result at the end of its EXE stage**
  - **sub uses the result at the beginning of its EXE stage, which is right after the cycle for add's EXE**
  - Requires extra connections in the datapath



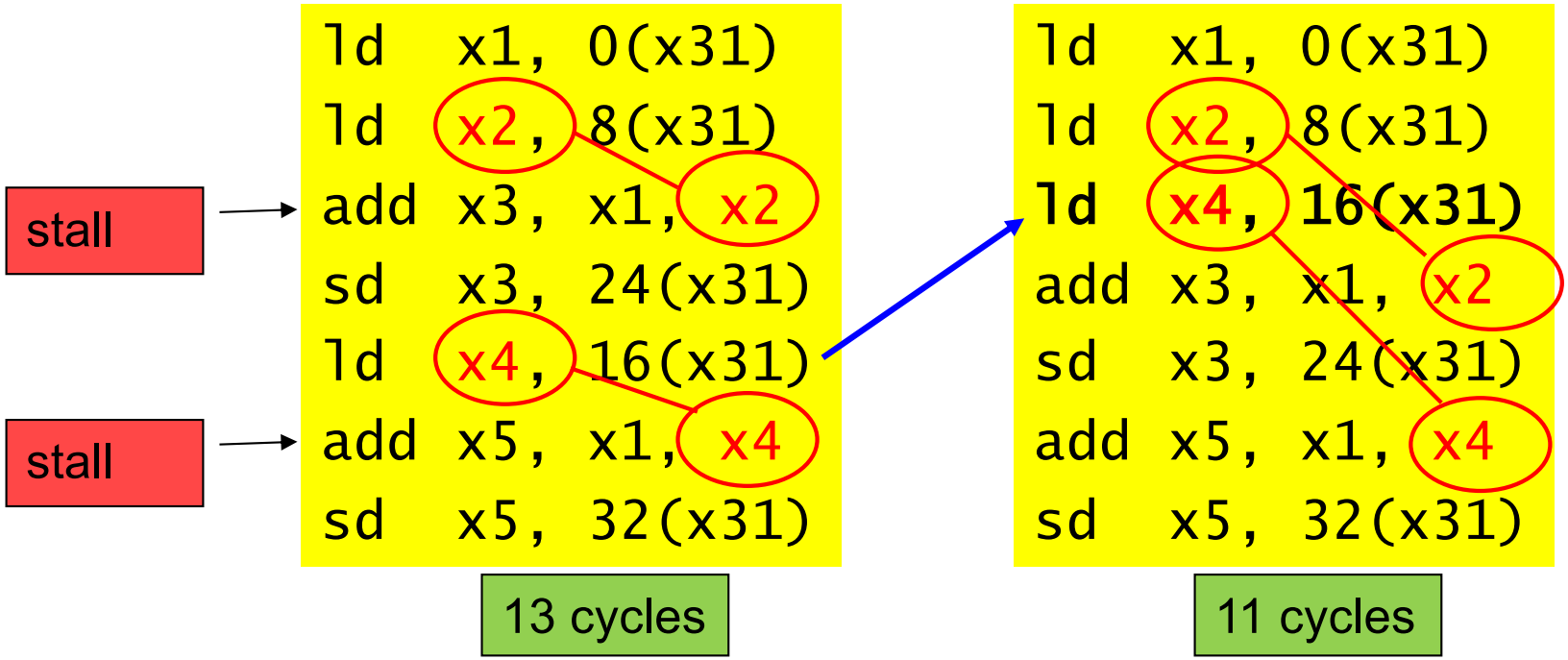
Load-Use Data Hazard

- Load produce the results after the MEM stage
  - Sub use the result at the beginning of the EXE stage, which is in the same cycle as load's MEM, thus, not possible to forward
- Can't avoid stalls by forwarding for load-use
  - If value not computed when needed
  - Can't forward backward in time!



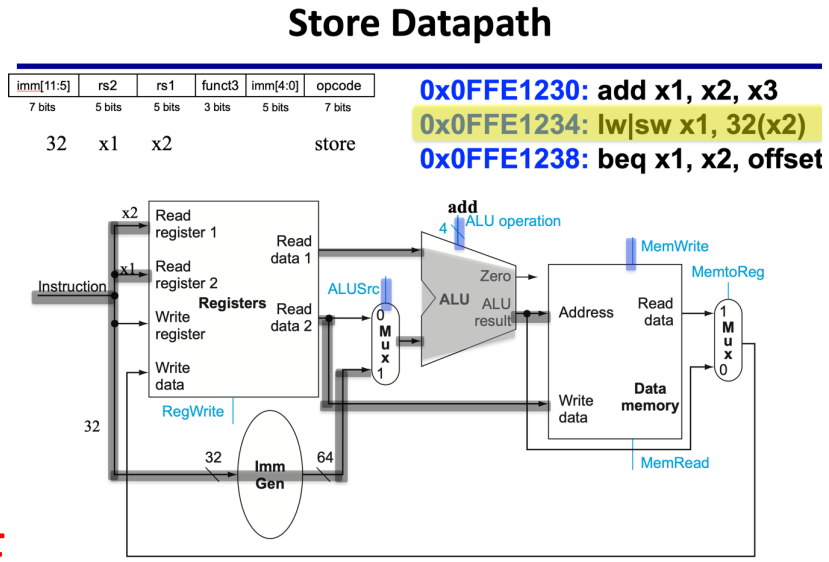
Code Scheduling to Avoid Stalls (Software Solution)

- Reorder code to avoid use of load result in the next instruction
- C code for `a = b + e; c = b + f;`



To Check Cycles Delayed and How Forward Works in Different Cases

- In the 5-stage pipeline, check whether the results can be generated before it is being used
  - If so, forwarding
  - If not, stall
- Load-Use
- Produce-Store
  - sw rs2, offset(rs1)
    - sw needs rs1 to be ready at the EXE stage
    - sw needs rs2 to be ready at the MEM stage



add **x9,** x7, x8
sw x10, 32**(x9)**

2-cycle delay if no forwarding
No delay with forwarding
(Forwarding from EXE to EXE)

add **x9**, x7, x8
sw **x9**, 32(x31)

2-cycle delay if no forwarding
No delay with forwarding
(forwarding from EXE to MEM)

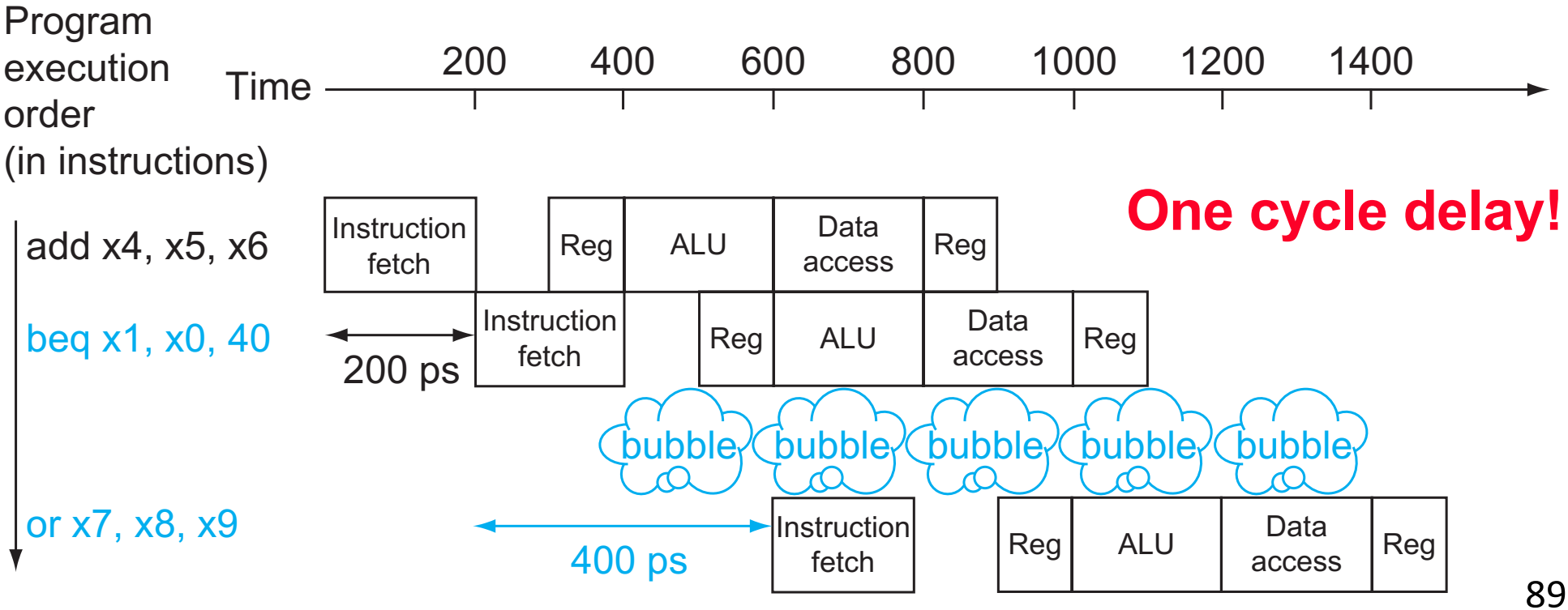
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline might fetch incorrect instruction in the next cycle after a beq instru is fetched
    - Still working on ID stage of branch

- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - **Add hardware to do it in ID stage**

Stall on Branch

- Wait until branch outcome determined before fetching next instruction
  - One cycle stall (bubble) if branch condition is determined at ID stage
  - Two cycles stall if branch condition is determined at EXE stage

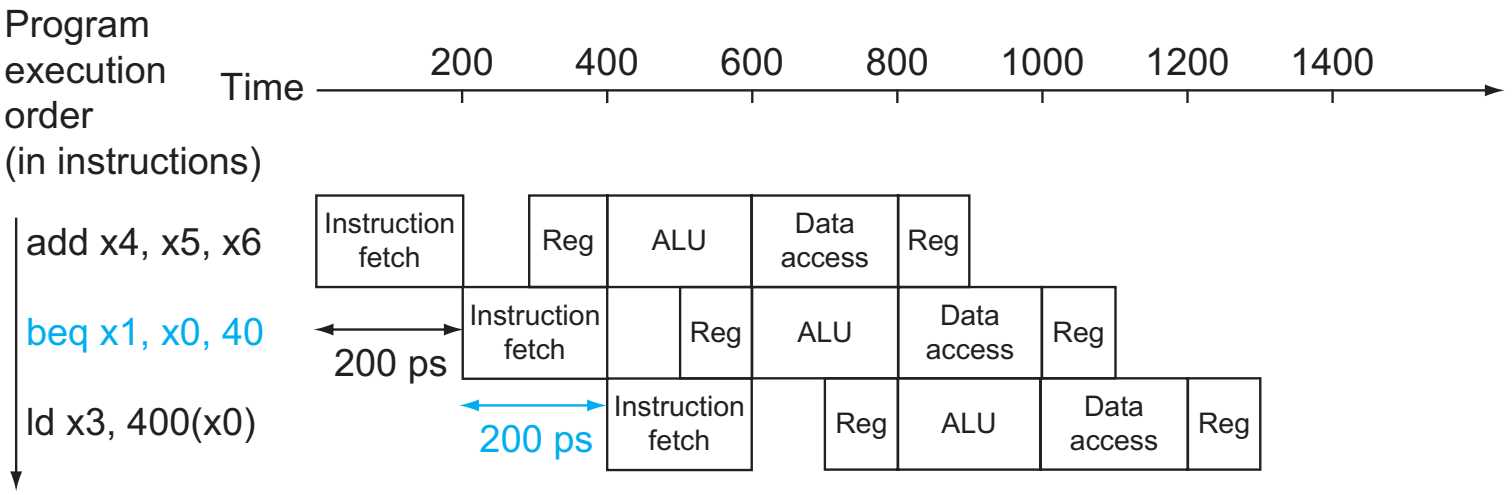


Branch Prediction

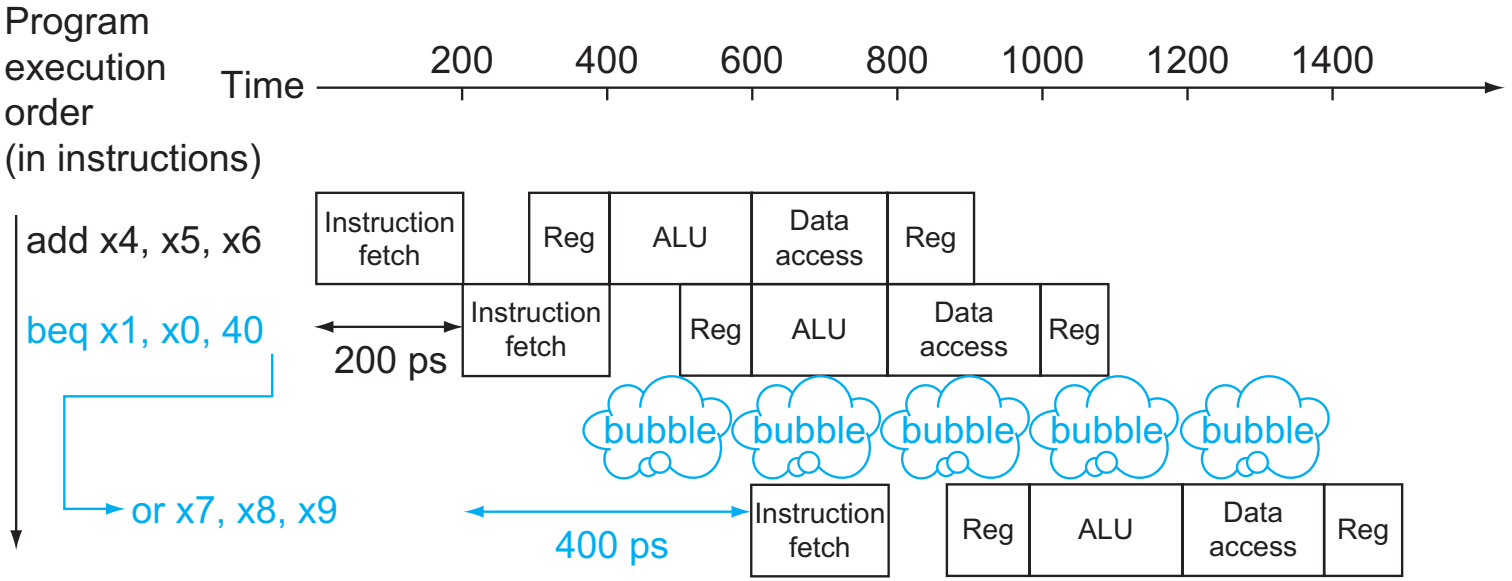
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

RISC-V with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

Pipeline Execution Diagram: Steps

1. Identify RAW dependencies between two instructions **that are one after the other or there is one instruction in between**
   - AL-Use: 2-cycle delay without forwarding, no delay with forwarding
   - Load-Use: 2-cycle delay without forwarding, 1 cycle delay with forwarding
     - With forwarding, we can reschedule load to eliminate the 1 cycle delay even with forwarding
   - No need to looking for RAW dependency between instructions that are far from each other (>=1 instructions in between)
     - Thus only check for the two instructions that could be executed **one after another or has one other instruction in between**
2. Identify branch instruction
   - 1 cycle delay (or two cycles delay) depending on the implementation (question)
3. Pipeline diagrams (4 situations)
   - a) No pipeline at all, one cycle per stage, no overlap
   - b) Pipeline with no forwarding, 2 cycle delay for AL-USE, Load-USE, beq (EXE outcome)
   - c) Pipeline with forwarding, 1 cycle delay for Load-use, and 2 cycle-delay for beq
   - d) Pipeline with forwarding and load-use rescheduling: reschedule the instruction to eliminate the 1-cycle delay for load-use

- No any two instructions can be in the same stage in the same cycle
  - Structural hazard

```
for (i=1; i<M-1; i++) B2[i] = B[i-1] + B[i] + B[i+1];
```

- Base address B and B2 are in register x22 and x23. i is stored in register x5, M is stored in x4.

Using beq (==) for (<) to exit

```
        add x5, x0, 1       // i=0
        add x22, x4, -1     // loop bound x22 has M-1
LOOP: beq x5, x22, Exit
        slliw x6, x5, 2     // x6 now store i*4, slliw is i<<2 (shift left logic)
        add x7, x22, x6     // x7 now stores address of B[i].
        lw x9, 0(x7)        // load B[i] from memory location (x7+0) to x9
        lw x10, -4(x7)      // load B[i-1] to x10
        add x9, x10, x9     // x9 = B[i] + B[i-1]
        lw x10, 4(x7)       //load B[i+1] to x10
        add x9, x10, x9     // x9 = B[i-1] + B[i] + B[i+1]
        add x8, x23, x6     // x8 now stores the address of B2[i]
        sw x9, 0(x8)        // store value for B2[i] from register x9 to memory (x8+0)
        addi x5, x5, 1      // i++
        beq x0, x0, LOOP
Exit:
```

```
for (i=1; i<M-1; i++) B2[i] = B[i-1] + B[i] + B[i+1];
```

- Base address B and B2 are in register x22 and x23. i is stored in register x5, M is stored in x4.

Using beq (==) for (<) to exit

```
1.          add x5, x0, 1
2.          add x22, x4, -1
3.    LOOP: beq x5, x22, Exit
4.          slliw x6, x5, 2
5.          add x7, x22, x6
6.          lw x9, 0(x7)
7.          lw x10, -4(x7)
8.          add x9, x10, x9
9.          lw x10, 4(x7)
10.         add x9, x10, x9
11.         add x8, x23, x6
12.         sw x9, 0(x8)
13.         addi x5, x5, 1
14.         beq x0, x0, LOOP
15.   Exit:
```

RAW Dependencies

Instruction that writes the register	Instruction that reads the register	The register	# instructions in between	Load-use
add x5, x0, 1	beq x5, x22, Exit	x5	1	
add x22, x4, -1	beq x5, x22, Exit	x22	0	
slliw x6, x5, 2	add x7, x22, x6	x6	0	
add x7, x22, x6	lw x9, 0(x7)	x7	0	
add x7, x22, x6	lw x10, -4(x7)	x7	1	
lw x9, 0(x7)	add x9, x10, x9	x9	1	Y
lw x10, -4(x7)	add x9, x10, x9	x10	0	Y
lw x10, 4(x7)	add x9, x10, x9	x10	0	Y
add x9, x10, x9	sw x9, 0(x8)	x9	1	
add x8, x23, x6	sw x9, 0(x8)	x8	0	
addi x5, x5, 1	beq x5, x22, Exit	x5	1	

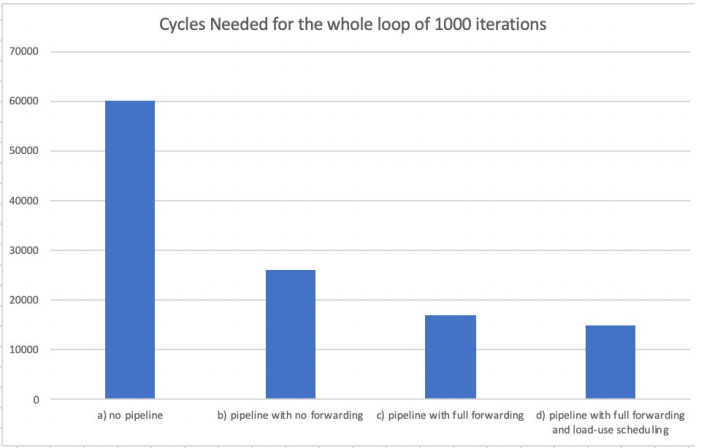
	Cycles Needed for the whole loop of 1000 iterations	
a) no	60000	Each iteration has 12 instructions, 5 cycles to finish each instruction. Thus each iteration needs 12*5 cycles, for total 1000 iterations, it needs 60000 plus the 10 cycles for the very first two instructions.
b) pip	26000	Each iteration needs 26 cycles, thus 26000 cycles for 1000 iterations. 26006 is the actual number of cycles if we count the starting 2 cycles and the ending 4 cycles.
c) pip	17000	First iteration takes 16 cycles, other iterations each takes 17 cycles. 17005 is the actual number of cycles if we count the starting 2 cycles and the ending 4 cycles.
d) pip	15000	First iteration takes 14 cycles, other iterations each takes 15 cycles. 15005 is the actual number of cycles if we count the starting 2 cycles and the ending

slliw x6, x5, 2	add x7, x22, x6	x6	0		
add x7, x22, x6	lw x9, 0(x7)	x7	0		
add x7, x22, x6	lw x10, -4(x7)	x7	1		
lw x9, 0(x7)	add x9, x10, x9	x9	1		Y
lw x10, -4(x7)	add x9, x10, x9	x10	0		Y
lw x10, 4(x7)	add x9, x10, x9	x10	0		Y

				Cycles																																
	a)	no pipeline	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	
		add x5, x0, 1	IF	ID	EX	ME	WB																													
		add x22, x4, -1						IF	ID	EX	ME	WB																							
	LOOP	beq x5, x22, Exit											IF	ID	EX	ME	WB																	
		slliw x6, x5, 2																IF	ID	EX	ME	WB													
		add x7, x22, x6																					IF	ID	EX	ME	WB								
		lw x9, 0(x7)																										IF	ID	EX	ME	WB				
		lw x10, -4(x7)																															IF	ID	EX	
		add x9, x10, x9																																		
		lw x10, 4(x7)																																		
		add x9, x10, x9																																		
				start																																
	b)	pipeline no forwarding	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	
		add x5, x0, 1	IF	ID	EX	ME	WB																													
		add x22, x4, -1		IF	ID	EX	ME	WB																													
Two cycles delay because RAW hazard from add to beq for x22	LOOP	beq x5, x22, Exit					IF	ID	EX	ME	WB																									
Two cycles delay because of beq		slliw x6, x5, 2						IF	ID	EX	ME	WB																									
Two cycles deplay because of RAW hazards from slliw to add for x6		add x7, x22, x6							IF	ID	EX	ME	WB																							
Two cycles delay because of RAW hazard from add to lw for x7		lw x9, 0(x7)								IF	ID	EX	ME	WB																						
		lw x10, -4(x7)									IF	ID	EX	ME	WB																					
Two cycles delay because Load-Use RAW hazard from lw to add for x10		add x9, x10, x9										IF	ID	EX	ME	WB																				
		lw x10, 4(x7)											IF	ID	EX	ME	WB																			
Two cycles delay because Load-Use RAW hazard from lw to add for x10		add x9, x10, x9												IF	ID	EX	ME	WB																	
		add x8, x23, x6													IF	ID	EX	ME	WB																
Two cycles delay because of RAW hazard from add to sw for x8		sw x9, 0(x8)														IF	ID	EX	ME	WB															
		addi x5, x5, 1															IF	ID	EX	ME	WB														
		beq x0, x0, LOOP																IF	ID	EX	ME	WB													
Two cycles delay because of beq		beq x5, x22, Exit																	IF	ID	EX	ME	WB												
	second iteration	slliw x6, x5, 2																		IF	ID	EX	ME	WB											
		add x7, x22, x6																			IF	ID	EX	ME	WB										
		lw x9, 0(x7)																				IF	ID	EX	ME	WB									
		lw x10, -4(x7)																					IF	ID	EX	ME	WB								
		add x9, x10, x9																						IF	ID	EX	ME	WB							
		lw x10, 4(x7)																							IF	ID	EX	ME	WB						
		add x9, x10, x9																								IF	ID	EX	ME	WB					
		add x8, x23, x6																									IF	ID	EX	ME	WB				
		sw x9, 0(x8)																										IF	ID	EX	ME	WB			
		addi x5, x5, 1																											IF	ID	EX	ME	WB		
		beq x0, x0, LOOP																												IF	ID	EX	ME	WB		

The first iteration, 26 cycles

	Cycles Needed for the whole loop of 1000 iterations	
a) no	60000	Each iteration has 12 instructions, 5 cycles to finish each instruction. Thus each iteration needs 12*5 cycles, for total 1000 i
b) pip	26000	Each iteration needs 26 cycles, thus 26000 cycles for 1000 iterations. 26006 is the actual number of cycles if we count the s
c) pip	17000	First iteration takes 16 cycles, other iterations each takes 17 cycles. 17005 is the actual number of cycles if we count the sta
d) pip	15000	First iteration takes 14 cycles, other iterations each takes 15 cycles. 15005 is the ac Vertical (Value) Axis e count the sta



			c)	pipeline with forwar	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
				add x5, x0, 1	IF	ID	EX	ME	WB																																					
				add x22, x4, -1		IF	ID	EX	ME	WB																																					
			LOOP	beq x5, x22, Exit			IF	ID	EX	ME	WB																																			
	Two cycles delay because of beq			slliw x6, x5, 2						IF	ID	EX	ME	WB																															
				add x7, x22, x6							IF	ID	EX	ME	WB																														
				lw x9, 0(x7)								IF	ID	EX	ME	WB																													
		load-use hazard, 1 cycle delay		**lw x10, -4(x7)**									IF	ID	EX	ME	WB																												
				add x9, x10, x9										IF	ID	EX	ME	WB																											
		load-use hazard, 1 cycle delay		**lw x10, 4(x7)**											IF	ID	EX	ME	WB																										
				add x9, x10, x9												IF	ID	EX	ME	WB																									
				add x8, x23, x6													IF	ID	EX	ME	WB																								
				sw x9, 0(x8)														IF	ID	EX	ME	WB																							
				addi x5, x5, 1															IF	ID	EX	ME	WB																						
				beq x0, x0, LOOP																IF	ID	EX	ME	WB																					
	2 cycles delay because of beq			beq x5, x22, Exit																	IF	ID	EX	ME	WB																			
	2 cycles delay because of beq			slliw x6, x5, 2																		IF	ID	EX	ME	WB																		
				add x7, x22, x6																			IF	ID	EX	ME	WB																		
				lw x9, 0(x7)																				IF	ID	EX	ME	WB																	
				lw x10, -4(x7)																					IF	ID	EX	ME	WB																
				add x9, x10, x9																						IF	ID	EX	ME	WB															
				lw x10, 4(x7)																							IF	ID	EX	ME	WB														
				add x9, x10, x9																								IF	ID	EX	ME	WB													
				add x8, x23, x6																									IF	ID	EX	ME	WB											
				sw x9, 0(x8)																										IF	ID	EX	ME	WB										
				addi x5, x5, 1																											IF	ID	EX	ME	WB									
				beq x0, x0, LOOP																												IF	ID	EX	ME	WB								
			d)	pipeline with forwar	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
				add x5, x0, 1	IF	ID	EX	ME	WB																																					
				add x22, x4, -1		IF	ID	EX	ME	WB																																					
			LOOP	beq x5, x22, Exit			IF	ID	EX	ME	WB																																			
				slliw x6, x5, 2						IF	ID	EX	ME	WB																																	
				add x7, x22, x6							IF	ID	EX	ME	WB																															
				lw x9, 0(x7)								IF	ID	EX	ME	WB																													
	The four instructions are rescheduled			**lw x10, -4(x7)**									IF	ID	EX	ME	WB																											
	so we have two loads first and then			**lw x11, 4(x7)**										IF	ID	EX	ME	WB																										
	two add, we will need to use one			**add x9, x10, x9**											IF	ID	EX	ME	WB																									
	register x11 to help on this			**add x9, x11, x9**												IF	ID	EX	ME	WB																								
				add x8, x23, x6													IF	ID	EX	ME																								
				sw x9, 0(x8)														IF	ID	EX	WB																							
				addi x5, x5, 1															IF	ID	EX	WB																						
				beq x0, x0, LOOP																IF	ID	EX	WB																					
	2 cycles delay because of beq			beq x5, x22, Exit																	IF	ID	EX	ME	WB																		
	2 cycles delay because of beq			slliw x6, x5, 2																		IF	ID	EX	ME	WB																	
				add x7, x22, x6																			IF	ID	EX	ME	WB																	
				lw x9, 0(x7)																				IF	ID	EX	ME	WB																
				lw x10, -4(x7)																					IF	ID	EX	ME	WB															
				lw x11, 4(x7)																						IF	ID	EX	ME	WB														
				add x9, x10, x9																							IF	ID	EX	ME	WB													
				add x9, x11, x9																								IF	ID	EX	ME	WB												
				add x8, x23, x6																									IF	ID	EX	ME												

The first iteration, 16 cycles

The second iteration, and the rest ..., 17 cycles

The first iteration, 14 cycles

The second iteration, and the rest ..., 15 cycles

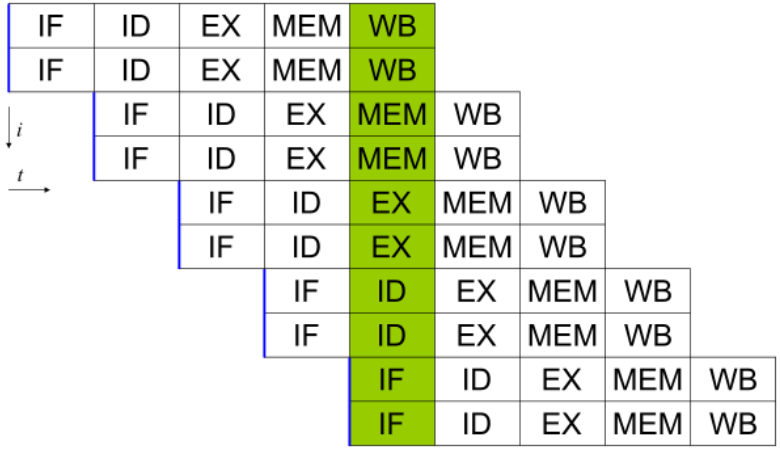
Chapter 4: The Processor

- **Lecture**
  - **4.1 Introduction**
  - **4.2 Logic Design Conventions**
  - **4.3 Building a Datapath**
- **Lecture**
  - **4.4 A Simple Implementation Scheme**
- **Lecture**
  - **4.5 An Overview of Pipelining**
- **Lecture (Pipeline implementation), will not be covered!**
  - 4.6 Pipelined Datapath and Control
  - 4.7 Data Hazards: Forwarding versus Stalling
  - 4.8 Control Hazards
  - ~~4.9 Exceptions~~
  - ~~4.13 Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations~~
- **Lecture (Advanced pipeline techniques and real-world CPU examples)**
  - 4.10 Parallelism via Instructions
  - 4.11 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines
  - 4.12 Going Faster: Instruction-Level Parallelism and Matrix Multiply
  - ~~4.14 Fallacies and Pitfalls~~
  - **4.15 Concluding Remarks**

Instruction-Level Parallelism (ILP)

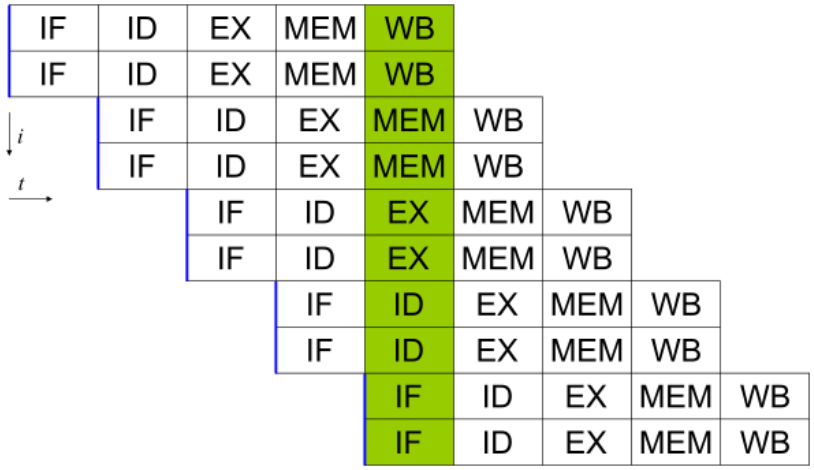
- Pipelining: executing multiple instructions in parallel
  - CPI ~= 1
- To increase ILP
  - Deeper pipeline by having more stages
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
- Performance of Multiple issue
  - E.g., 4GHz 2-way multiple-issue
  - IPC (Instruction Per Cycle): 2
    - peak CPI = 0.5
    - Instr/Second: $4*10^9 * 2 = 8*10^9$
  - But dependencies reduce this in practice
    - Pipeline hazards for single issue happen
    - Not always have two instruction to be issued per cycle

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i+1*		IF	ID	EX	MEM	WB			
Instruction *i+2*			IF	ID	EX	MEM	WB		
Instruction *i+3*				IF	ID	EX	MEM	WB	
Instruction *i+4*					IF	ID	EX	MEM	WB



Multiple Issue

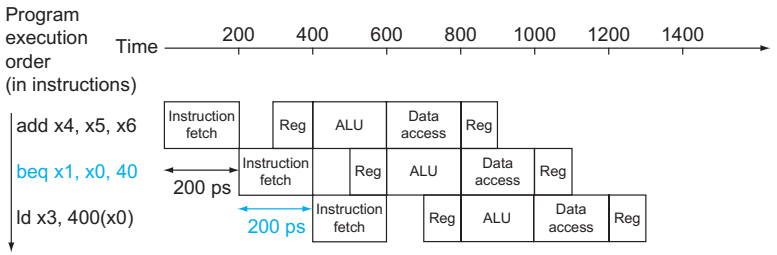
- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- **Dynamic multiple issue**
  - **CPU examines instruction stream and chooses instructions to issue each cycle**
  - **Compiler can help by reordering instructions**
  - **CPU resolves hazards using advanced techniques at runtime**

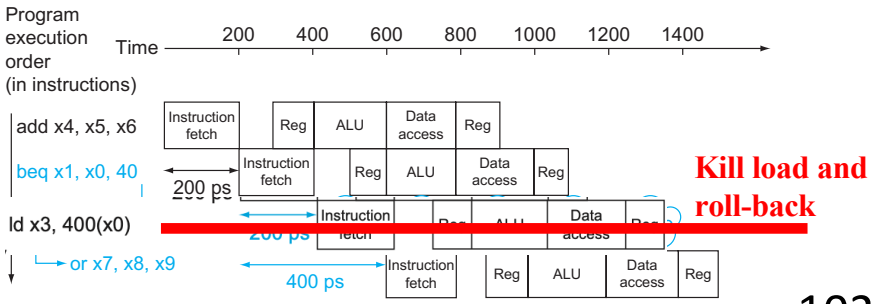


High-end processors (desktop, server) use dynamic multiple issue

Speculation

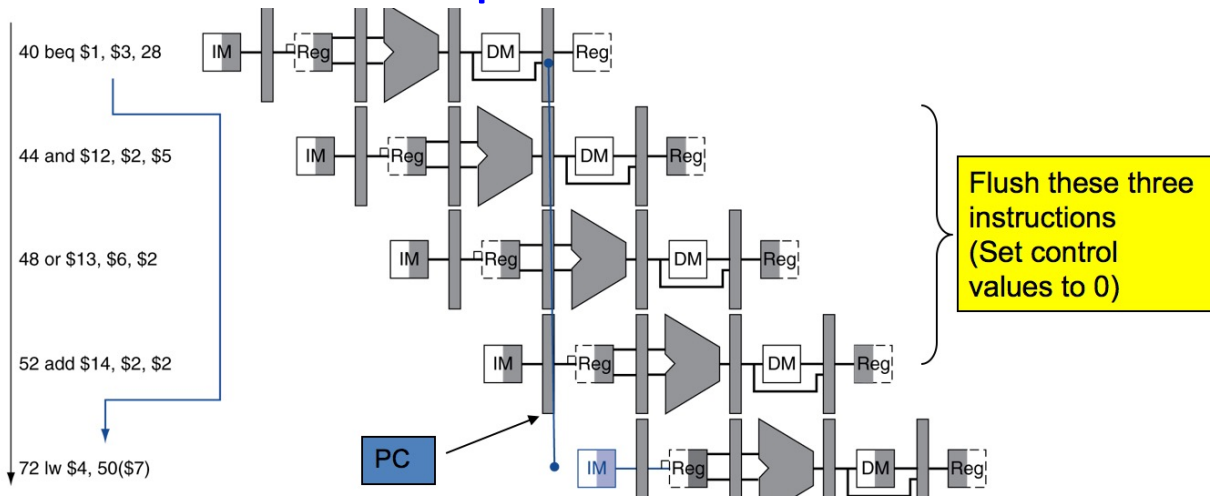
- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated





Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include "fix-up" instructions to recover from incorrect guess
    - Move lw to remove load-use cycle delay in RAW hazards
    - Schedule delayed slot
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation



Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
    - **2 IPC: ALU/BEQ + LW/SW**
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

RISC-V with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

RISC-V with Static Dual Issue

- Double resources
  - 2 set of register R/W ports
  - 2 ALUs
    - Top for Load/store
    - Bottom for AL and BEQ

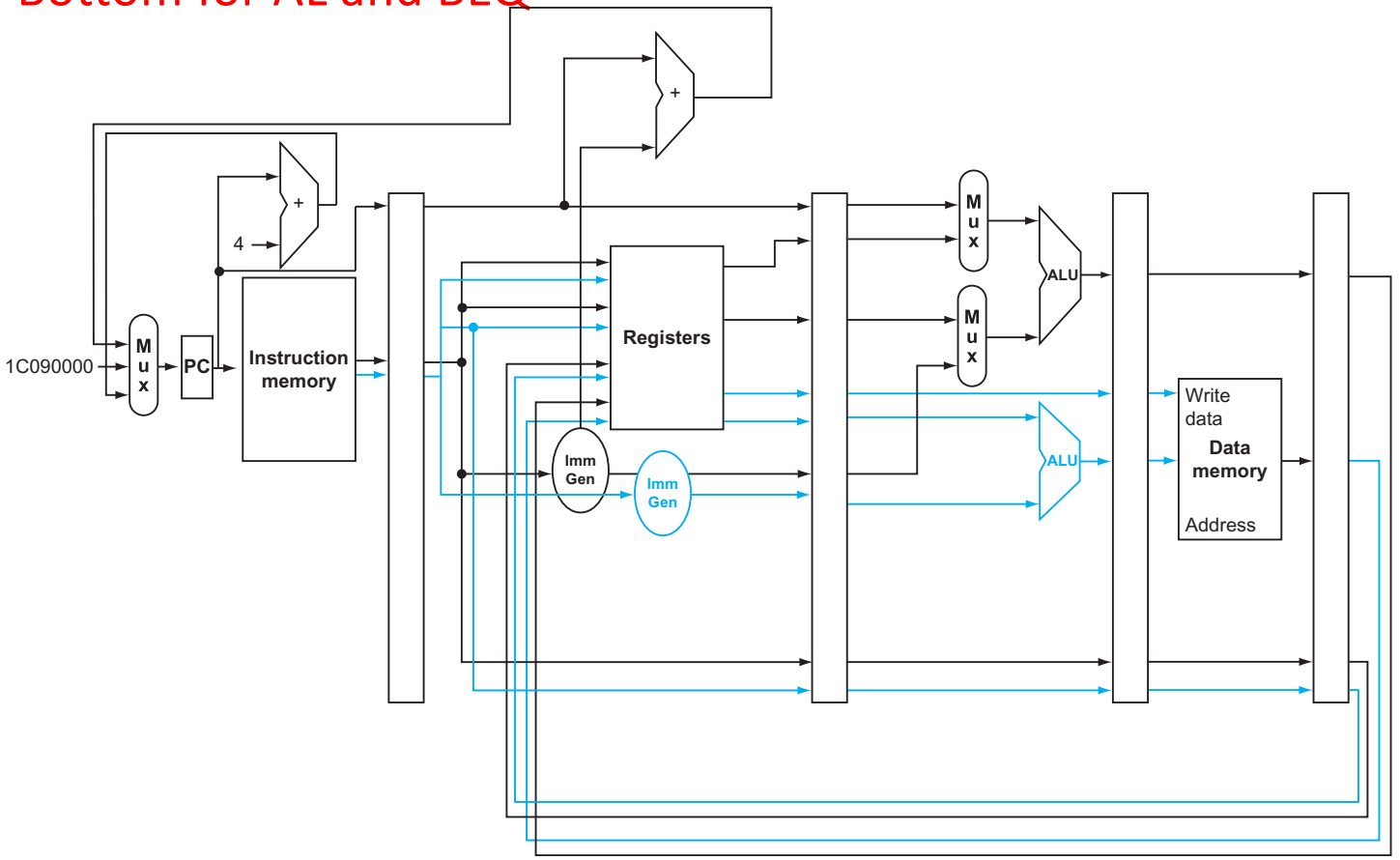


Figure 4.66

Hazards in the Dual-Issue RISC-V

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - `add  x10, $s0, $s1`
      `load $s2, 0(x10)`
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Instruction type	Pipe stages					
ALU or branch instruction	IF	ID	EX	MEM	WB	
Load or store instruction	IF	ID	EX	MEM	WB	
ALU or branch instruction		IF	ID	EX	MEM	
Load or store instruction		IF	ID	EX	MEM	
ALU or branch instruction			IF	ID	EX	
Load or store instruction			IF	ID	EX	
ALU or branch instruction				IF	ID	
Load or store instruction				IF	ID	

Scheduling Example

```
for (i=1000;i!=0;i--)
  A[i] += a;
```
Each element is 8 bytes

- Schedule this for dual-issue RISC-V

```
Loop: ld   x31, 0(x20)       # x31=array element
      add  x31, x31, x21     # add scalar in x21
      sd   x31, 0(x20)       # store result
      addi $20, x20, -8      # decrement pointer
      blt  x22, x20, Loop # compare to loop limit
                             # branch if x20 > x22
```

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		ld x31,0(x20)	1
	addi x20,x20,-8		2
	add x31,x31,x21		3
	blt x22,x20,Loop	sd x31,8(x20)	4

Figure 4.67

- **addi and ld CANNOT be in one cycle**
  - **Load-use (ld-add) hazard: 1 cycle delay**
- **IPC = 5/4 = 1.25 (c.f. peak IPC = 2)**

Loop Unrolling

```
for (i=1000;i!=0;i--)
  A[i] += a;
```

Unrolling with factor 2

```
for (i=1000;i!=0;i-=2) {
  A[i] += a;;
  A[i-1] += a;
}
```

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
  - For two calculations, e.g. A[i]+=a
    - 2 beq vs 1 beq; 2 i-1 vs 1 i-2
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Loop Unrolling Example

Unrolling with factor 4

```
for (i=1000;i!=0;i-=4){
    A[i] += a;
    A[i-1] += a;
    A[i-2] += a;
    A[i-3] += a;
}
```

- Load-use hazard
  - 1 cycle use delay

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi x20,x20,-32	ld x28,0(x20)	1
		ld x29,24(x20)	2
	add x28,x28,x21	ld x30,16(x20)	3
	add x29,x29,x21	ld x31,8(x20)	4
	add x30,x30,x21	sd x28,32(x20)	5
	add x31,x31,x21	sd x29,24(x20)	6
		sd x30,16(x20)	7
	blt x22,x20,Loop	sd x31,8(x20)	8

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Figure 4.68

Summary for Four Iterations of the Loop

Unrolling with factor 4

```
for (i=1000;i!=0;i--)
   A[i] += a;
```

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		ld x31,0(x20)	1
	addi x20,x20,-8		2
	add x31,x31,x21		3
	blt x22,x20,Loop	sd x31,8(x20)	4

```
for (i=1000;i!=0;i-=4){
   A[i] += a;;
   A[i-1] += a;
   A[i-2] += a;
   A[i-3] += a;
}
```

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi x20,x20,-32	ld x28,0(x20)	1
		ld x29,24(x20)	2
	add x28,x28,x21	ld x30,16(x20)	3
	add x29,x29,x21	ld x31,8(x20)	4
	add x30,x30,x21	sd x28,32(x20)	5
	add x31,x31,x21	sd x29,24(x20)	6
		sd x30,16(x20)	7
	blt x22,x20,Loop	sd x31,8(x20)	8

- Original version + single issue
  - Total 20 instructions → ~5 cycles/calculation
- Original version + multi-issue
  - About 4 cycles/calculation
- Unrolling by 4 + single issue
  - 14 instructions → 3.5 cycles/calculation (14/4)
- Unrolling + multi-issue
  - About 2 clocks/calculation (8/4)

Static Multiple Issue

- Compiler groups instructions into "issue packets"
  - Group of instructions that can be issued on a single cycle
    - 2 IPC: ALU/BEQ + LW/SW
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

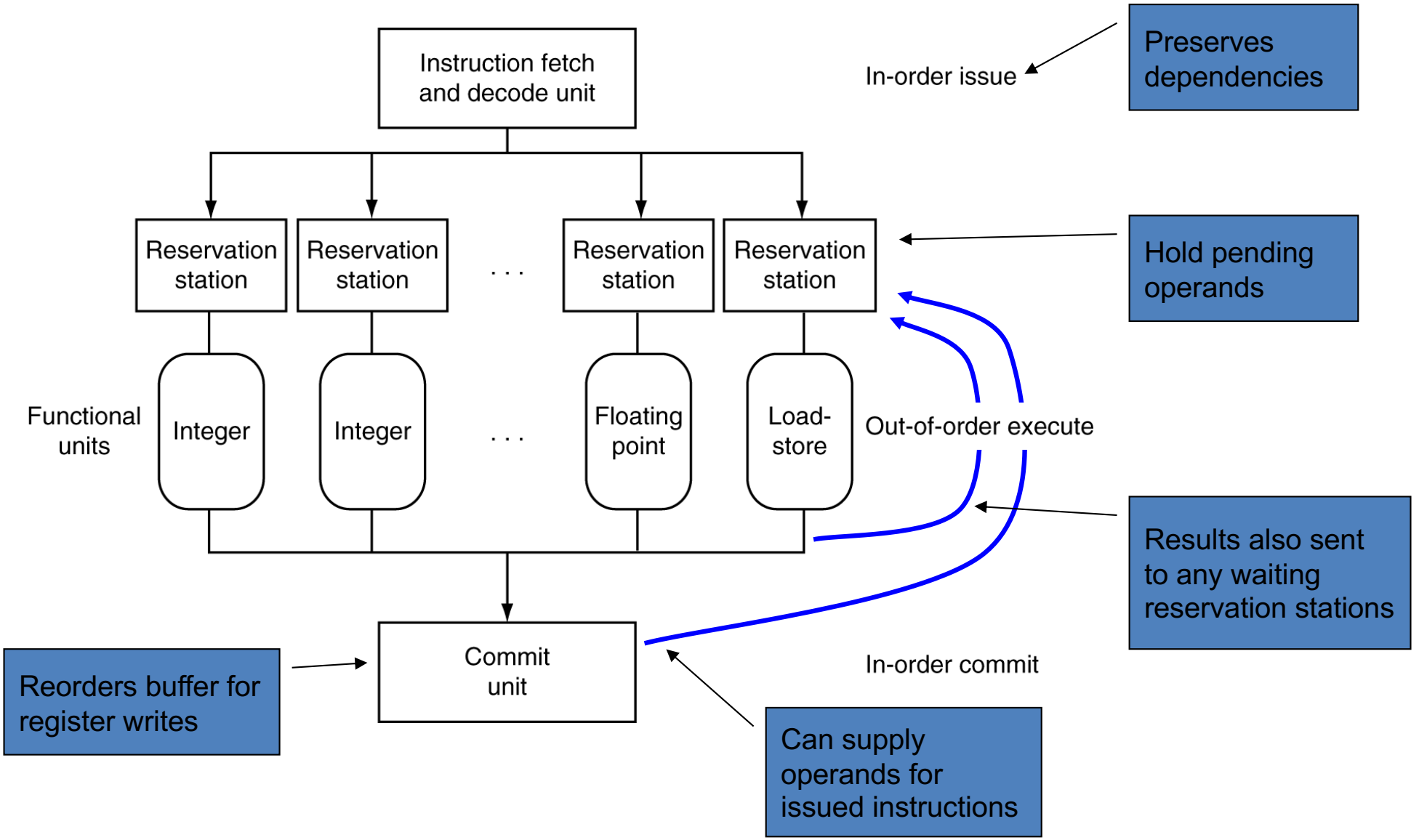
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
ld    x31, 0(x21)
add   x1, x31, x2
sub   x23,x23, x3
andi  x5, x23, 20
```

  - Can start sub while add is waiting for lw

Dynamically Scheduled CPU



Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

ARM Cortex-A53 Pipeline

- Used as the basis for several tablets and cell phones
  - Dual-issue, statically scheduled superscalar with dynamic issue detection → 0.5 CPI ideally

Processor	ARM A53
Market	Personal Mobile Device
Thermal design power	100 milliWatts (1 core @ 1 GHz)
Clock rate	1.5 GHz
Cores/Chip	4 (configurable)
Floating point?	Yes
Multiple Issue?	Dynamic
Peak instructions/clock cycle	2
Pipeline Stages	8
Pipeline schedule	Static In-order
Branch prediction	Hybrid
1st level caches/core	16-64 KiB I, 16-64 KiB D
2nd level cache/core	128–2048 KiB (shared)
3rd level cache (shared)	(platform dependent)

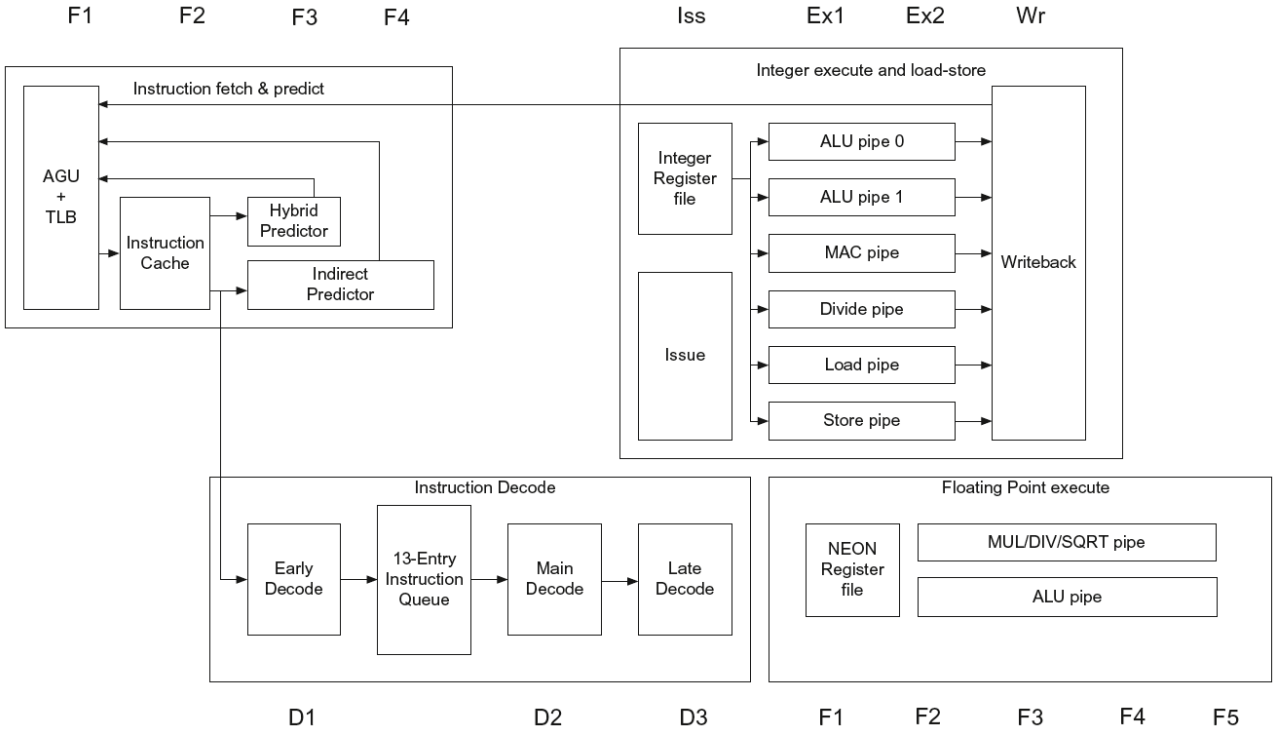


Figure 4.72

ARM Cortex-A53 Performance using SPEC2006

- Ideal CPI: 0.5 since it is 2-way multi-issue (IPC=2)
  - Best case 1.0, median case 1.3, worst 8.6
  - 60% stalls due to pipelining hazards
  - 40% stalls due to the memory hierarchy

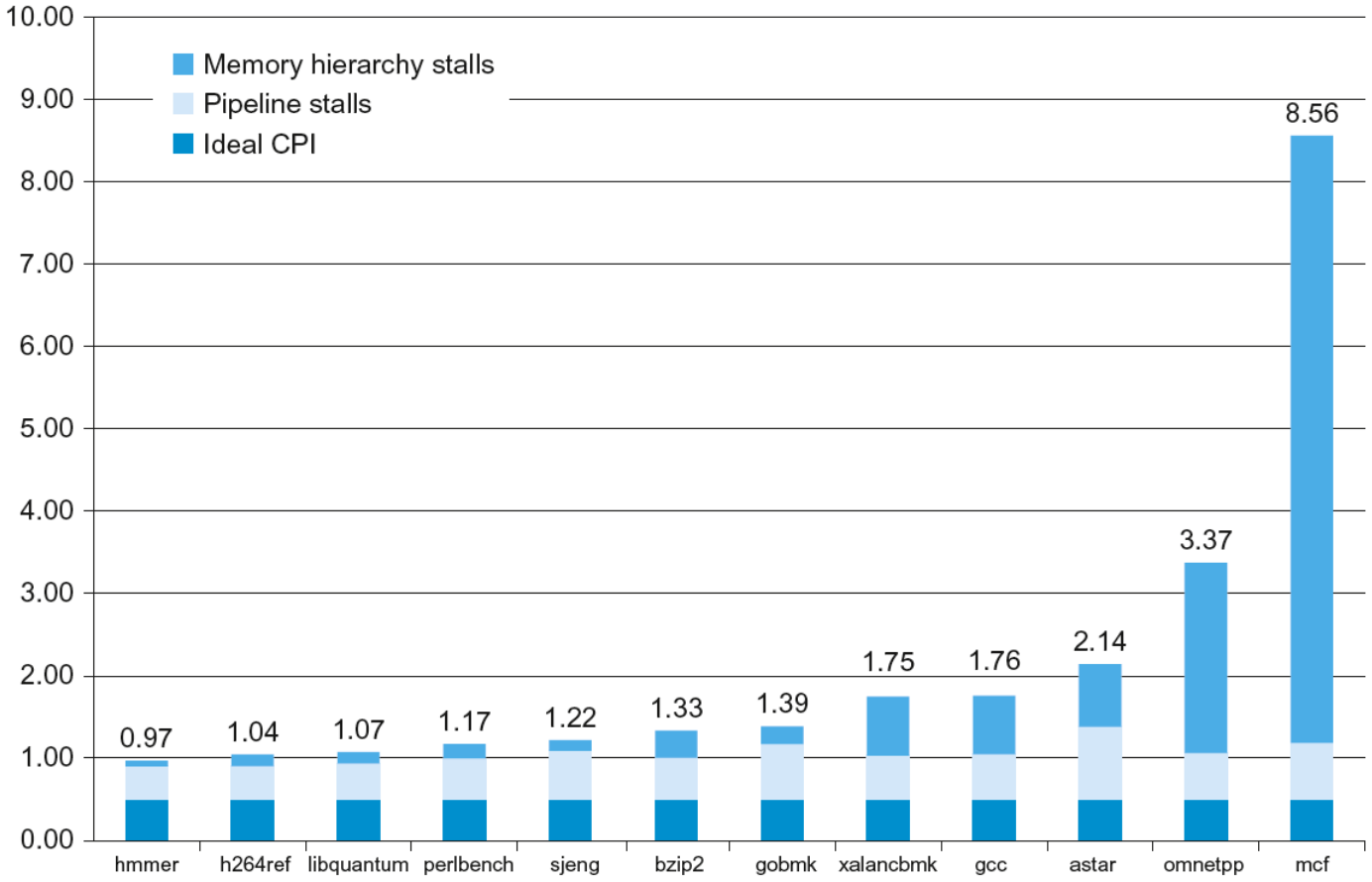
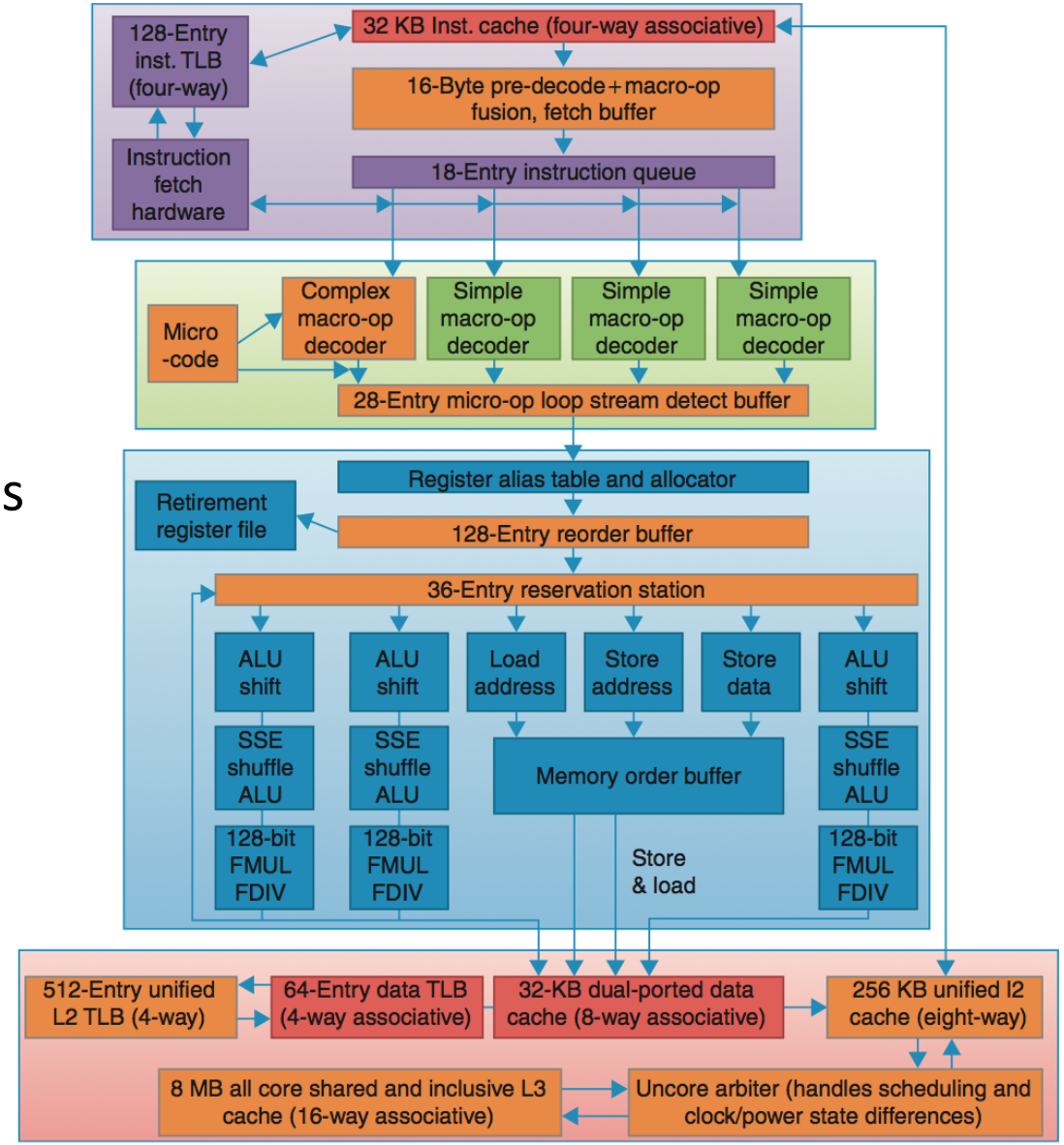


Figure 4.73

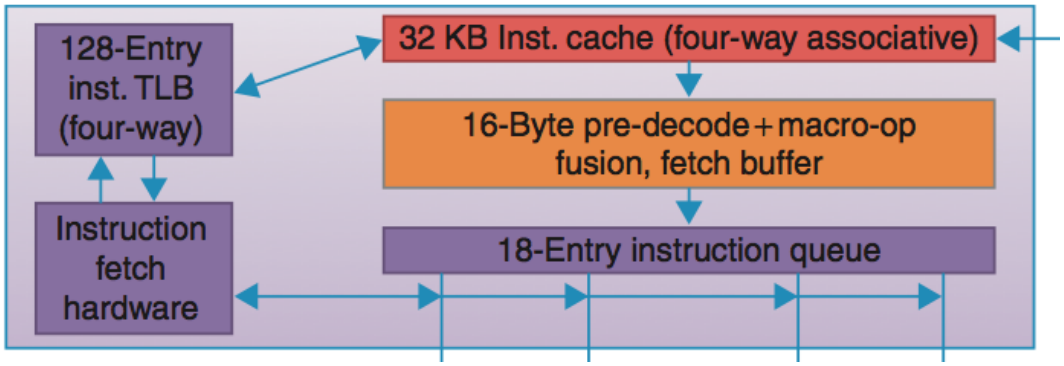
Intel Core i7

- Aggressive out-of-order speculative
- 14 stages pipeline,
- Branch mispredictions costing 17 cycles.
- 48 load and 32 store buffers.
- Six independent functional units
  - 6-wide superscalar

Processor	Intel Core i7 920
Market	Server, Cloud
Thermal design power	130 Watts
Clock rate	2.66 GHz
Cores/Chip	4
Floating point?	Yes
Multiple Issue?	Dynamic
Peak instructions/clock cycle	4
Pipeline Stages	14
Pipeline schedule	Dynamic Out-of-order with Speculation
Branch prediction	2-level
1st level caches/core	32 KiB I, 32 KiB D
2nd level cache/core	256 KiB (per core)
3rd level cache (shared)	2–8 MiB

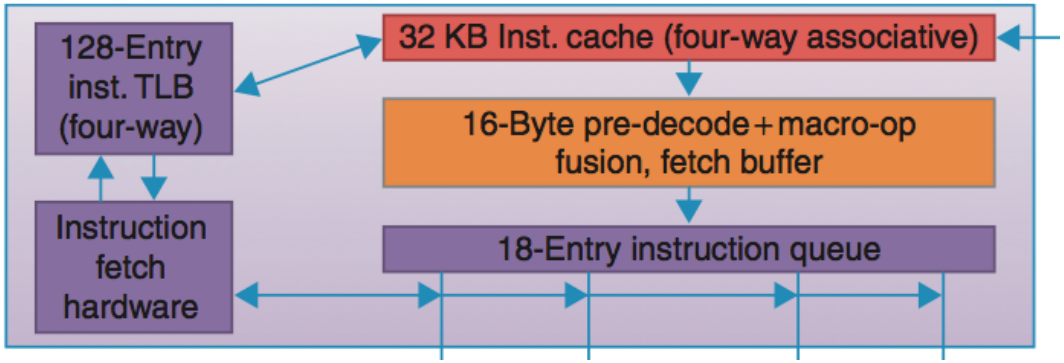


Core i7 Pipeline: IF



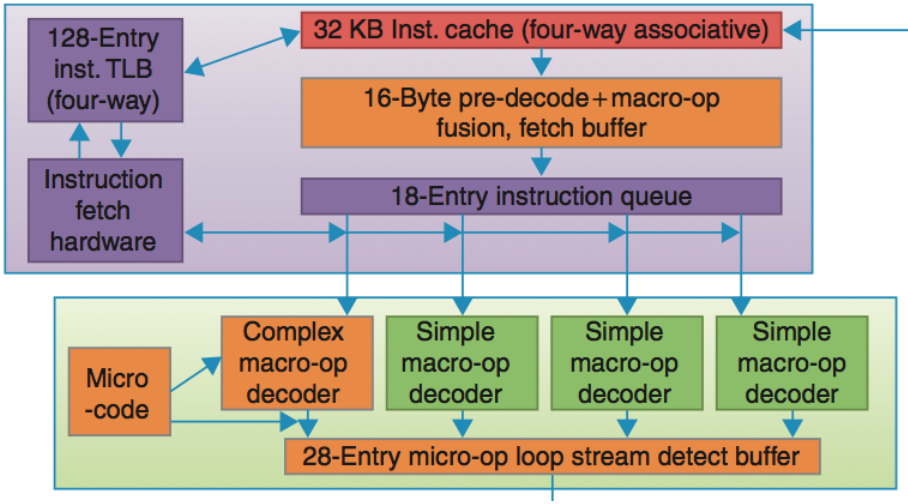
- Instruction fetch – Fetch 16 bytes from the I cache
  - A multilevel branch target buffer to achieve a balance between speed and prediction accuracy.
  - A return address stack to speed up function return.
  - Mispredictions cause a penalty of about 15 cycles.

Core i7 Pipeline: Predecode



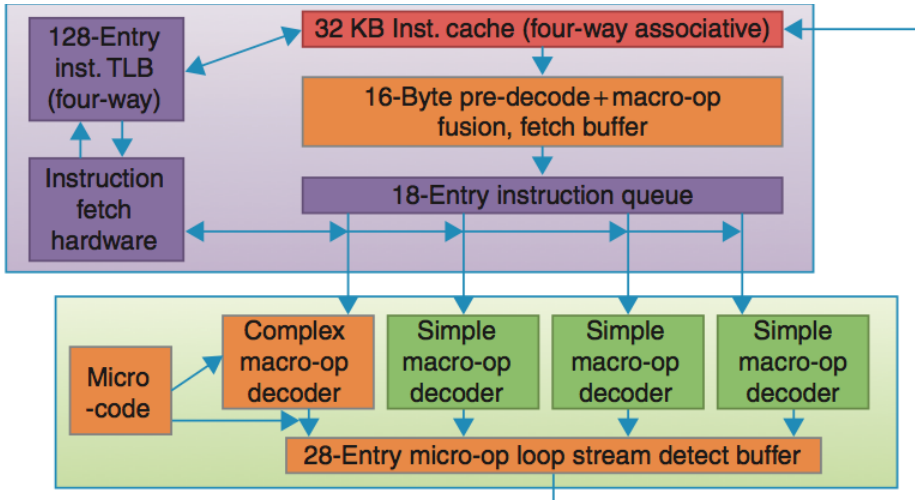
- Predecode –16 bytes instr in the predecode I buffer
  - *Macro-op fusion:* Fuse instr combinations such as compare followed by a branch into a single operation.
  - Instr break down: breaks the 16 bytes into individual x86 instructions.
    - nontrivial since the length of an x86 instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length.
  - Individual x86 instructions (including some fused instructions) are placed into the 18-entry instruction queue.

Core i7 Pipeline: Micro-op decode



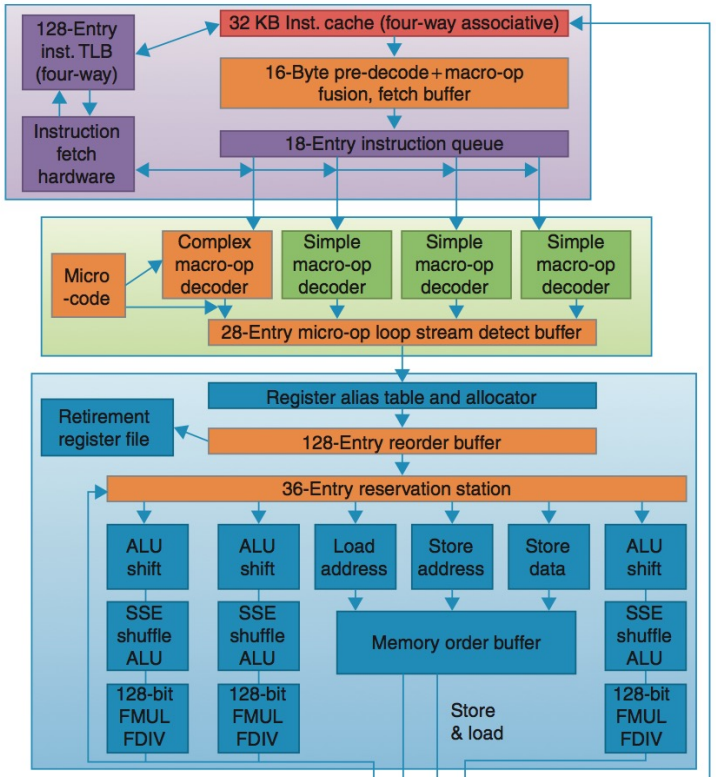
- Micro-op decode – Translate Individual x86 instructions into micro-ops.
  - Micro-ops are simple MIPS-like instructions that can be executed directly by the pipeline (RISC style)
    - introduced in the Pentium Pro in 1997 and has been used since.
  - Three simple micro-op decoders handle x86 instructions that translate directly into one micro-op.
  - One complex micro-op decoder produce the micro-op sequence of complex x86 instr;
    - produce up to four micro-ops every cycle
  - The micro-ops are placed according to the order of the x86 instructions in the 28- entry micro-op buffer.

Core i7 Pipeline: loop stream detection and microfusion



- *loop stream detection* and *microfusion by* the micro-op buffer preforms
  - If there is a sequence of instructions (less than 28 instrs or 256 bytes in length) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer
    - eliminating the need for the instruction fetch and instruction decode stages to be activated.
  - Microfusion combines instr pairs such as load/ALU operation and ALU operation/store and issues them to a single reservation station, thus increasing the usage of the buffer.
    - Study comparing the microfusion and macrofusion by Bird et al. [2007] discovered that microfusion had little impact on per-formance, while macrofusion appears to have a modest positive impact on integer performance and little impact on FP.

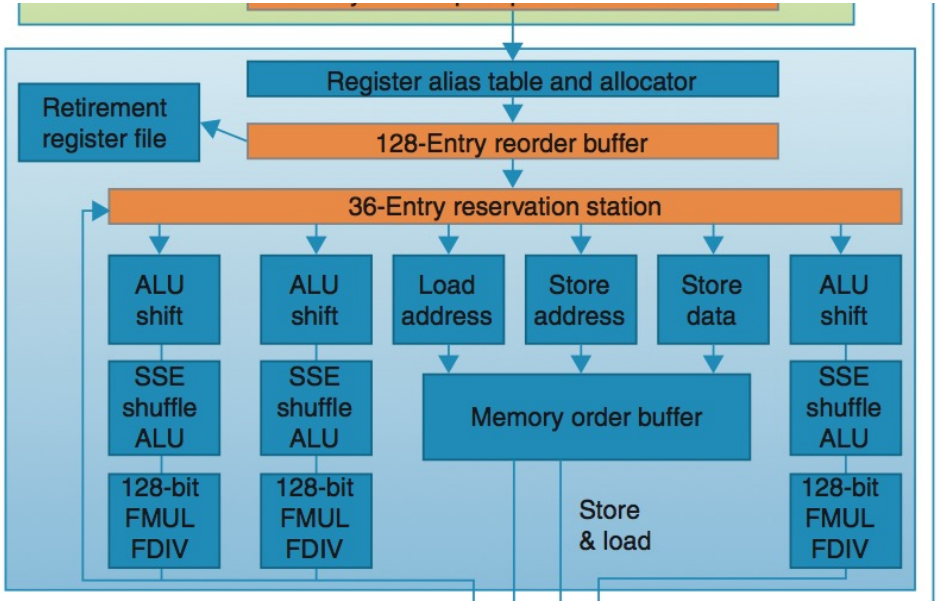
Core i7 Pipeline: Issue



- Basic instruction issue
  - Looking up the register location in the register tables
  - renaming the registers
  - allocating a reorder buffer entry
  - fetching any results from the registers or reorder buffer before sending the micro-ops to the reservation stations.
- 36-entry centralized reservation station shared by six functional units

  Up to six micro-ops may be dispatched to the functional units every clock cycle.

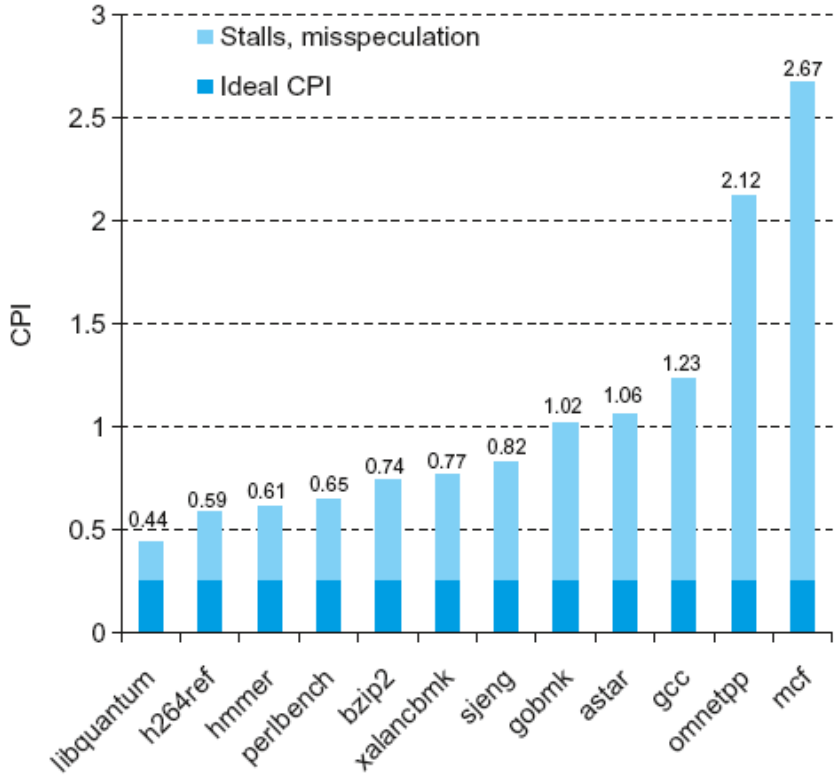
Core i7 Pipeline: EXE and Retirement

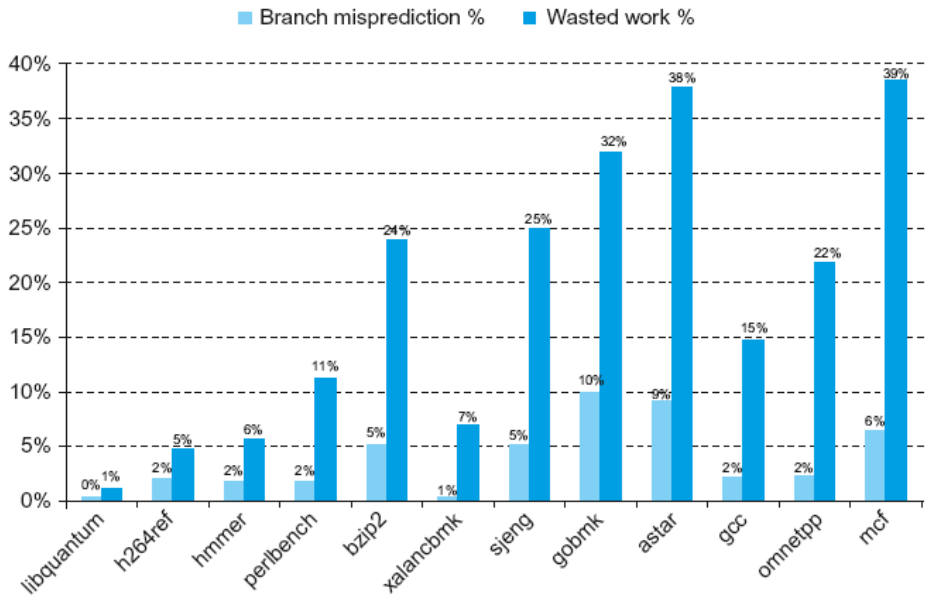


- Micro-ops are executed by the individual function units
  - results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state. The entry corresponding to the instruction in the reorder buffer is marked as complete.
- Retirement
  - When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

Core i7 Performance running SPEC2006 INT

- Ideal CPI: 0.25
- Best 0.44, median 0.79, worst 2.67;





Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Slides and Chapter Sections that are not covered for Fall 2020.

Chapter 4: The Processor

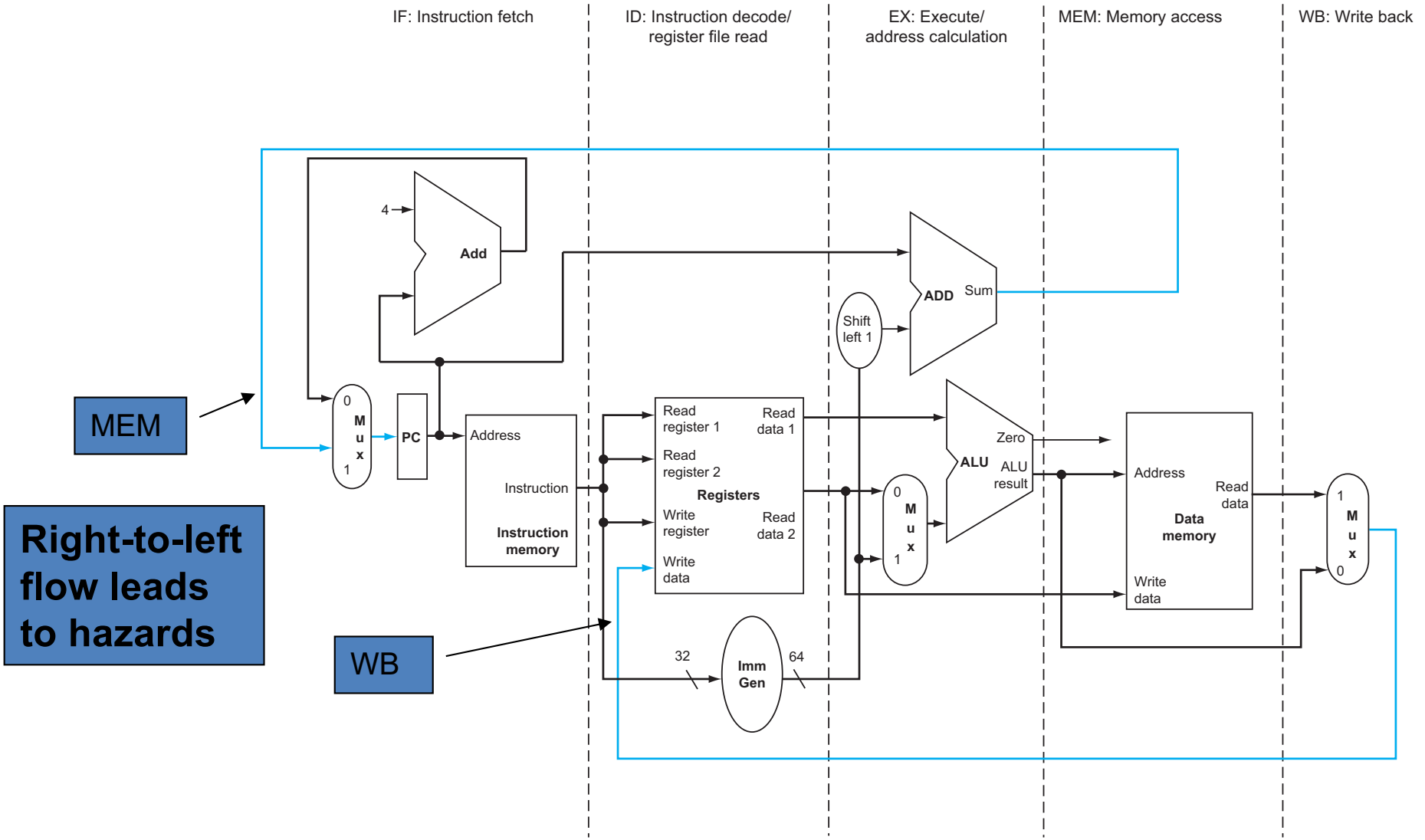
- **Lecture**
  - **4.1 Introduction**
  - **4.2 Logic Design Conventions**
  - **4.3 Building a Datapath**
- **Lecture**
  - **4.4 A Simple Implementation Scheme**
- **Lecture**
  - **4.5 An Overview of Pipelining**

☛ **Lecture (Pipeline implementation), will not be covered!**

  - 4.6 Pipelined Datapath and Control
  - 4.7 Data Hazards: Forwarding versus Stalling
  - 4.8 Control Hazards
  - ~~4.9 Exceptions~~
  - ~~4.13 Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations~~
- **Lecture (Advanced pipeline techniques and real-world CPU examples)**
  - **4.10 Parallelism via Instructions**
  - **4.11 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines**
  - 4.12 Going Faster: Instruction-Level Parallelism and Matrix Multiply
  - ~~4.14 Fallacies and Pitfalls~~
  - **4.15 Concluding Remarks**

MIPS Pipelined Datapath

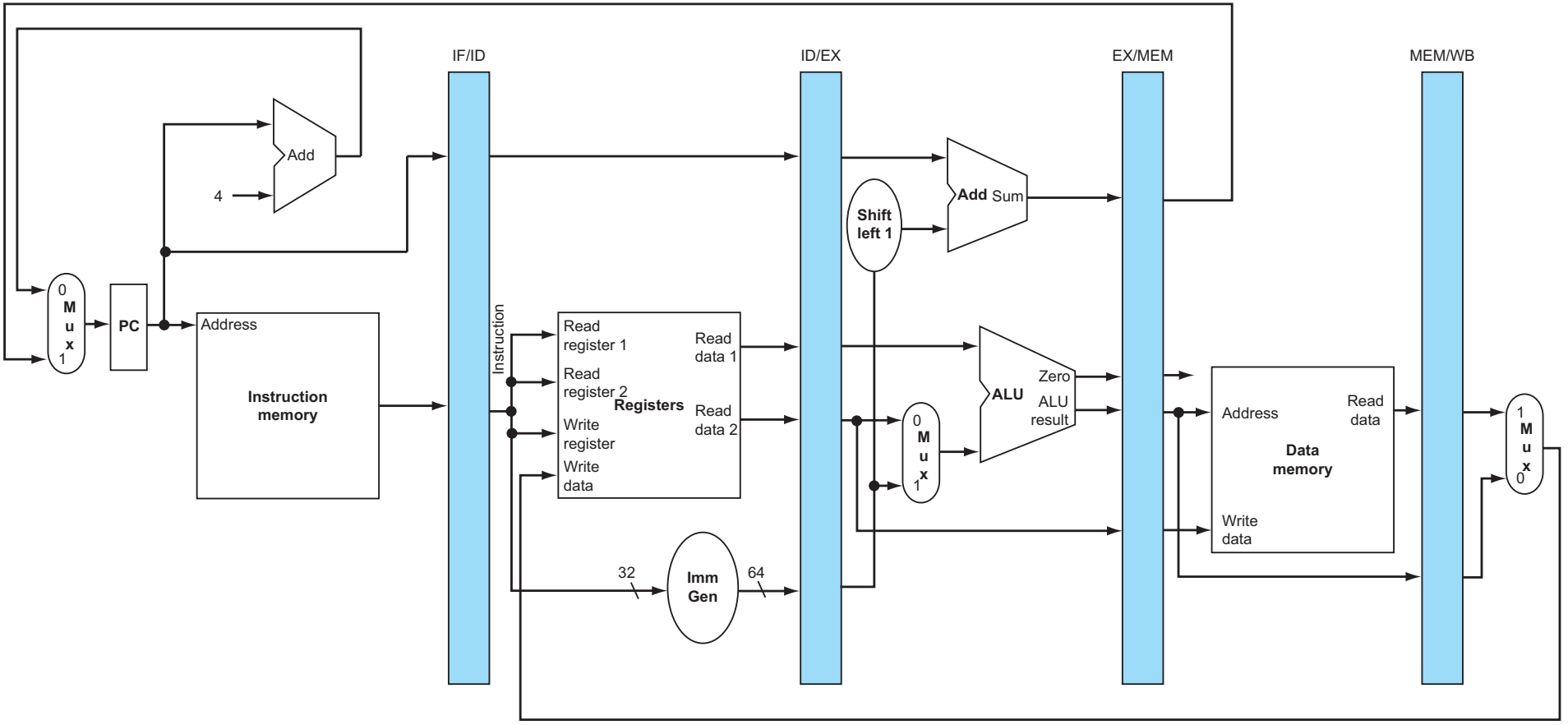
	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i+1*		IF	ID	EX	MEM	WB			
Instruction *i+2*			IF	ID	EX	MEM	WB		
Instruction *i+3*				IF	ID	EX	MEM	WB	
Instruction *i+4*					IF	ID	EX	MEM	WB



Pipeline registers

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i+1*		IF	ID	EX	MEM	WB			
Instruction *i+2*			IF	ID	EX	MEM	WB		
Instruction *i+3*				IF	ID	EX	MEM	WB	
Instruction *i+4*					IF	ID	EX	MEM	WB

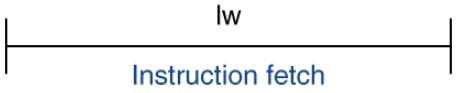
- Registers between stages
  - For each instruction, hold information produced in previous stage/cycle and pass on
  - Each register set (IF/ID, ID/EX, EX/MEM, MEM/WB) has the information for each of the instructions that are in the pipeline



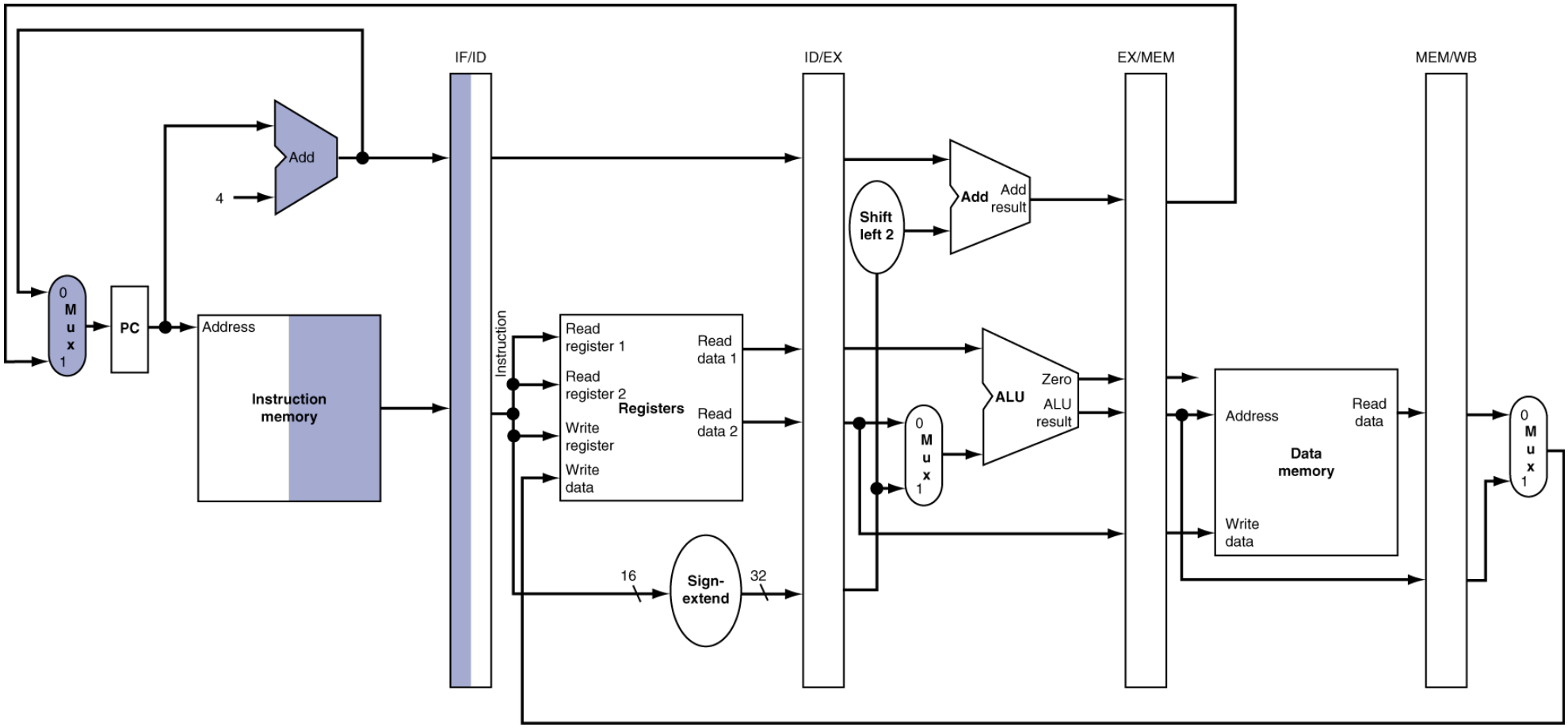
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

IF for Load, Store, …

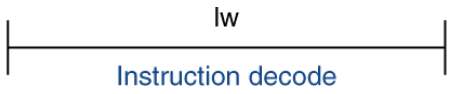


LW $4, 32($5)

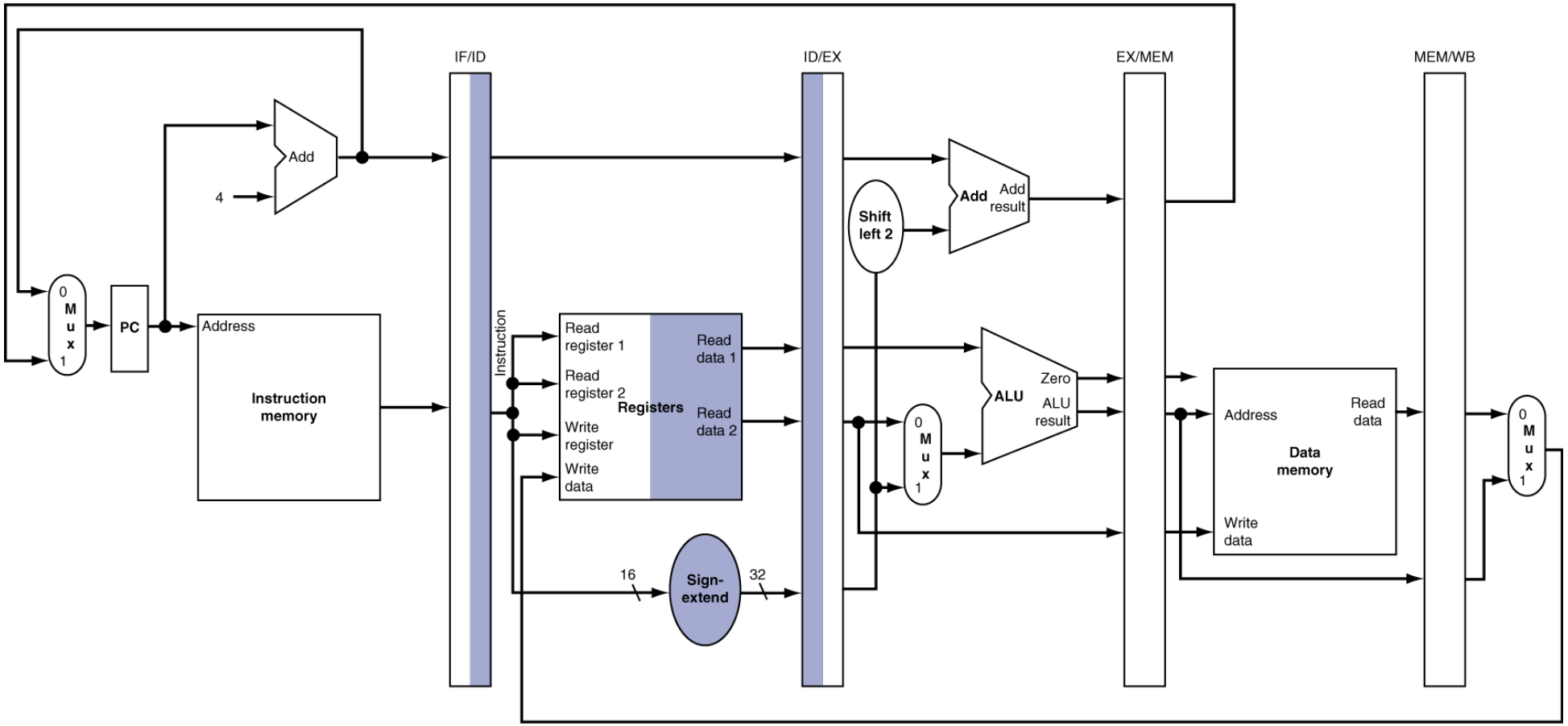


Instruction word and PC+4 are in the IF/ID pipeline register

ID for Load, Store, …



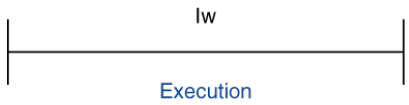
LW $4, 32($5)



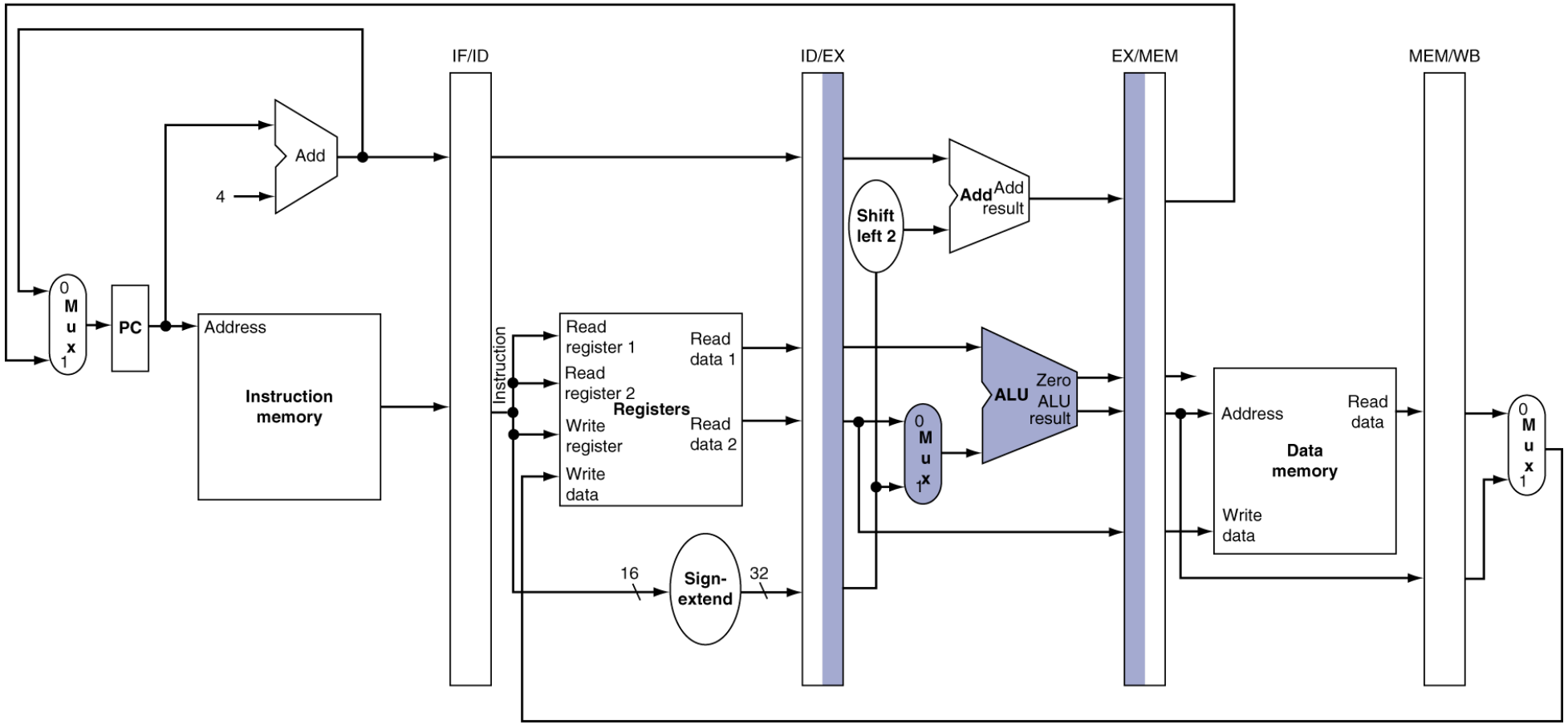
Value of $5, 32, and others are in ID|EX pipeline register

Similar info of the following instruction are now in IF|ID register

EX for Load



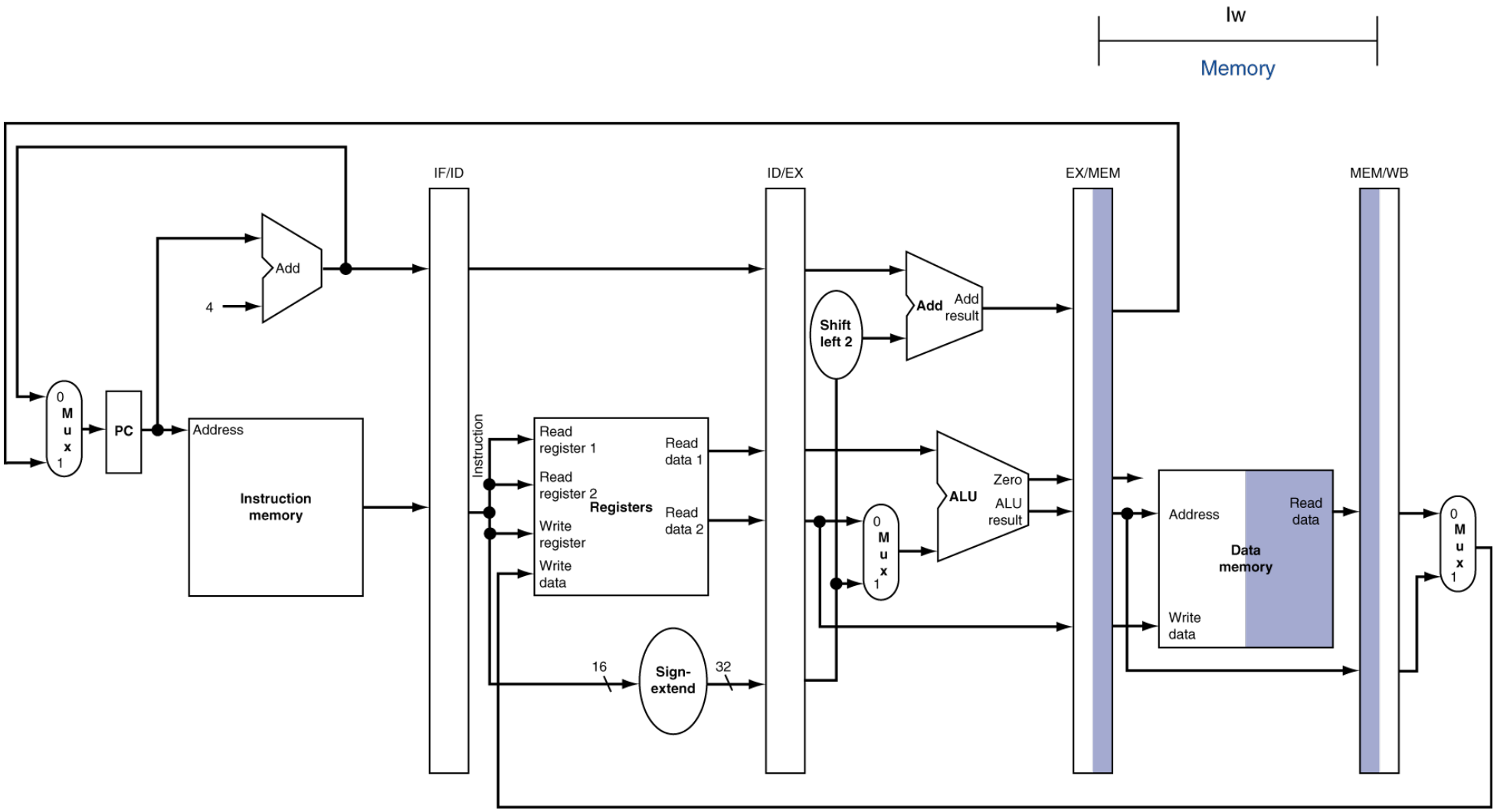
LW $4, 32($5)



Value of $5+32, and others are in EX|MEM pipeline register

MEM for Load

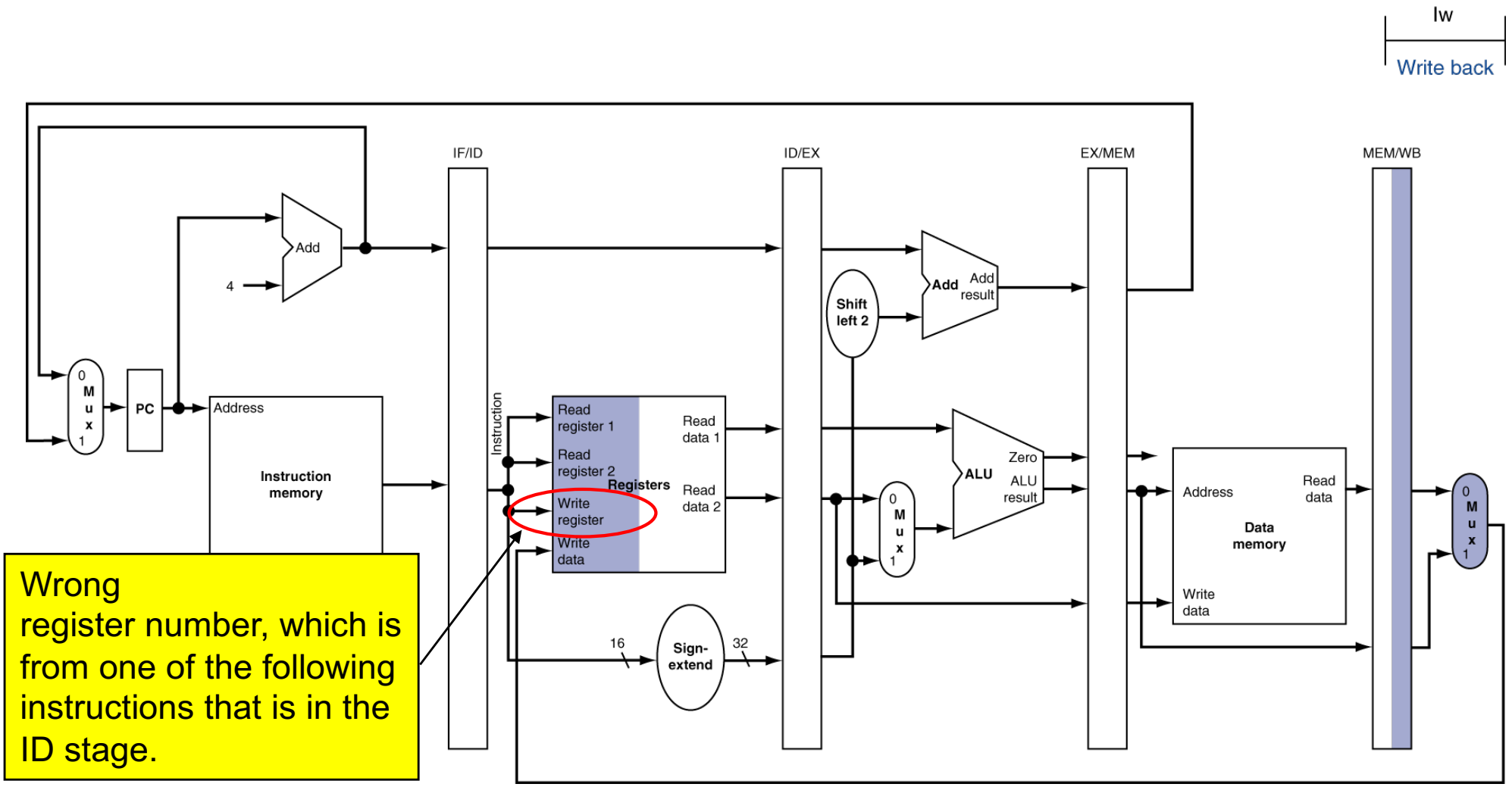
LW $4, 32($5)



Value of MEM[$5+32], and others are in MEM|WB pipeline register for WB

WB for Load

LW $4, 32($5)

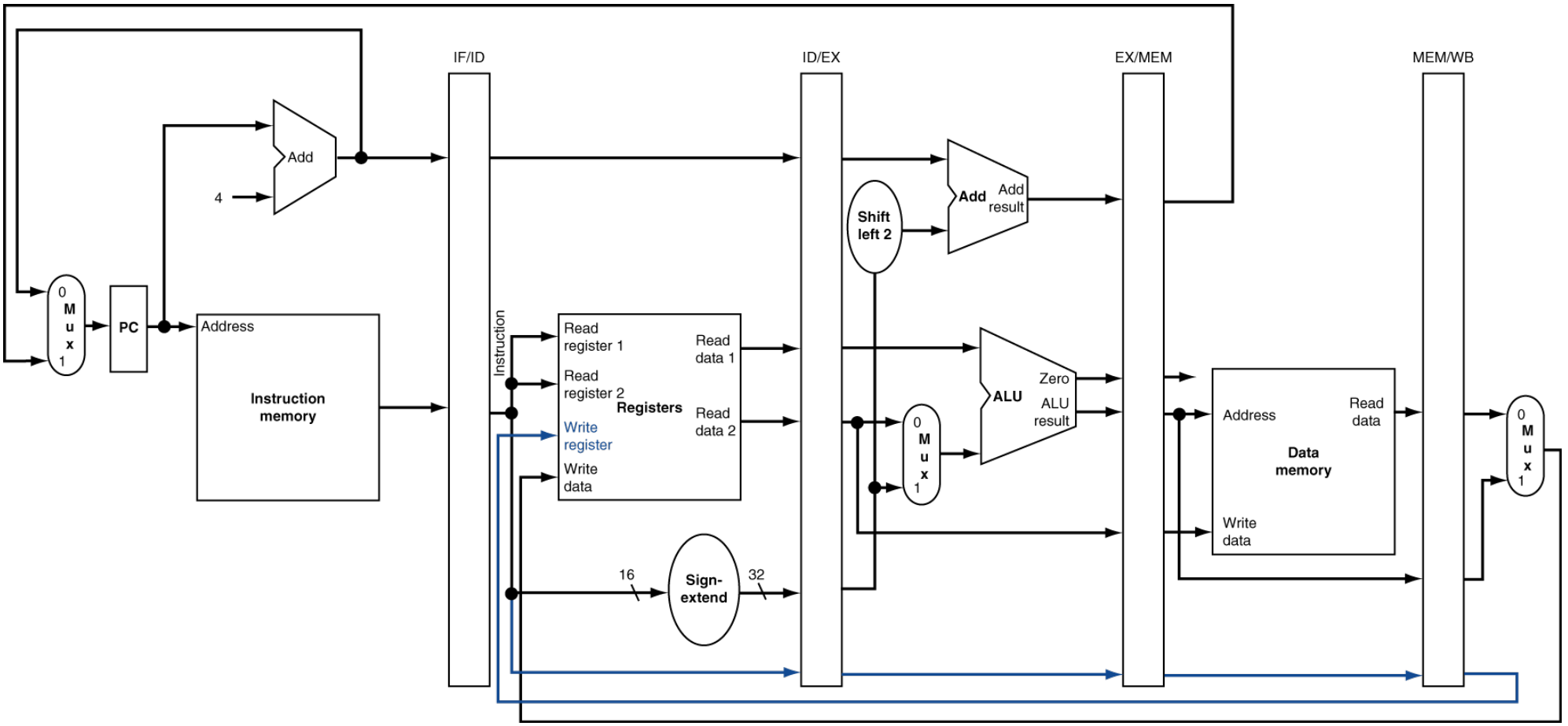


Wrong register number, which is from one of the following instructions that is in the ID stage.

Value of MEM[$5+32], and others are in MEM|WB pipeline register for WB

Corrected Datapath for Load

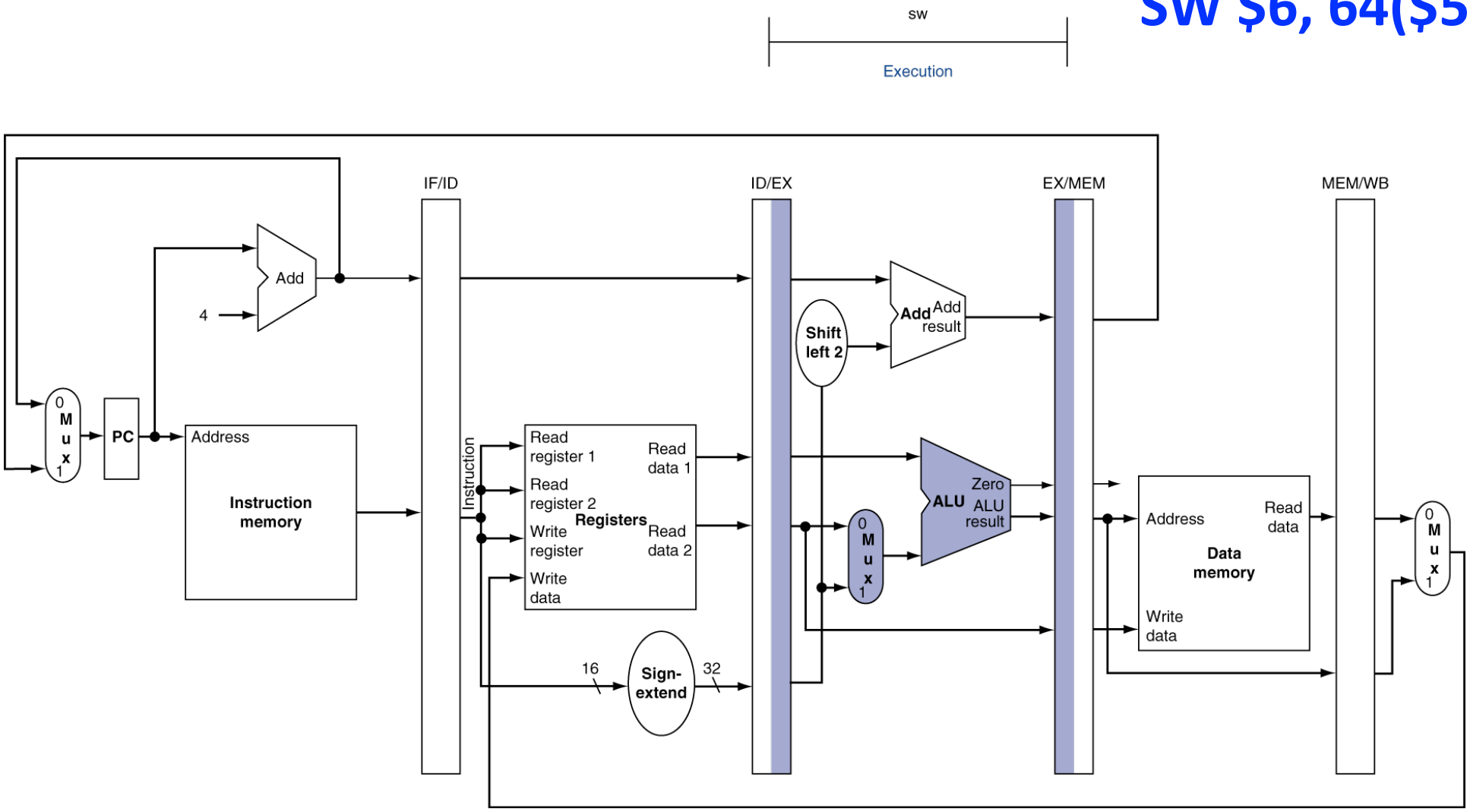
LW $4, 32($5)



LW completes and exits from the pipeline.

EX for Store

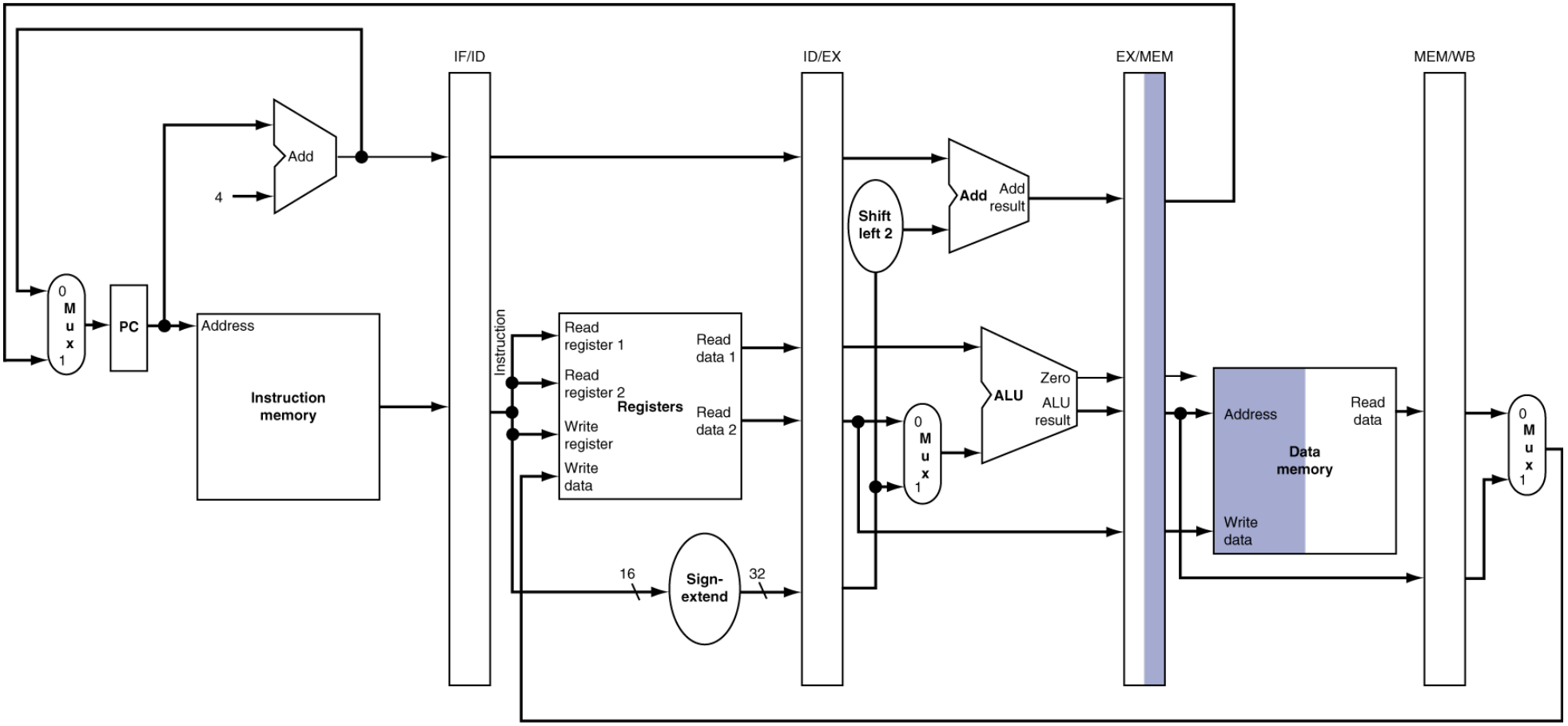
SW $6, 64($5)



Value of $5+64 and $6 are in EX|MEM pipeline register

MEM for Store

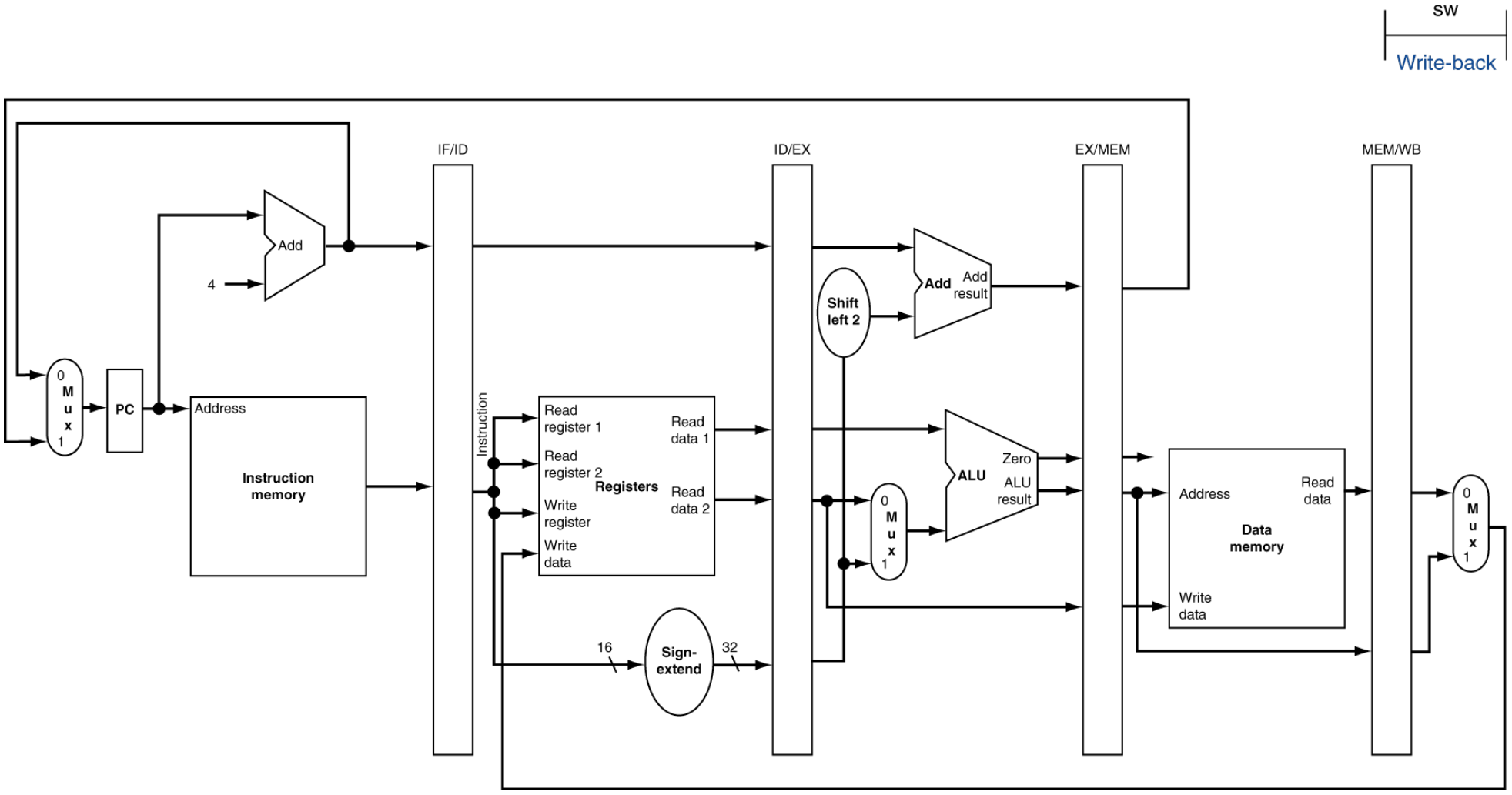
SW $6, 64($5)



$6 is written to MEM[$5+64]

WB for Store

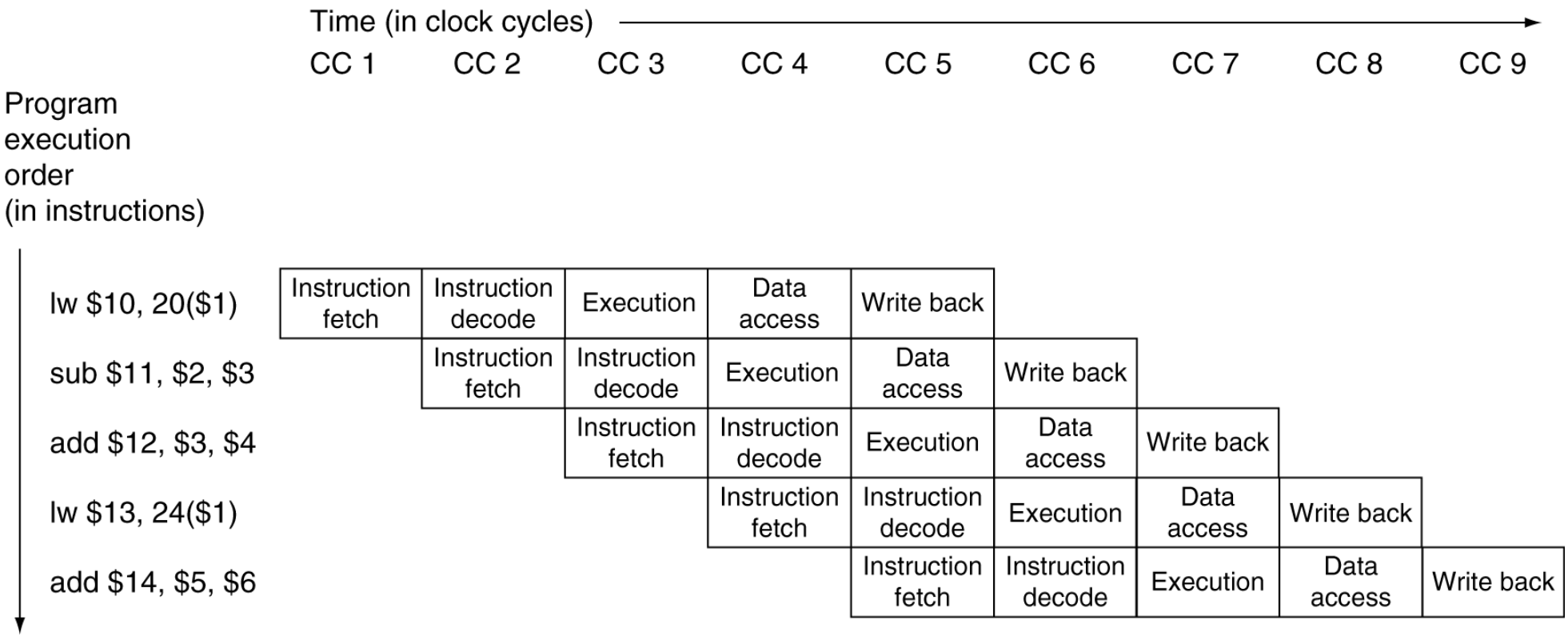
SW $6, 64($5)



Nothing to do for SW in WB stage and SW completes

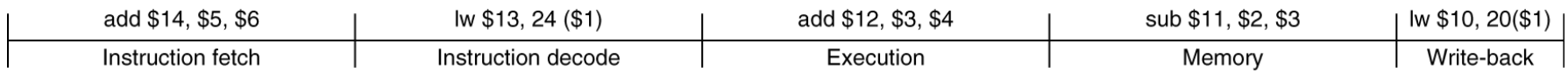
Multi-Cycle Pipeline Diagram

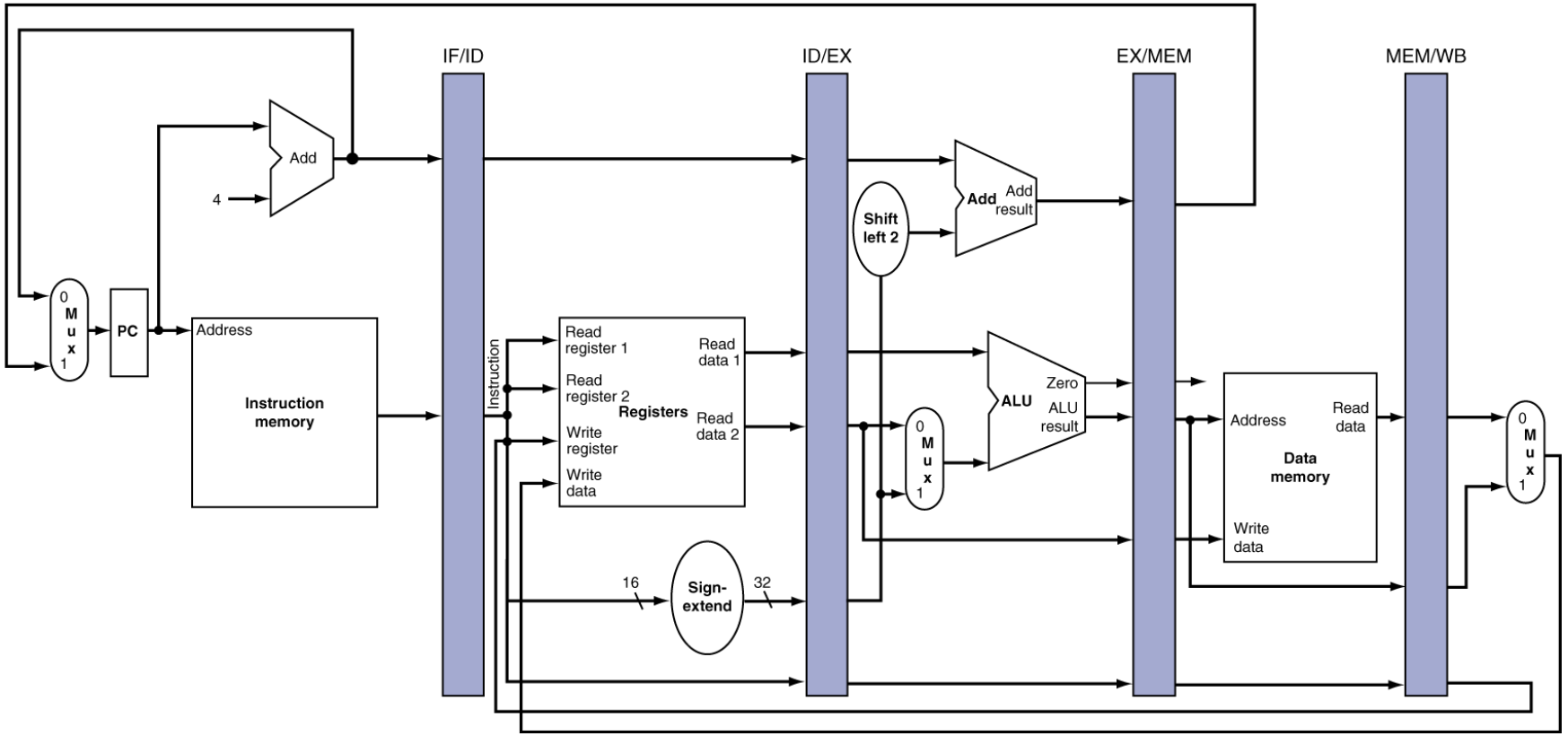
- Traditional form



Single-Cycle Pipeline Diagram

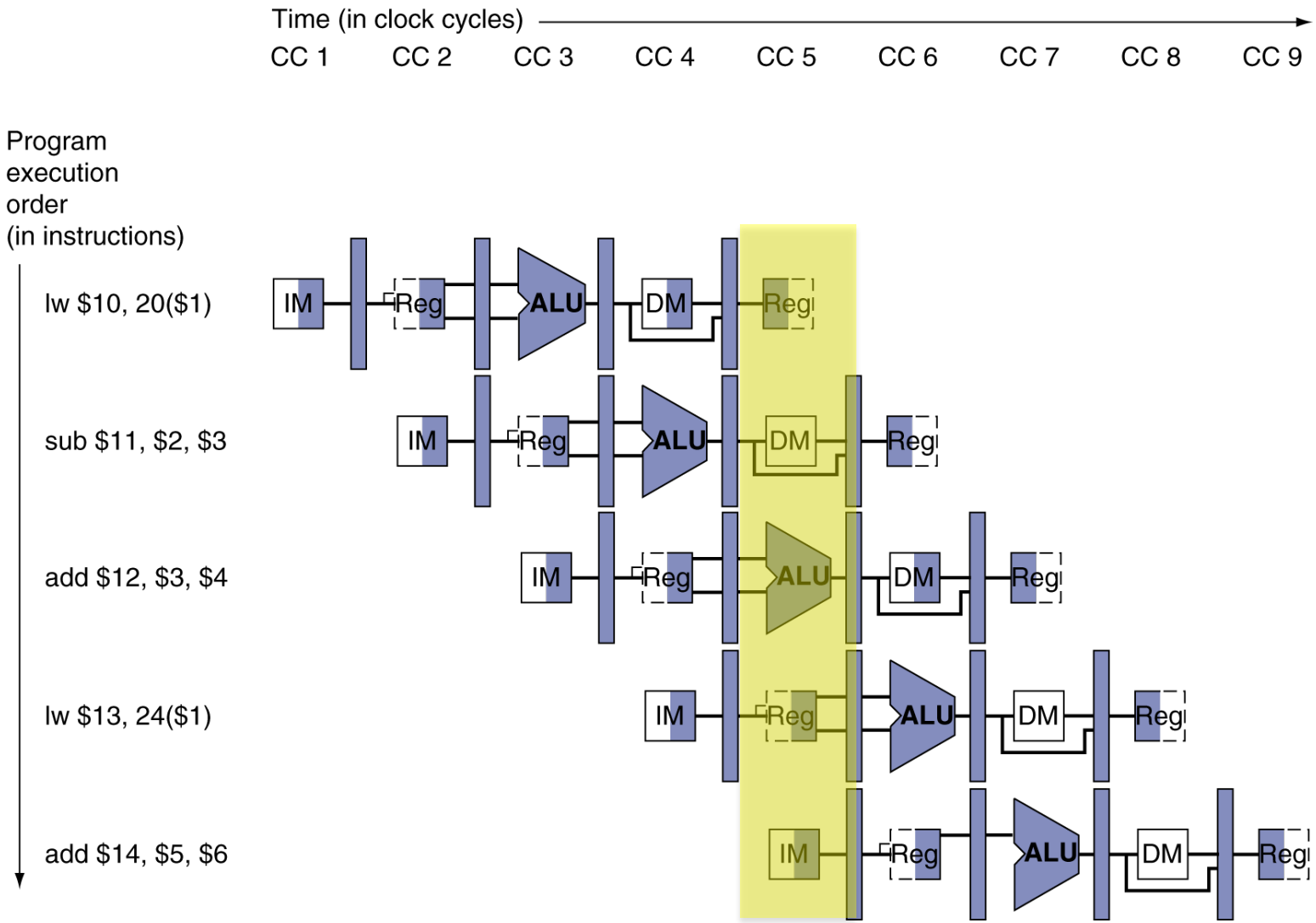
- State of pipeline in a given cycle



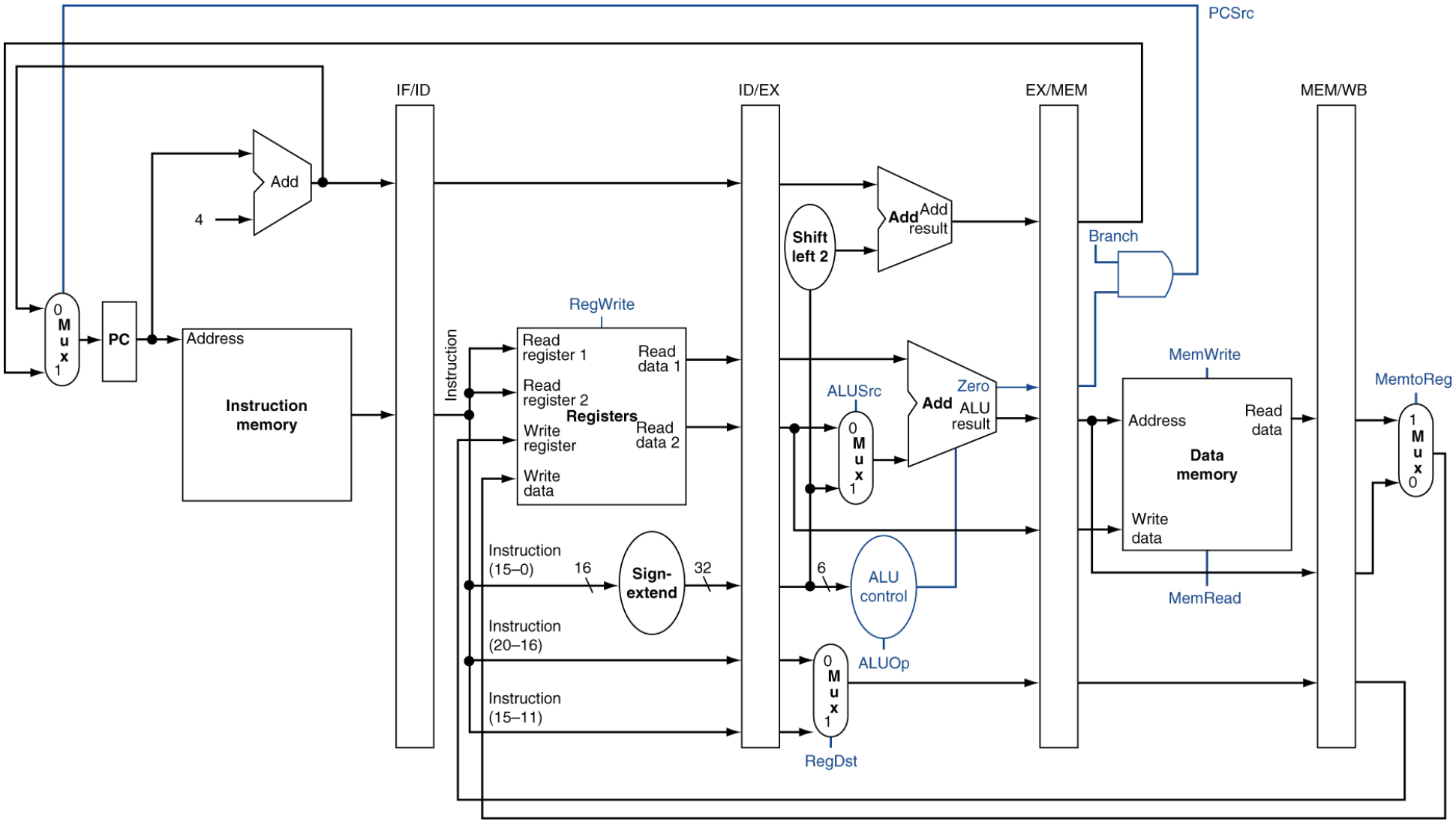


Multi-Cycle Pipeline Diagram

- Form showing resource usage

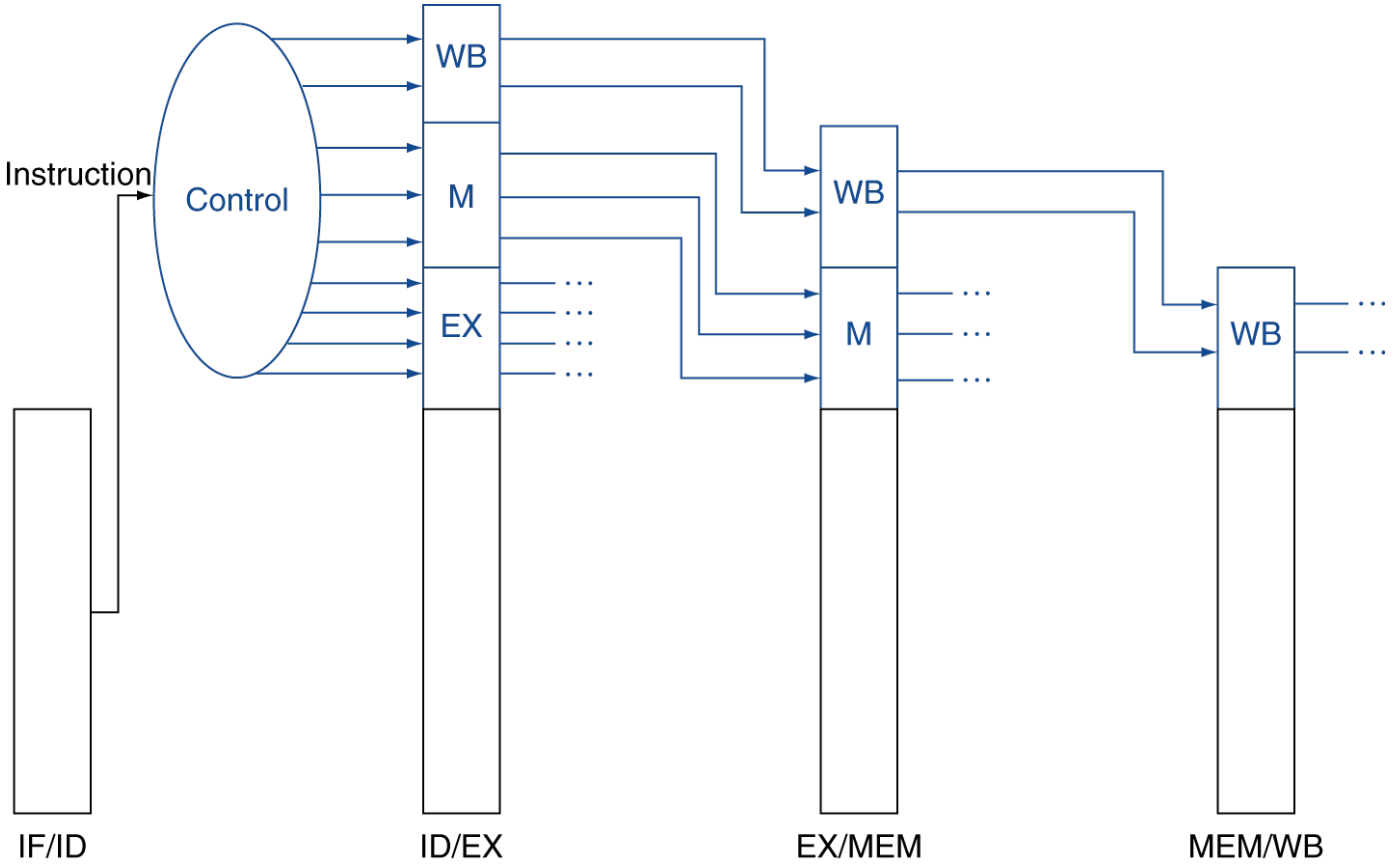


Pipelined Control (Simplified)

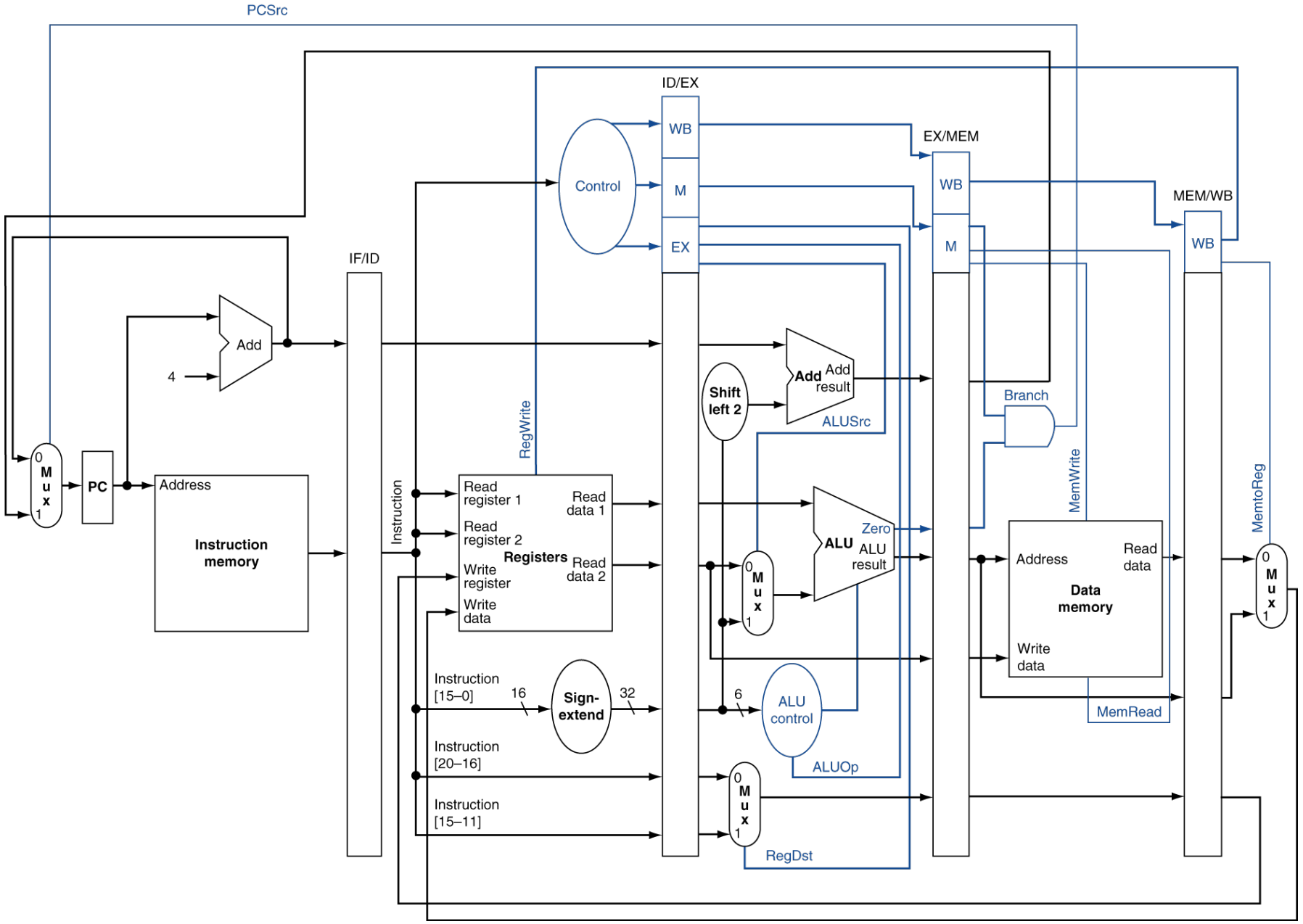


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



Data Hazards in ALU Instructions

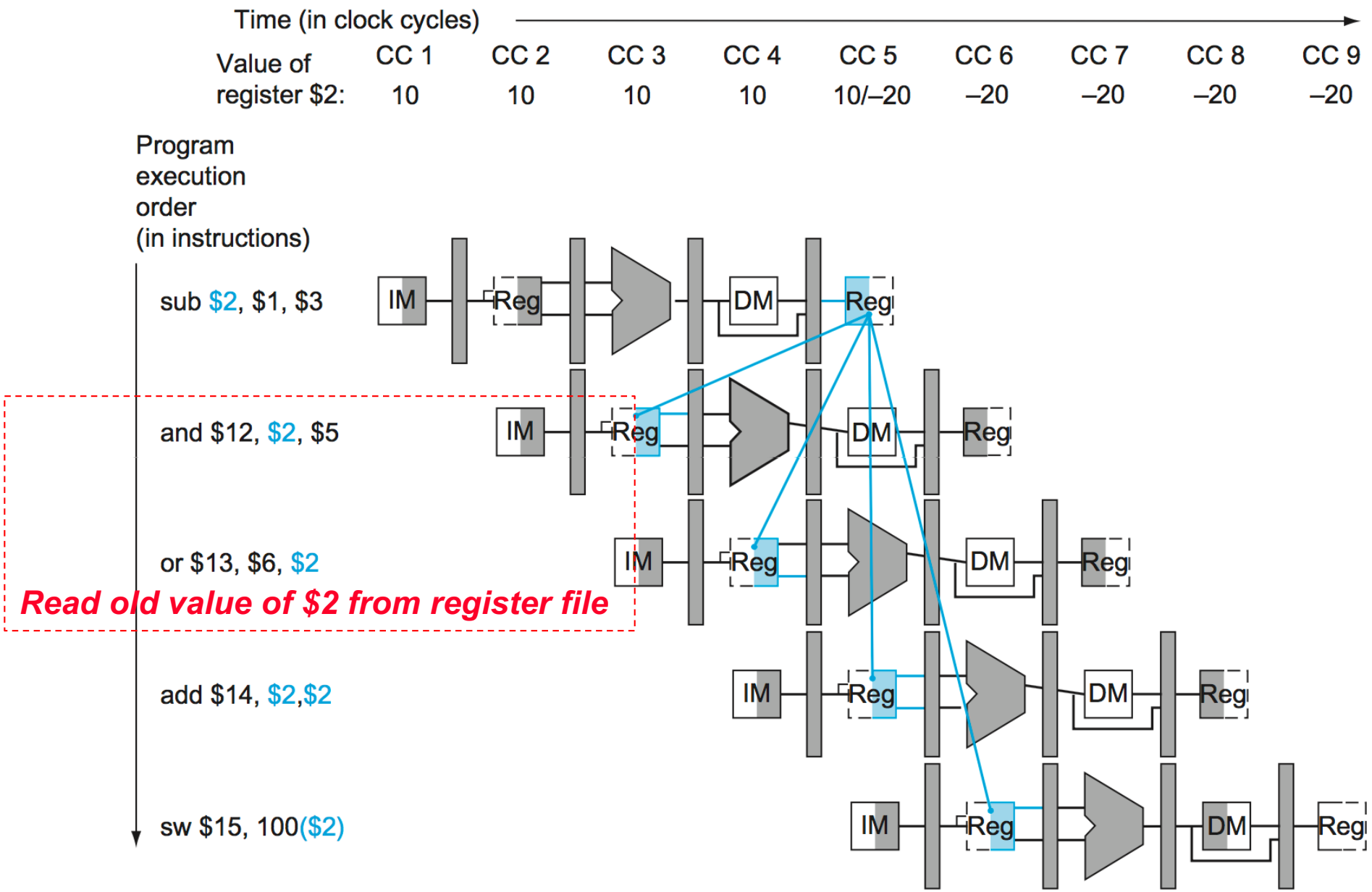
- Consider this sequence:

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

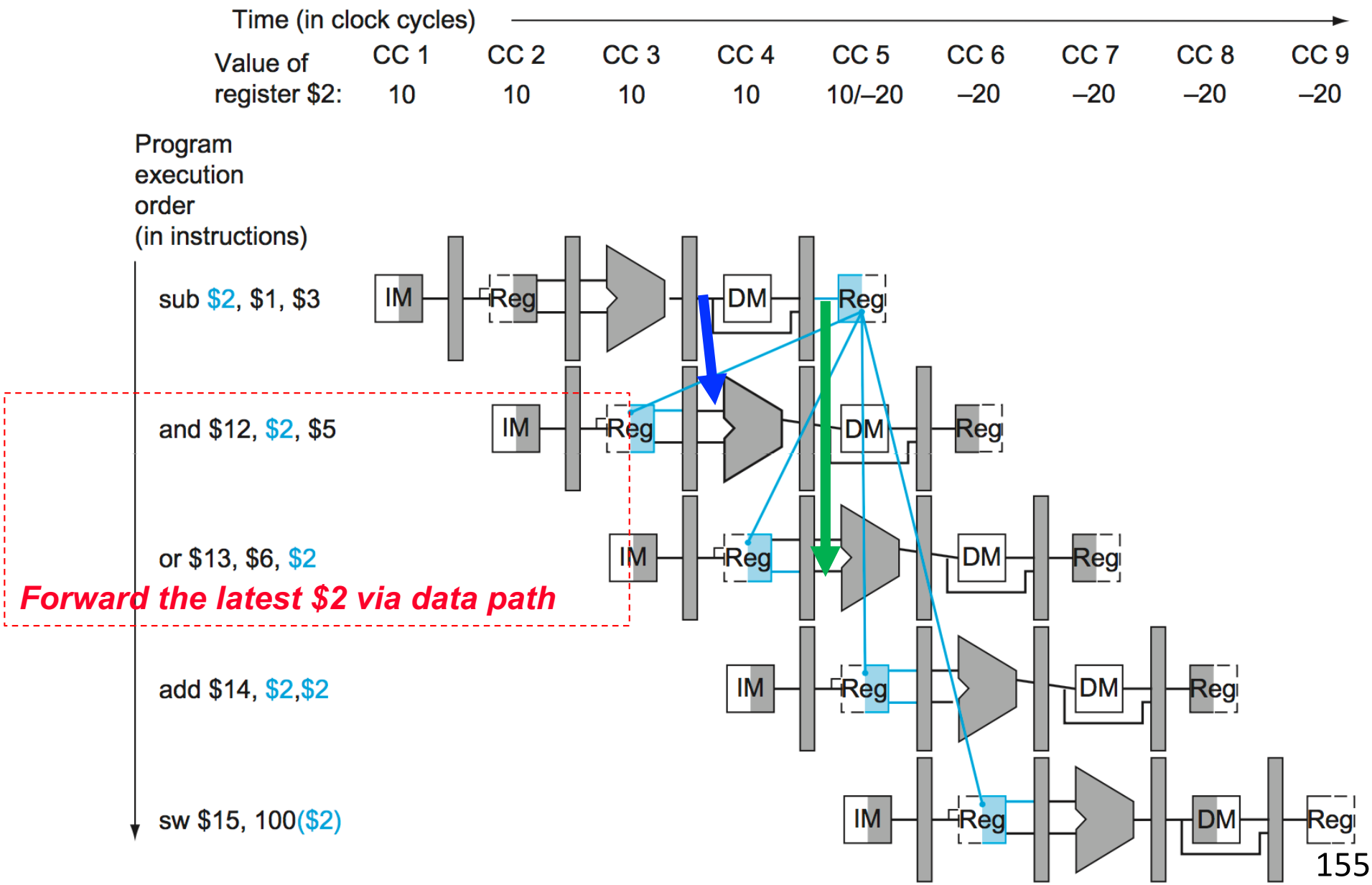
- Called Read After Write (RAW) hazards

- We can resolve hazards with forwarding
  - How do we detect when to forward?

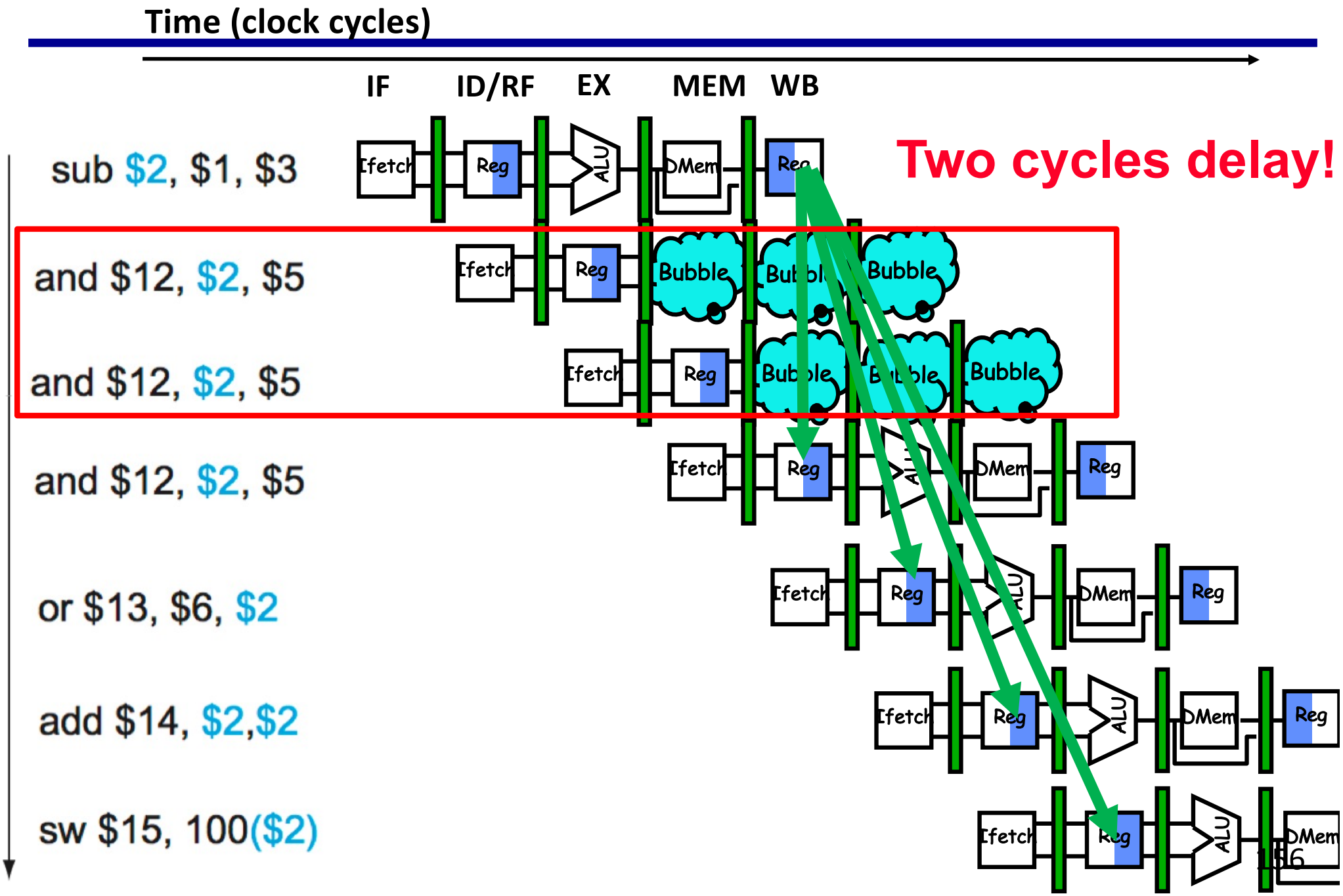
Data Dependency → Data Hazards



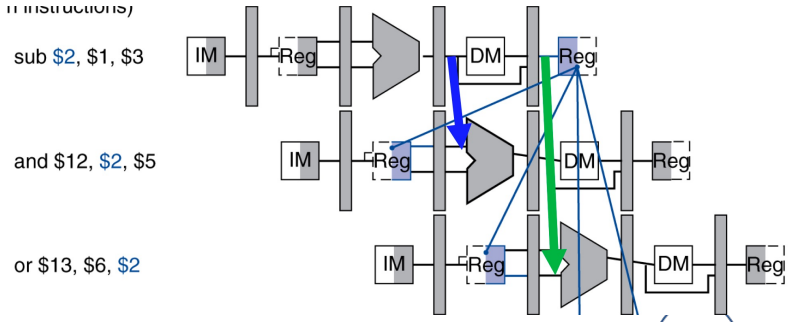
Solution #1: Handling RAW Hazards by Forwarding



Solution #2: Insert stalls



Datapath for Forwarding

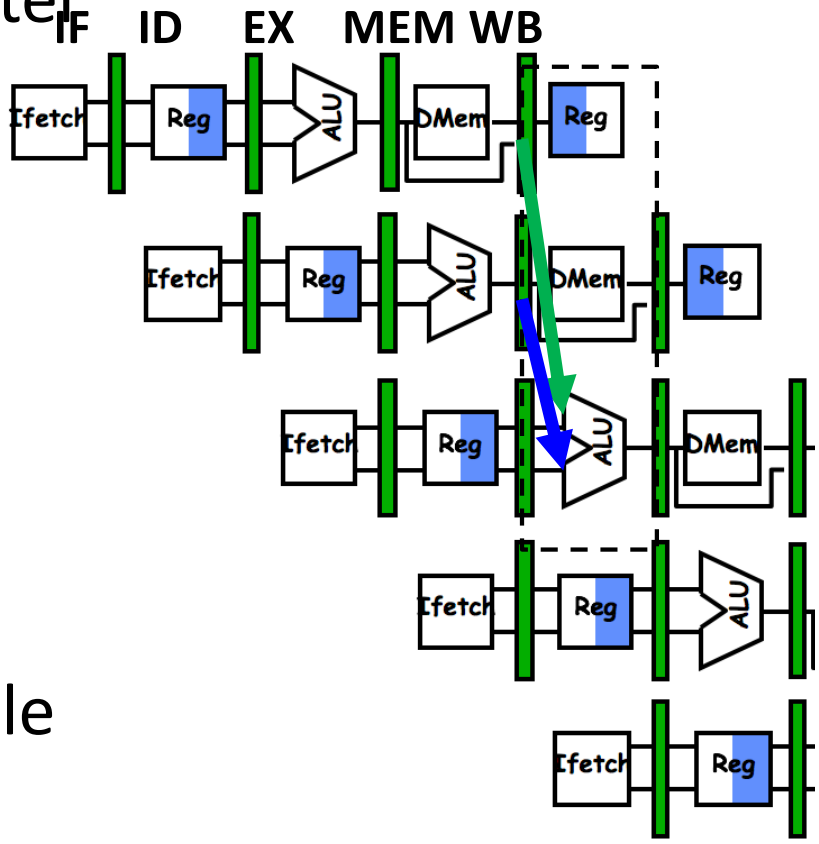


- Forwarding happens in two consecutive cycles

Detecting RAW Hazards

- Current instruction being executed in ID/EX register
- Previous instruction is in the EX/MEM register
- 2nd Previous is in the MEM/WB register

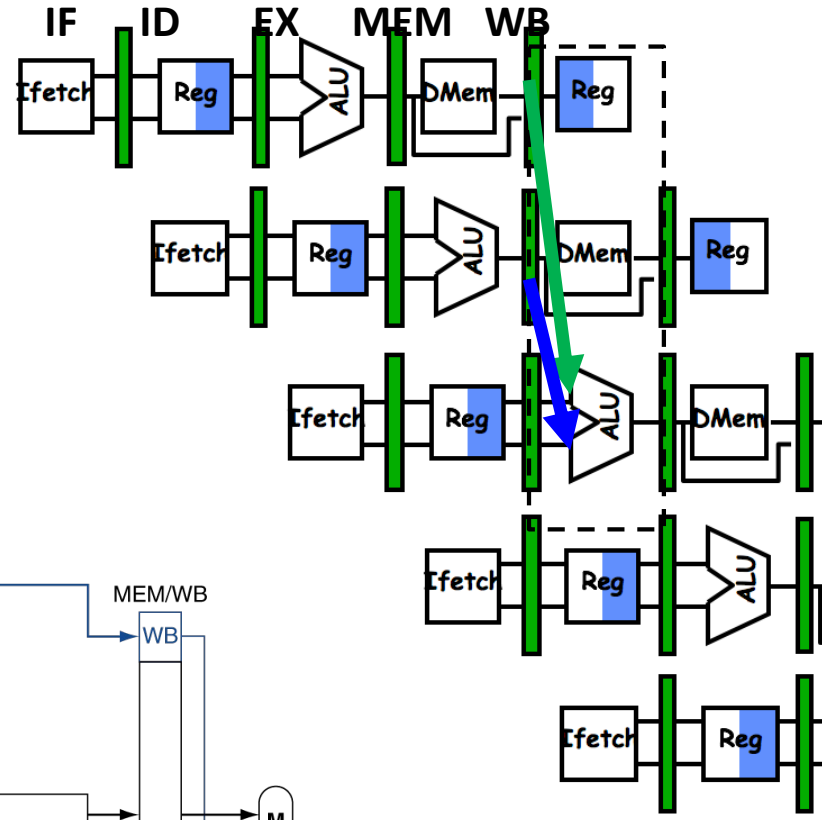
```
ADD R1, R2, R3  #2nd Previous in MEM/WB
SUB R6, R4, R5  #Previous in EX/MEM
AND R7, R1, R6  #Current in ID/EX
```

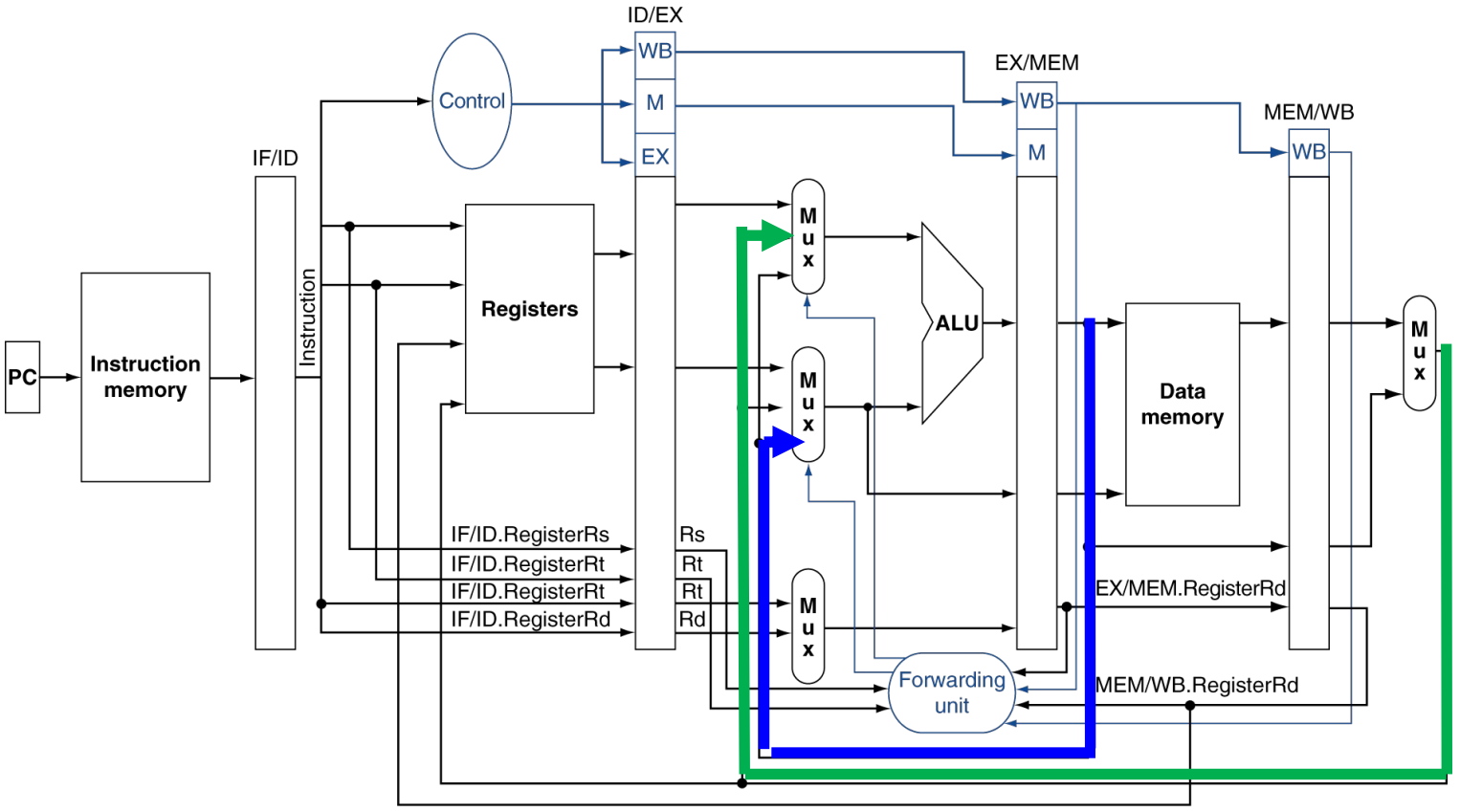


- Forwarding happens in the same cycle

Detecting RAW Hazards

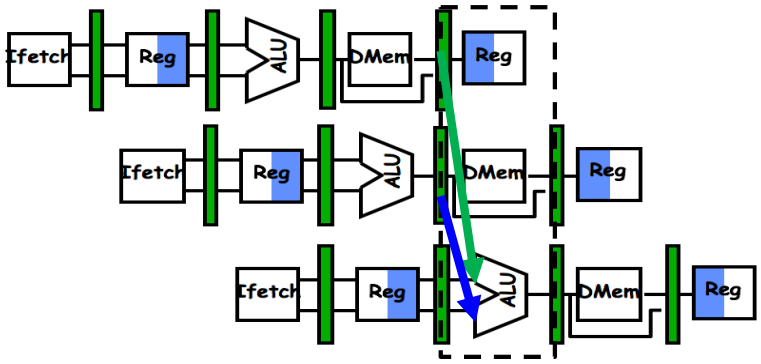
```
ADD  R1, R2, R3  #2nd Previous in MEM/WB
SUB  R6, R4, R5  #Previous in EX/MEM
AND  R7, R1, R6  #Current in ID/EX
```





Detecting RAW Hazards

```
ADD R1, R2, R3   #2nd Previous in MEM/WB
SUB R6, R4, R5   #Previous in EX/MEM
AND R7, R1, R6   #Current in ID/EX
```



- Pass register numbers along pipeline
  - ID/EX.RegisterRs = register number for Rs in ID/EX (Rs1)
  - ID/EX.RegisterRt = register number for Rt in ID/EX (Rs2)
  - ID/EX.RegisterRd = register number for Rd in ID/EX

- RAW Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

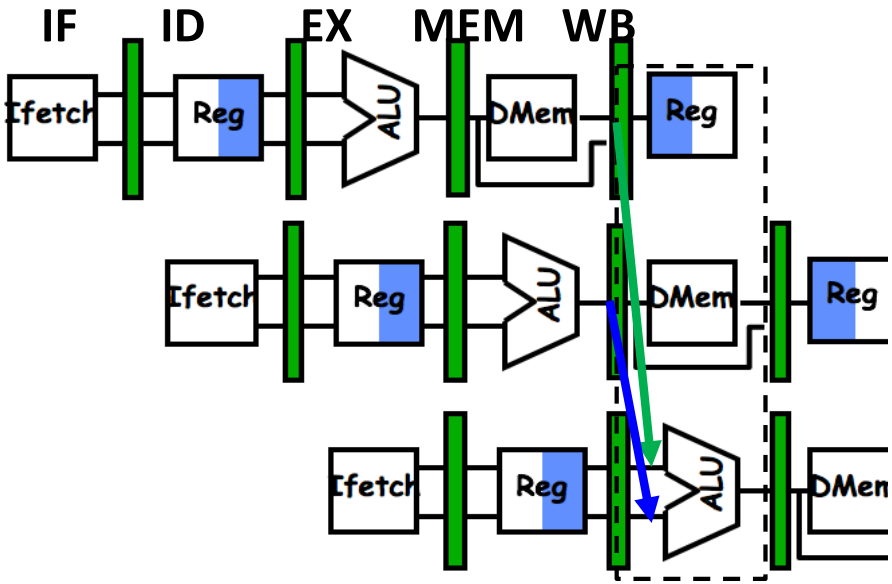
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

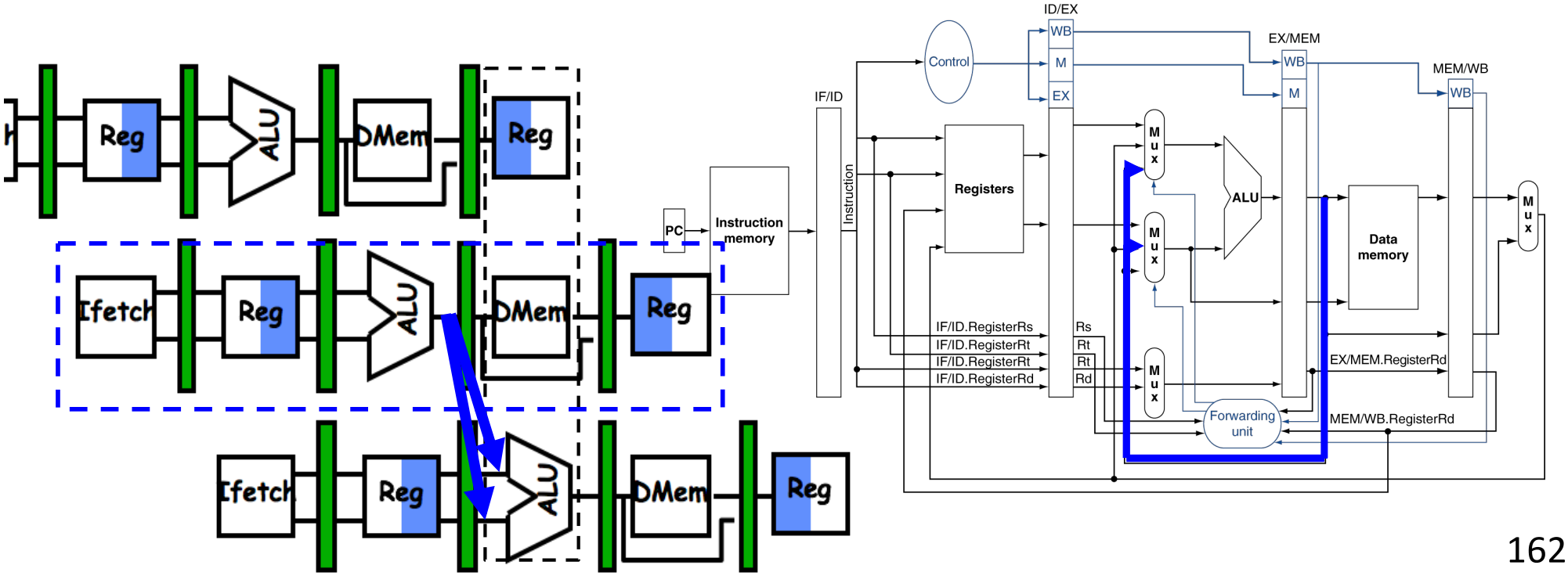
- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not R0
  - EX/MEM.RegisterRd ≠ 0
  - MEM/WB.RegisterRd ≠ 0

```
ADD R1, R2, R3   #2nd Previous in MEM/WB
SUB R6, R4, R5   #Previous in EX/MEM
AND R7, R1, R6   #Current in ID/EX
```



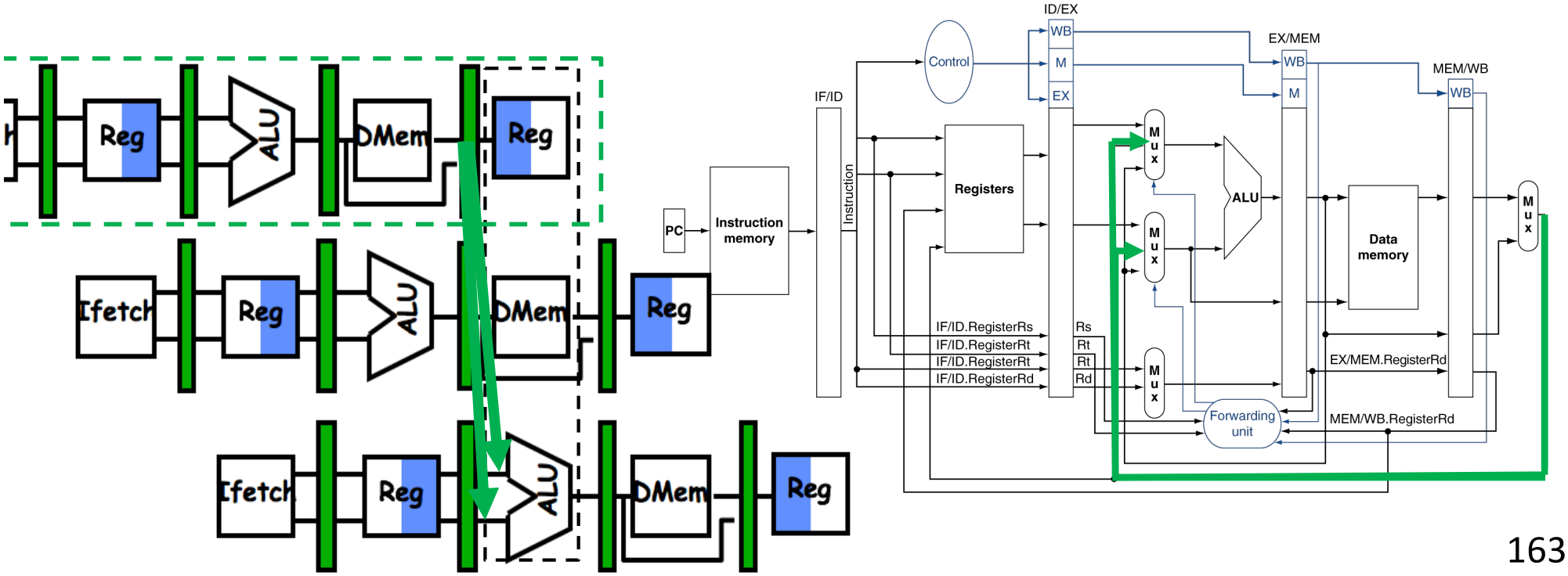
Forwarding Conditions

- Detecting RAW hazard with Previous Instruction
  - **if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))**
    ForwardA = 01 (Forward from EX/MEM pipe stage)
  - **if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))**
    ForwardB = 01 (Forward from EX/MEM pipe stage)



Forwarding Conditions

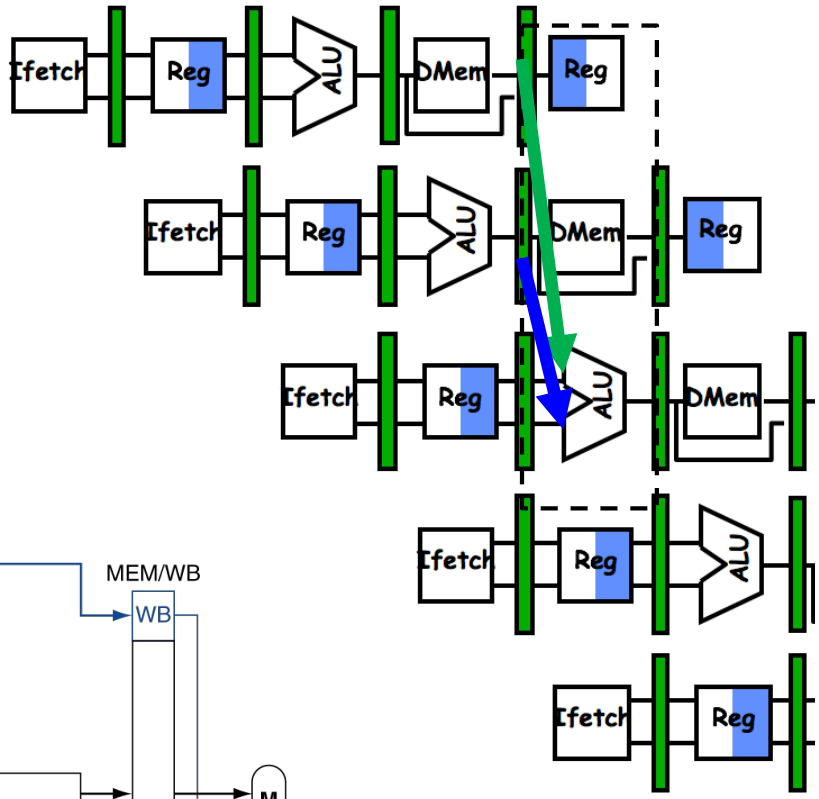
- Detecting RAW hazard with Second Previous
  - **if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))**
    ForwardA = 10 (Forward from MEM/WB pipe stage)
  - **if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))**
    ForwardB = 10 (Forward from MEM/WB pipe stage)

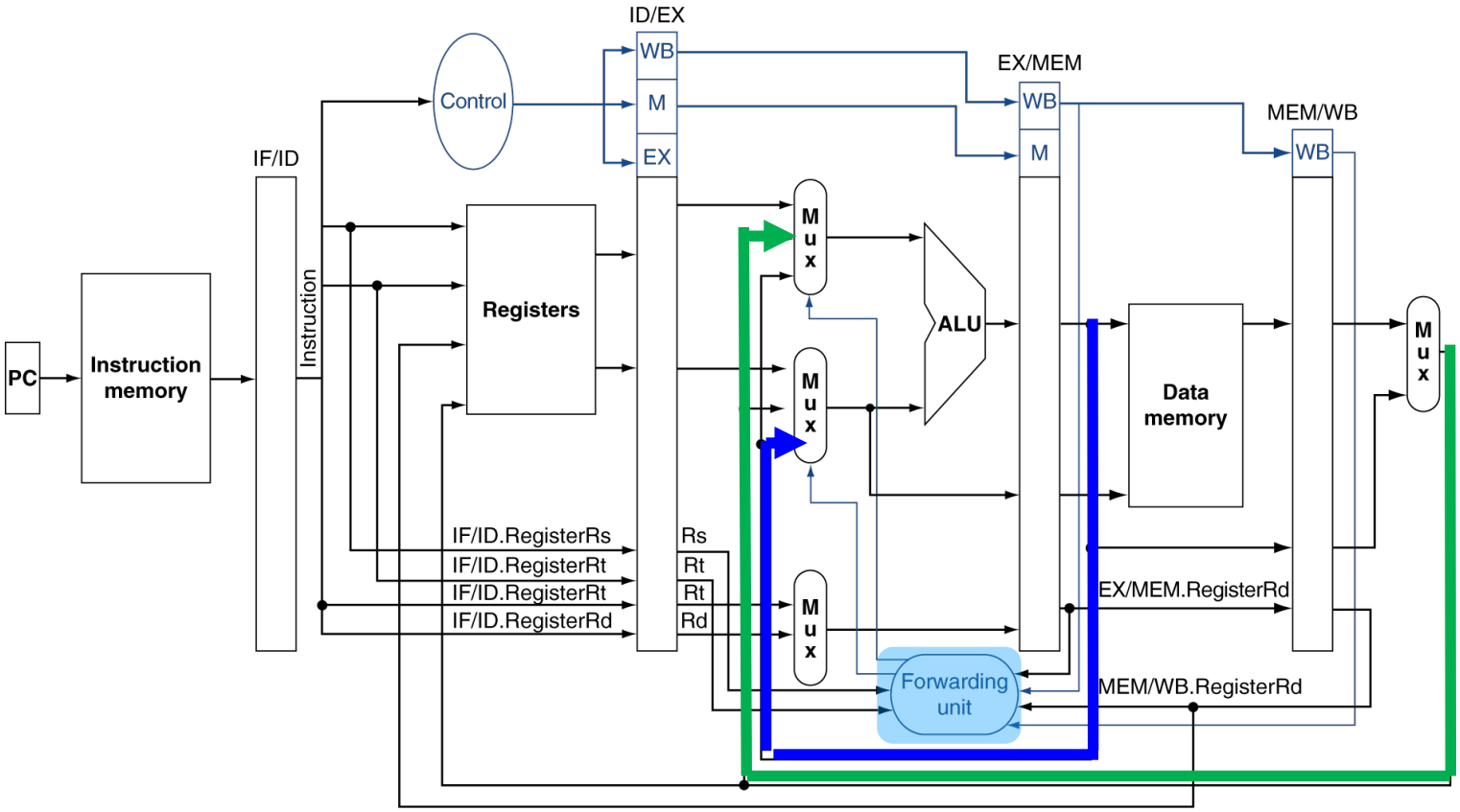


Control Signals During Forwarding:

Those Light Blue lines

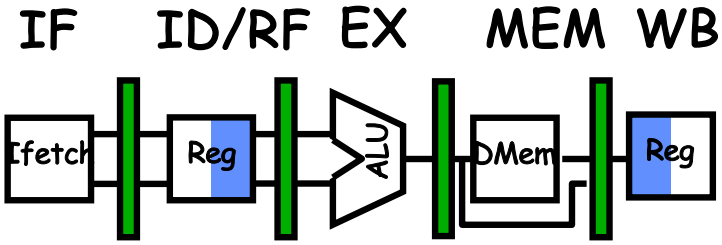
```
ADD  R1, R2, R3  #2nd Previous in MEM/WB
SUB  R6, R4, R5  #Previous in EX/MEM
AND  R7, R1, R6  #Current in ID/EX
```





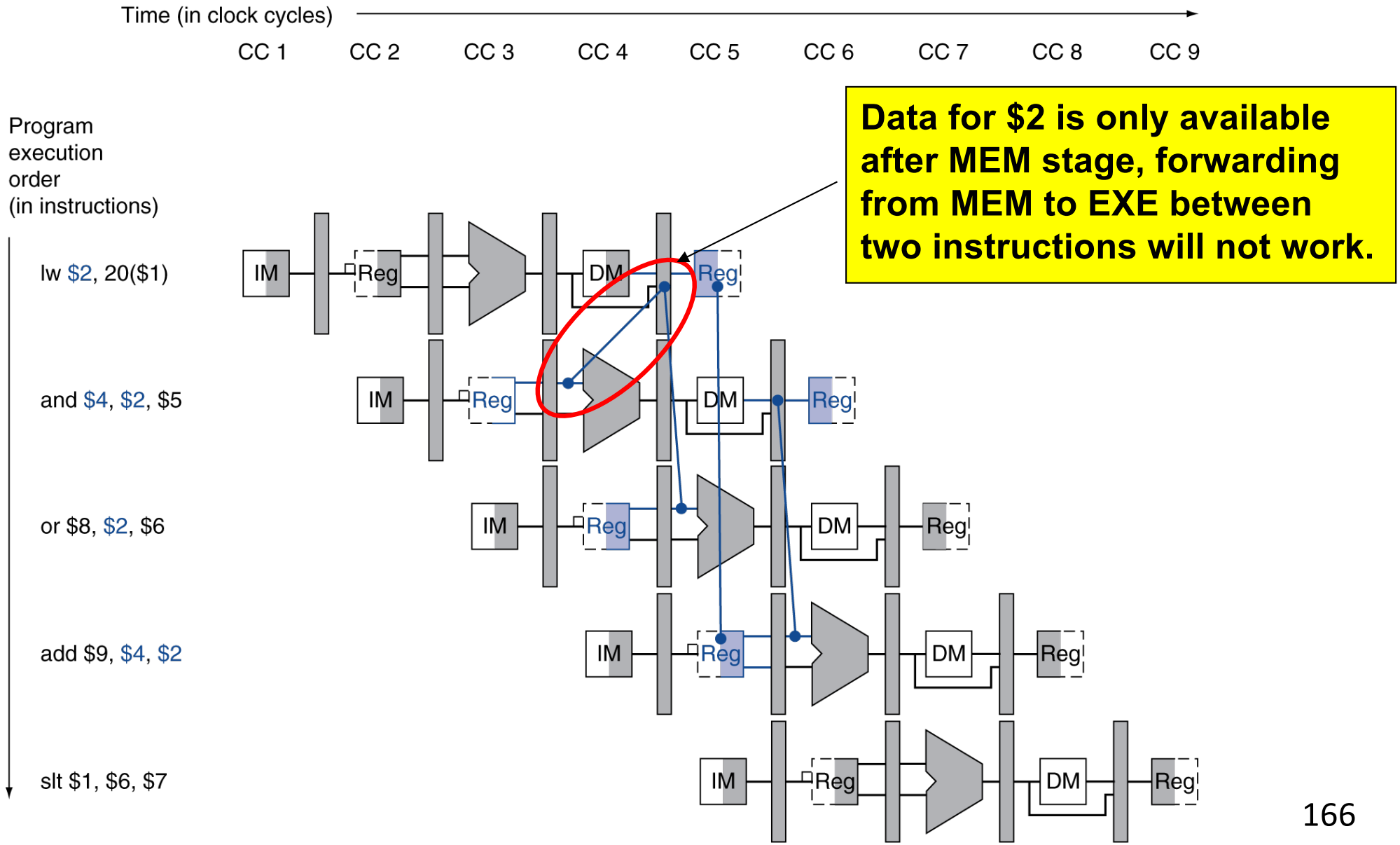
RAW Hazards with Load/Store

- **LW Rt, 20(Rs): load a word from memory @ [Rs]+20 into Rt**
  - ID/RF: Read register Rs: [Rs] (rs select)
  - EX: Calculate effective address: [Rs] + 20
  - **MEM: Memory read from [Rs]+20**
    - **Data is available in MEM|WB**
    - **Unlike ALU: data is available in EX|MEM**
  - WB: data write back to Rt (rt select)

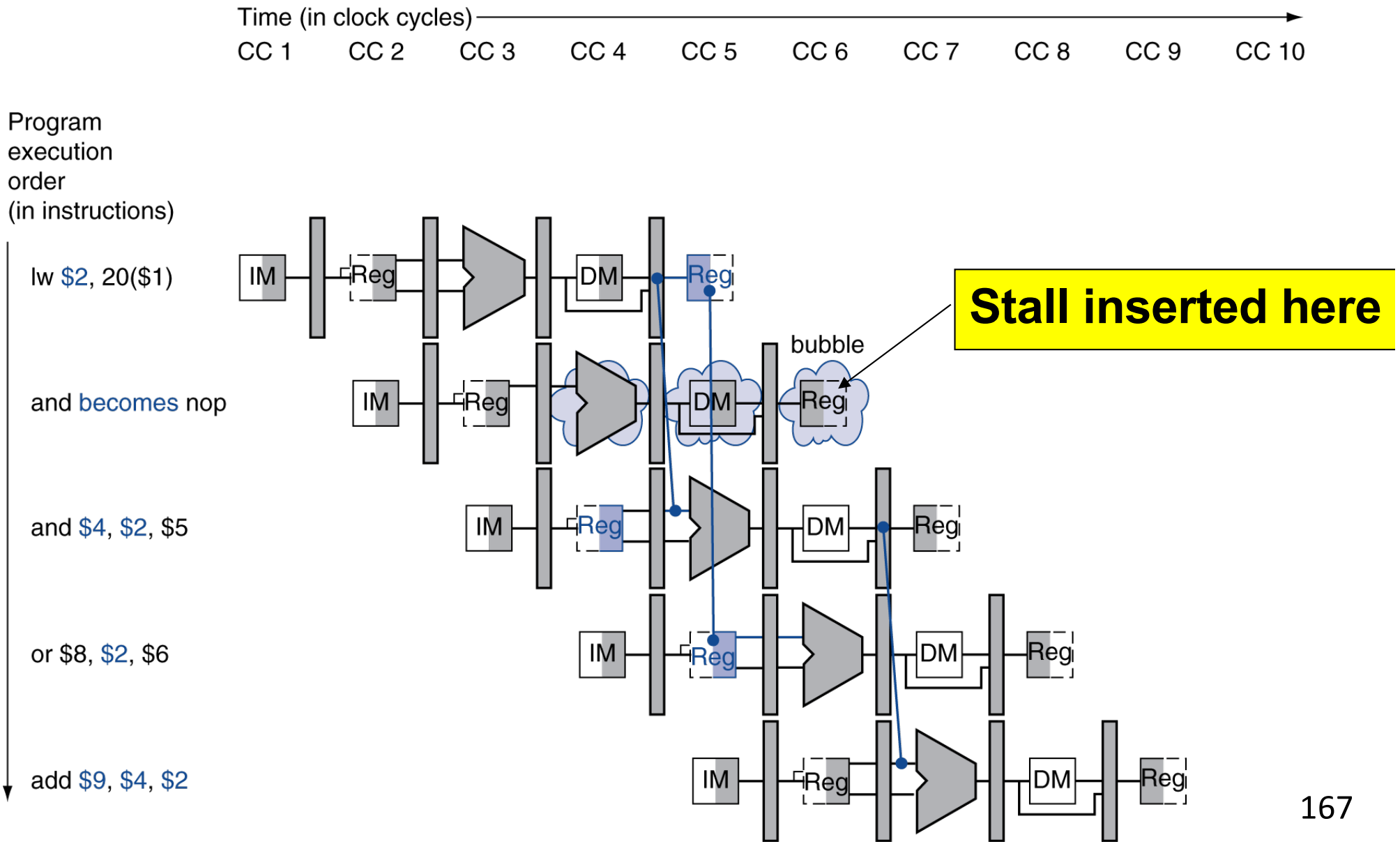


- **SW Rt,12(Rs): store a word in Rt in the memory @ [Rs]+12**
  - ID/RF: Read register Rs and Rt (rs and rt select, no rd)
    - Rs is needed in EX, and **Rt is needed in MEM**
  - EX: Calculate effective address: [Rs] + 12
  - **MEM: Memory write to [Rs]+12**
    - **Need Rt to be available**
    - **Unlike ALU, data needs to be available in ID|EX**
  - No need WB

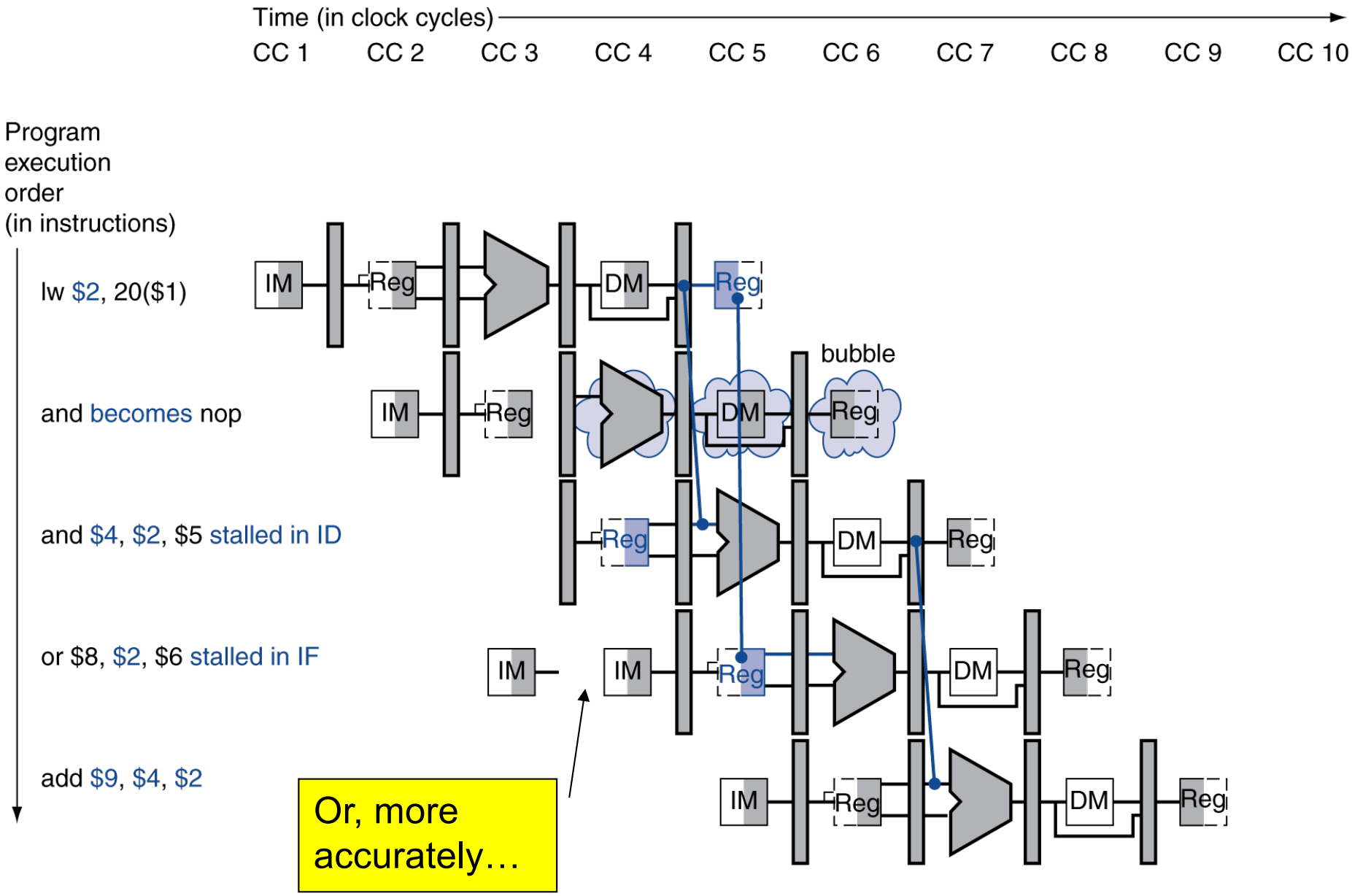
Load-Use RAW Data Hazard



Stall/Bubble in the Pipeline

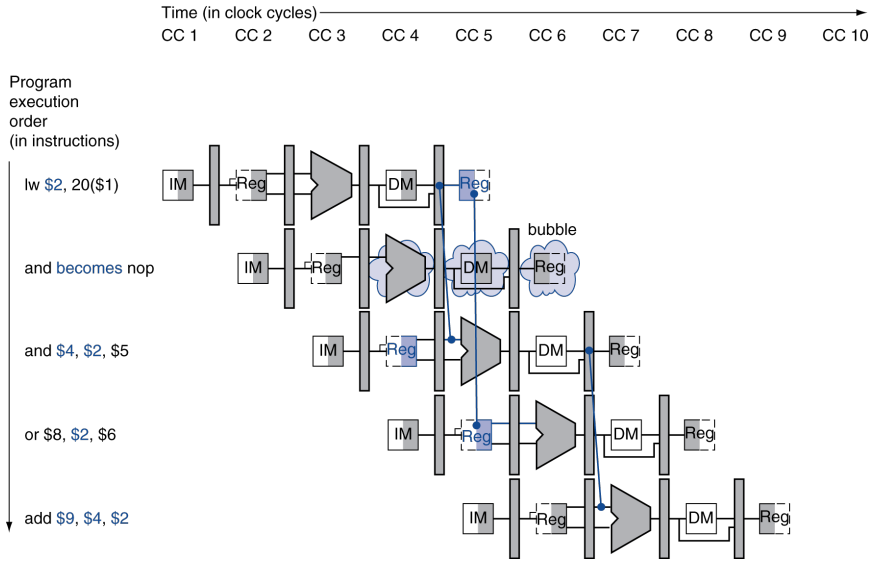


Stall/Bubble in the Pipeline



Load-Use Hazard Detection

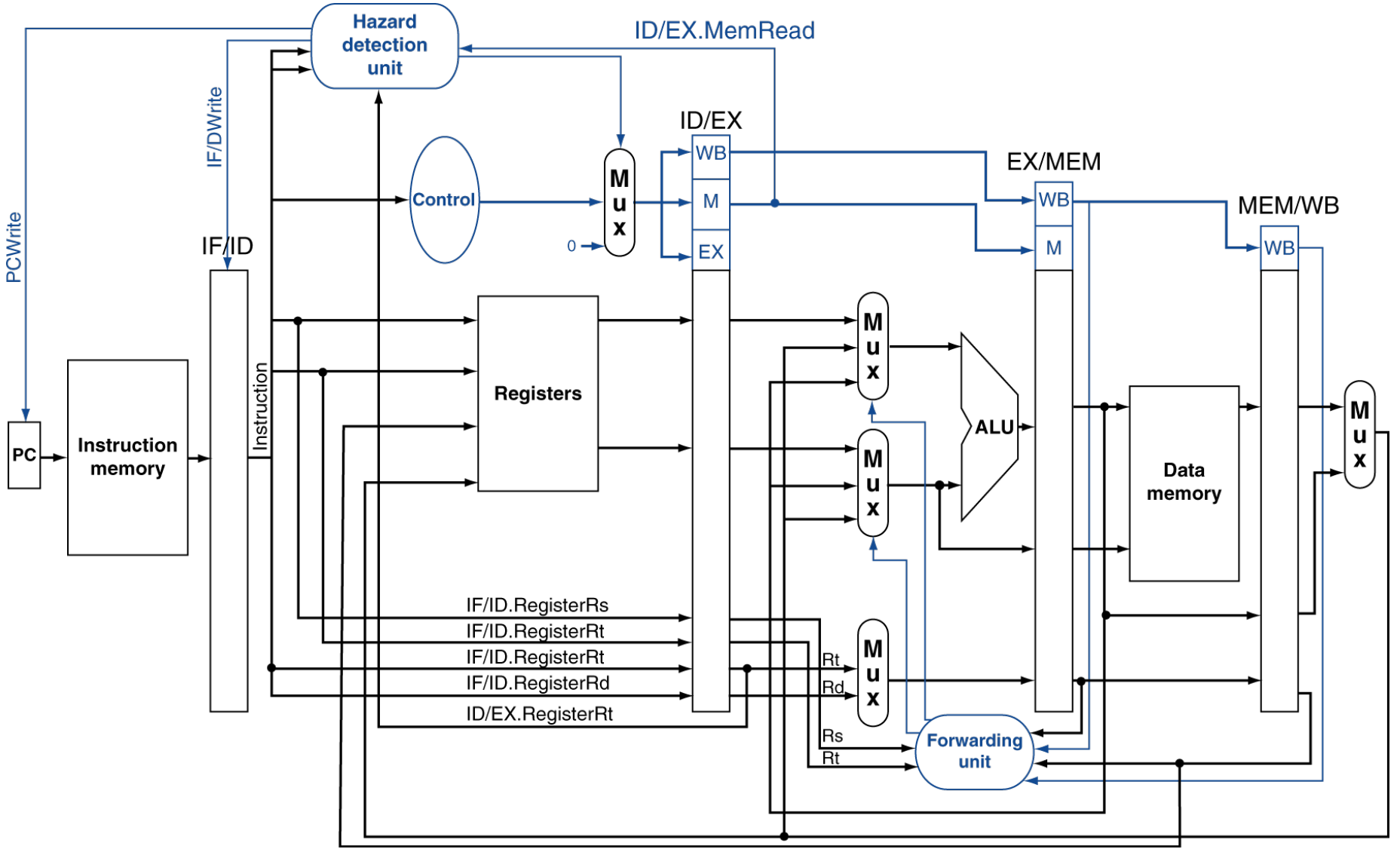
- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble



How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

Datapath with Hazard Detection



Compiler Scheduling for Removing Load-Use Stall

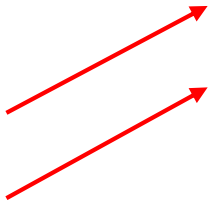
- Compilers can schedule code in a way to avoid load → ALU-use stalls

  a = b + c; d = e – f;

- Slow code: 2 stall cycles

```
lw   r10, (r1) # r1 = addr b
lw   r11, (r2) # r2 = addr c
                # stall
add  r12, r10, r11   # b + c
sw   r12, (r3)       # r3 = addr a
lw   r13, (r4)       # r4 = addr e
lw   r14, (r5)       # r5 = addr f
                # stall
sub  r15, r13, r14   # e - f
sw   r15, (r6)       # r6 = addr d
```

Fast code: No Stalls



```
lw   r10, 0(r1)
lw   r11, 0(r2)
lw   r13, 0(r4)
lw   r14, 0(r5)
add  r12, r10, r11
sw   r12, 0(r3)
sub  r15, r13, r14
sw   r15, 0(r6)
```

Stalls and Performance

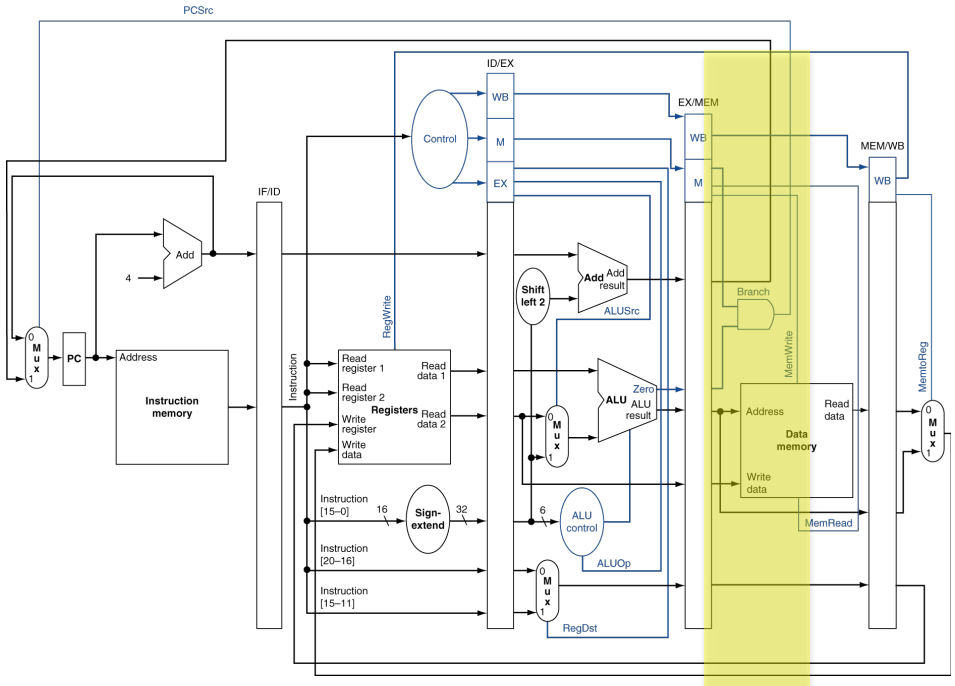
The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Control Hazards Because of Branches

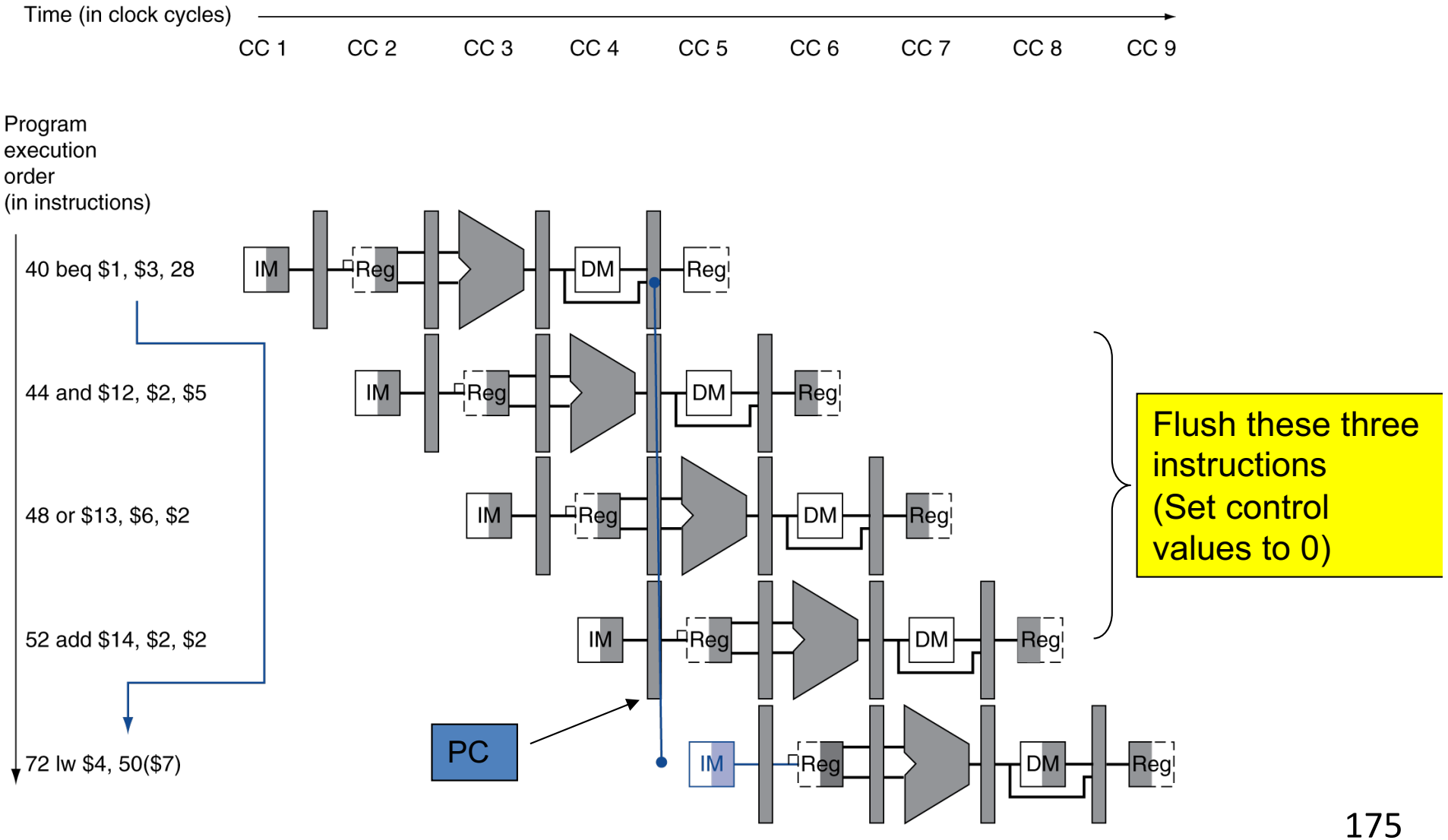
- Branch outcome determined in MEM

```
40        beq   $1,    $3, 28
44        and   $12, $2, $5
48        or    $13, $6,  $2
52        add   $14, $2, $2
...
72        lw    $4,  50($7)
```



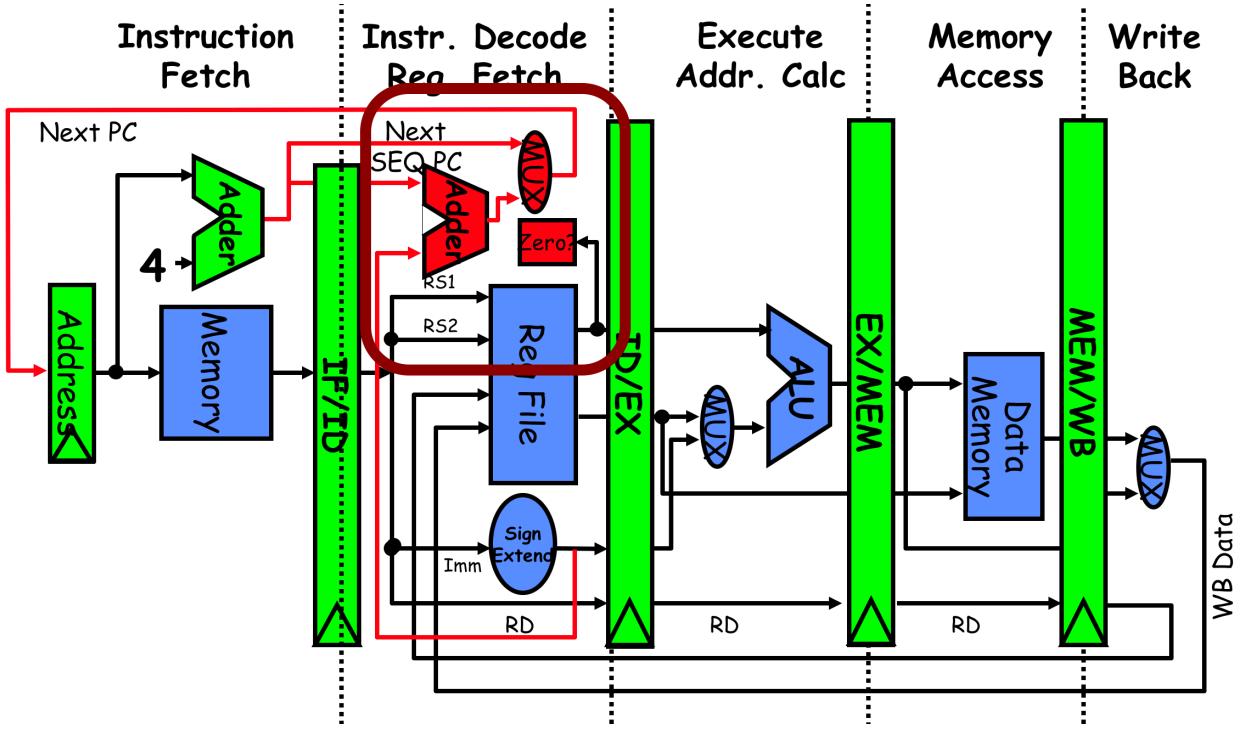
Control Hazards

- Branch outcome determined in MEM



Reducing Branch Delay

- In general, branch could cause 3 cycle delay
  - Since branch outcome is determined at MEM stage
- Move hardware to determine outcome at ID stage → 1 cycle delay
  - BEQZ instruction



  - For BEQ: add target address adder and Register comparator in the ID stage

Reducing Branch Delay

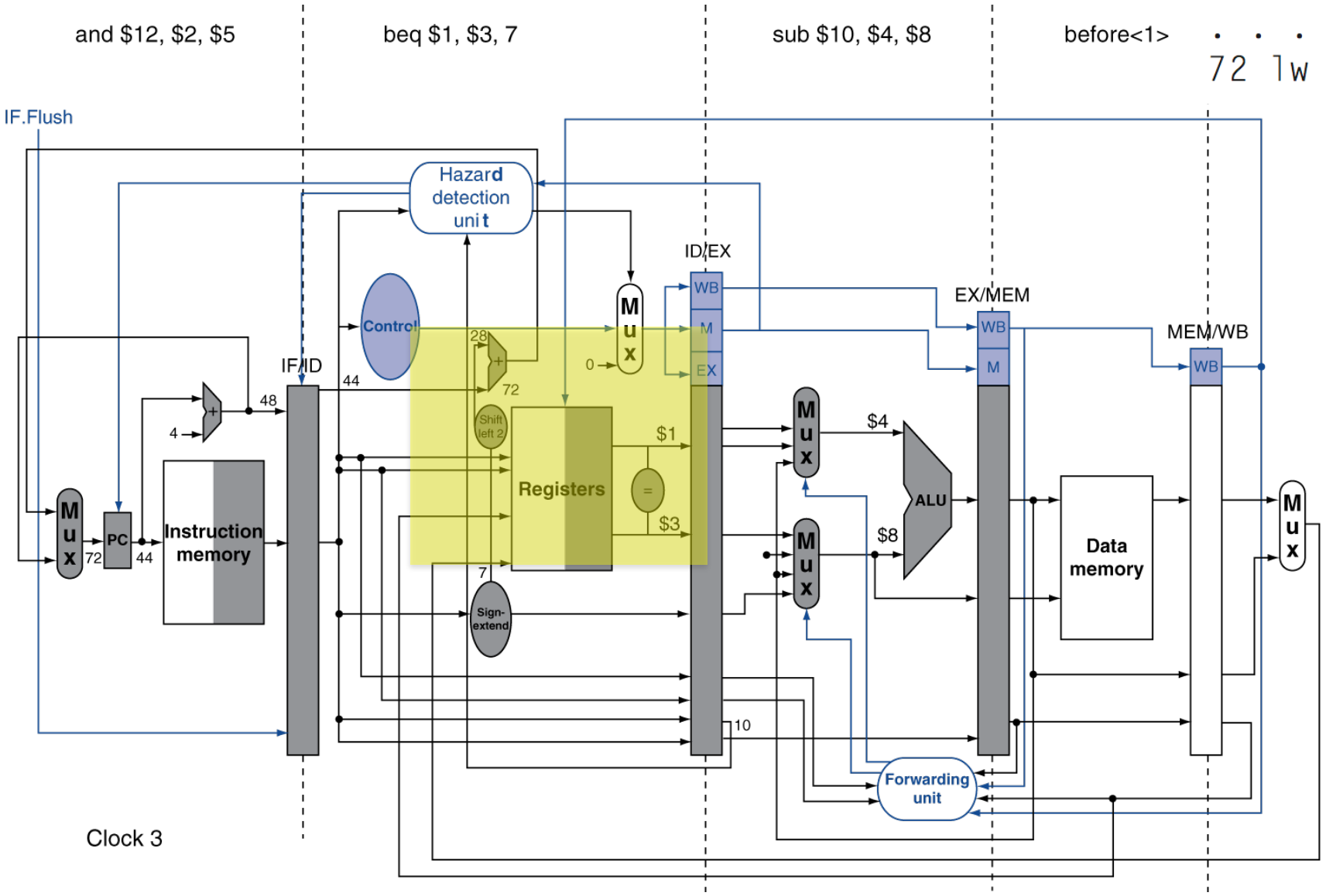
- Move hardware to determine outcome to ID stage → 1 cycle delay
  - Add Target address adder and Register comparator
- Example: branch taken

```
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40+4+7*4=72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```

Example: Branch Taken

```
36 sub $10, $4, $8
40 beq $1, $3, 7 #
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```

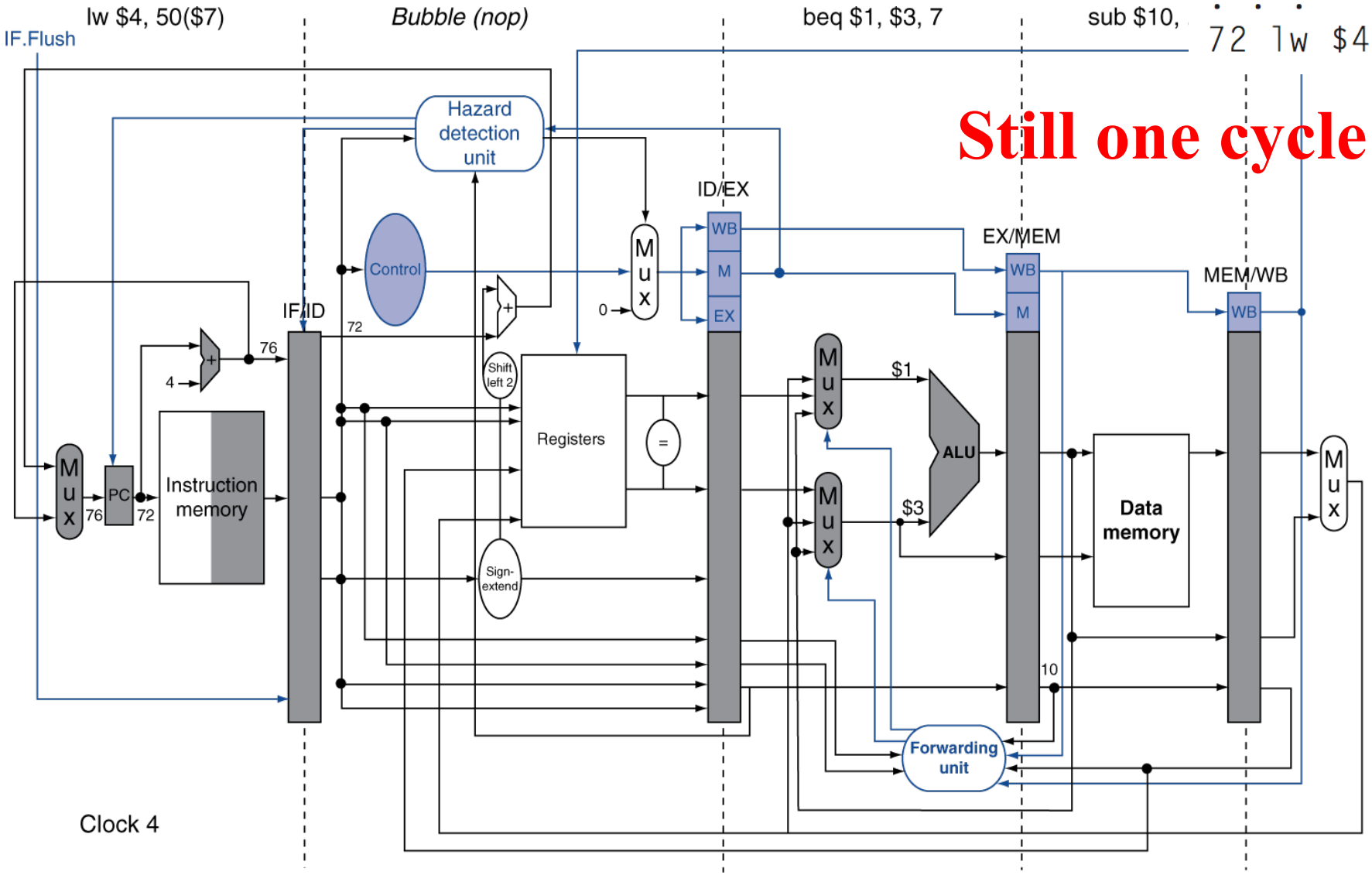
- Add is already fetched when beq outcome is determined



Example: Branch Taken

```
36 sub $10, $4, $8
40 beq $1, $3, 7 #
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```

- Add won't enter ID stage and branch target (lw) is fetched



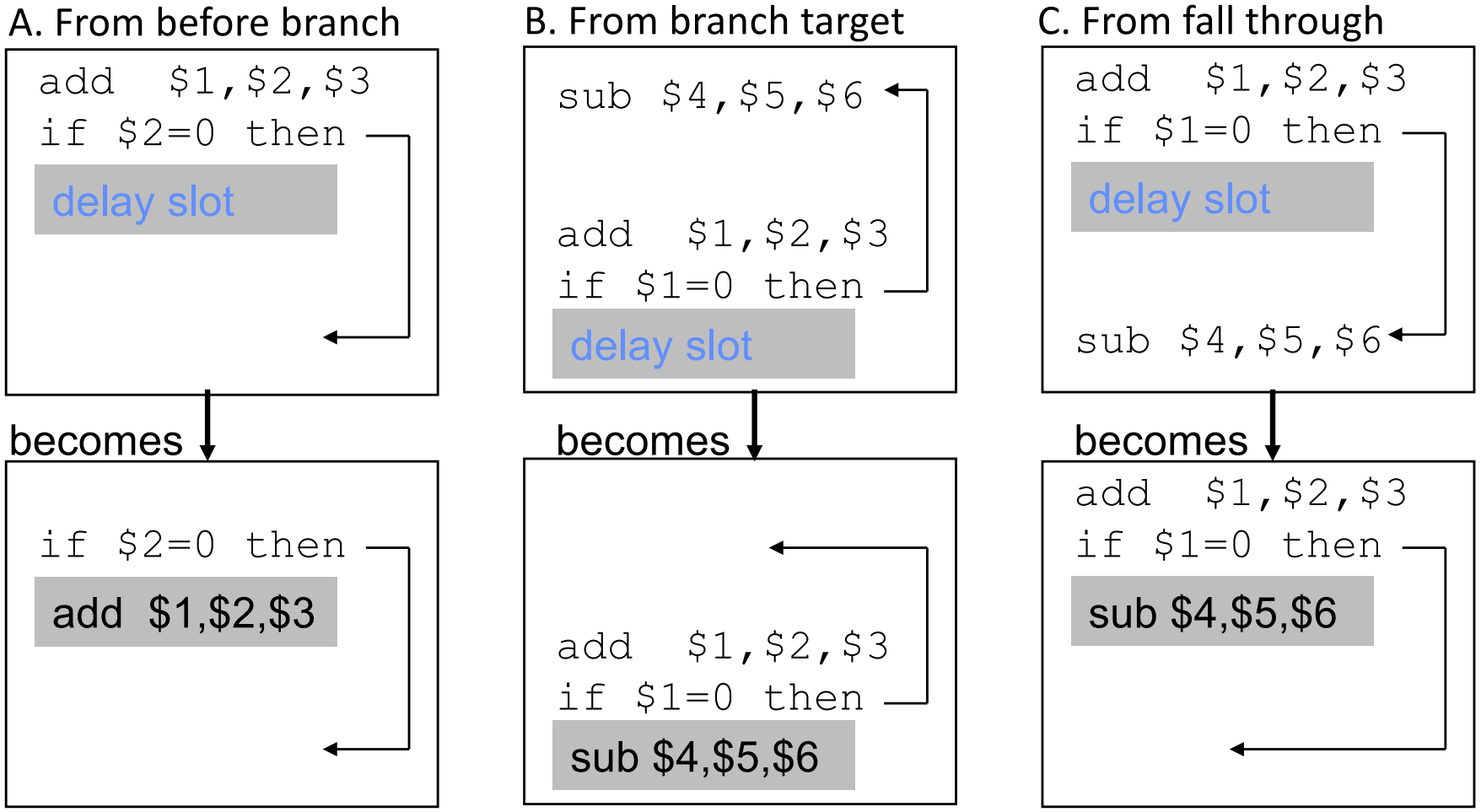
Still one cycle delay!

Four Branch Hazard Alternatives

- #1: Stall until branch direction is clear
- #2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - "Squash" instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% MIPS branches not taken on average
  - PC+4 already calculated, so use it to get next instruction
- #3: Predict Branch Taken
  - 53% MIPS branches taken on average
  - But haven't calculated branch target address in MIPS
    - MIPS still incurs 1 cycle branch penalty
    - Other machines: branch target known before outcome

Four Branch Hazard Alternatives

- #4: Schedule Branch Delay Slots
  - Exec an instruction in that delay slot regardless whether branch will be taken or not



Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

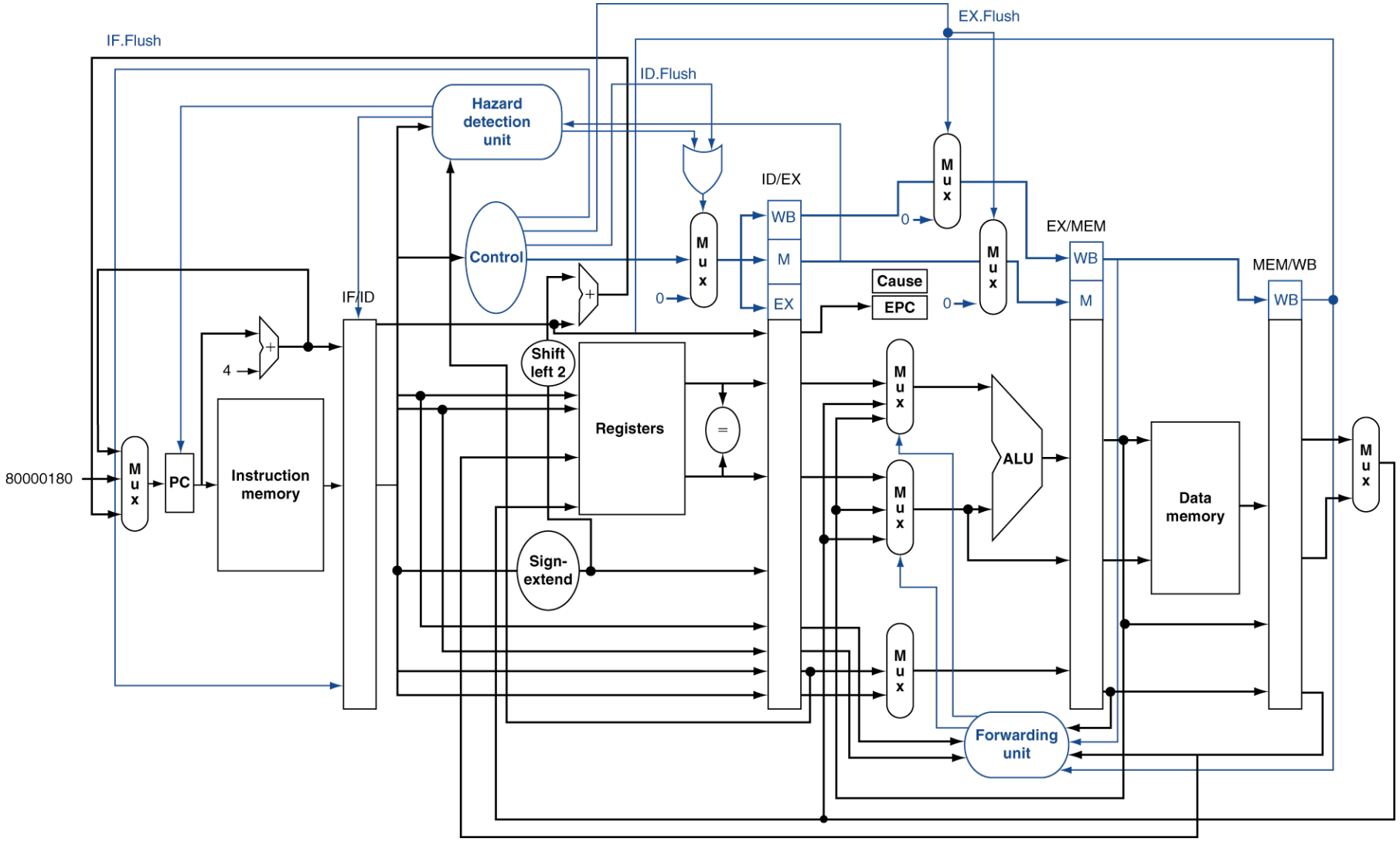
- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

  `add $1, $2, $1`
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Exception Example

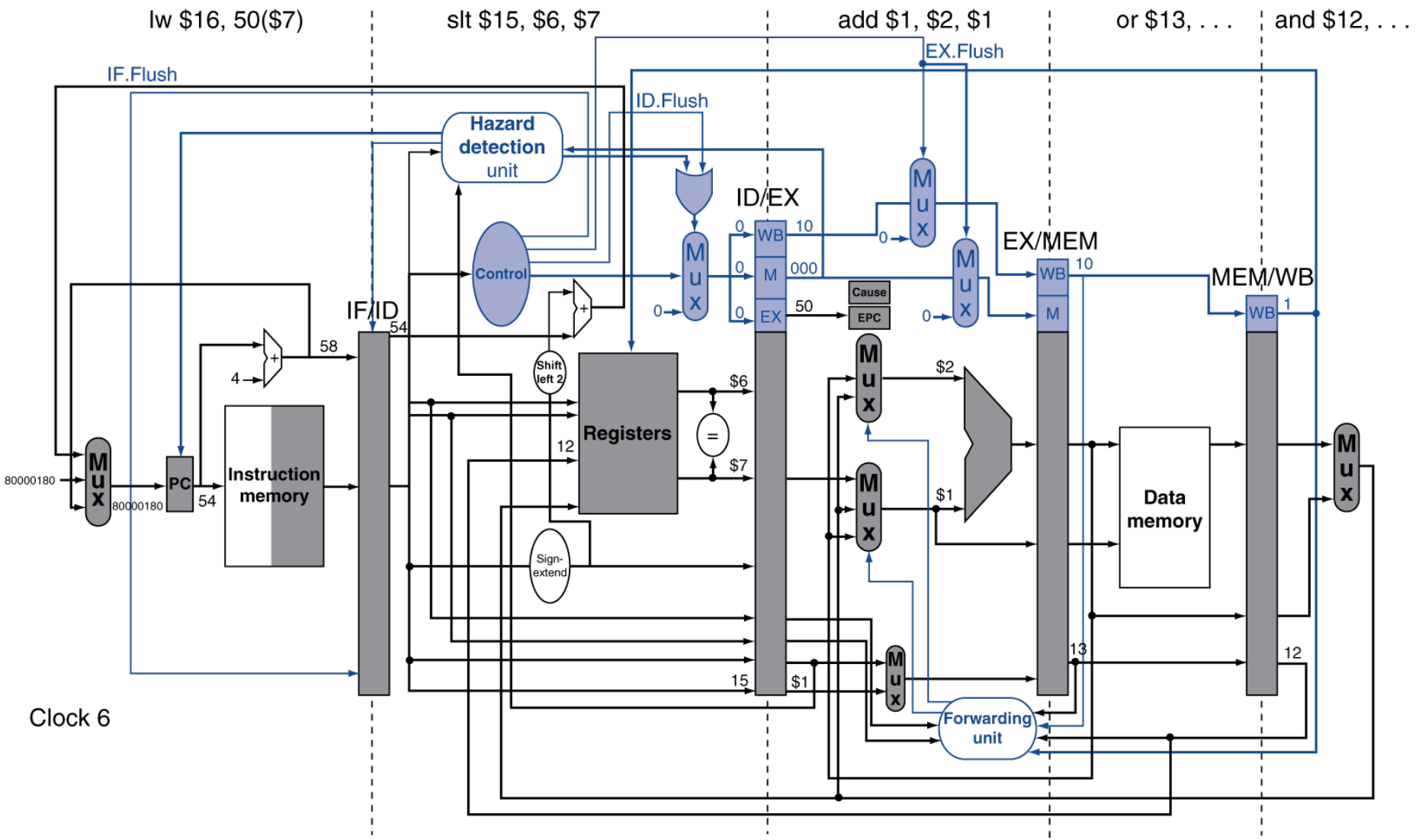
- Exception on `add` in

```
40   sub  $11, $2, $4
44   and  $12, $2, $5
48   or   $13, $2, $6
4C   add  $1,  $2, $1
50   slt  $15, $6, $7
54   lw   $16, 50($7)
…
```

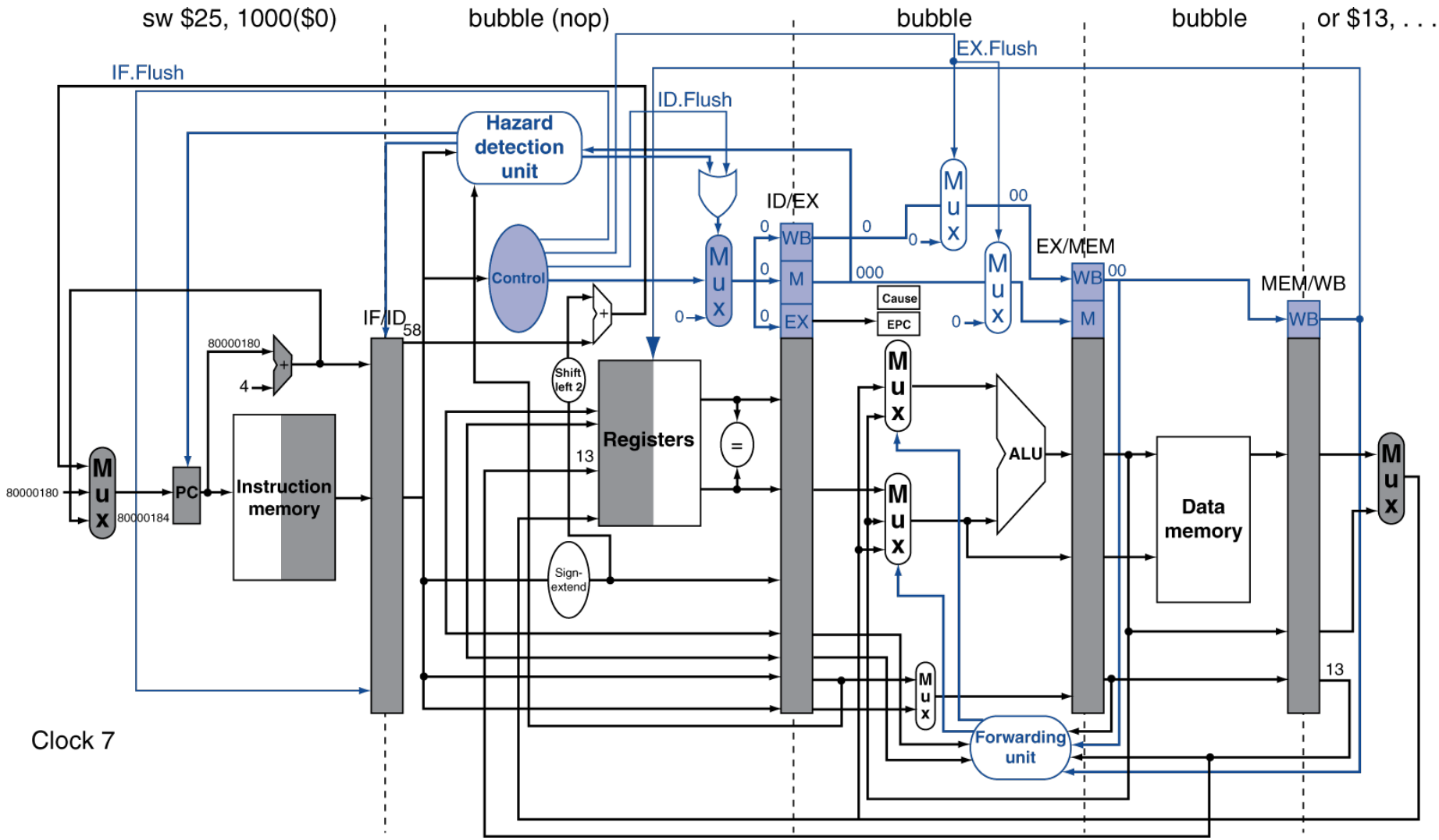
- Handler

```
80000180 sw   $25, 1000($0)
80000184 sw   $26, 1004($0)
…
```

Exception Example



Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

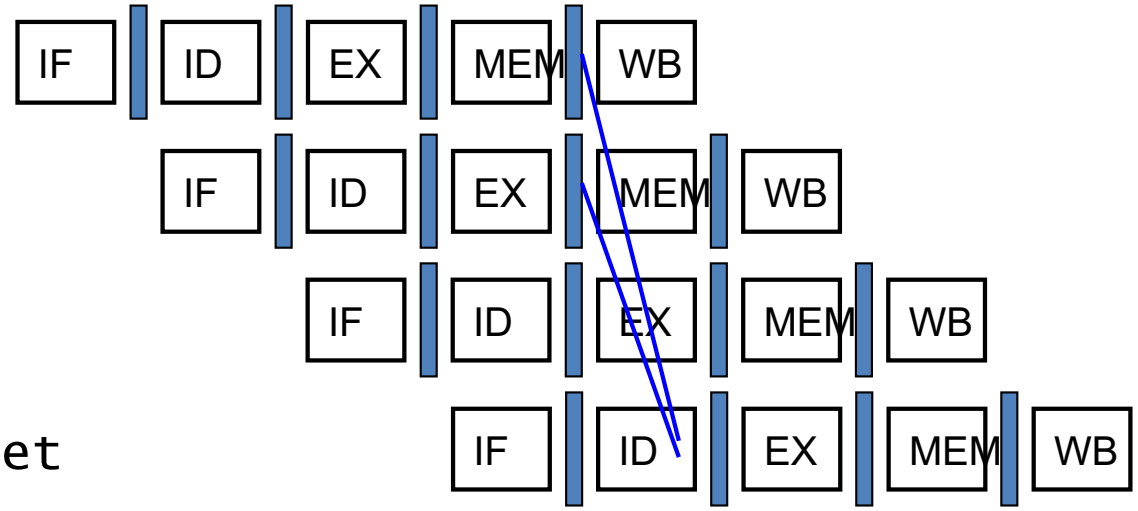
Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

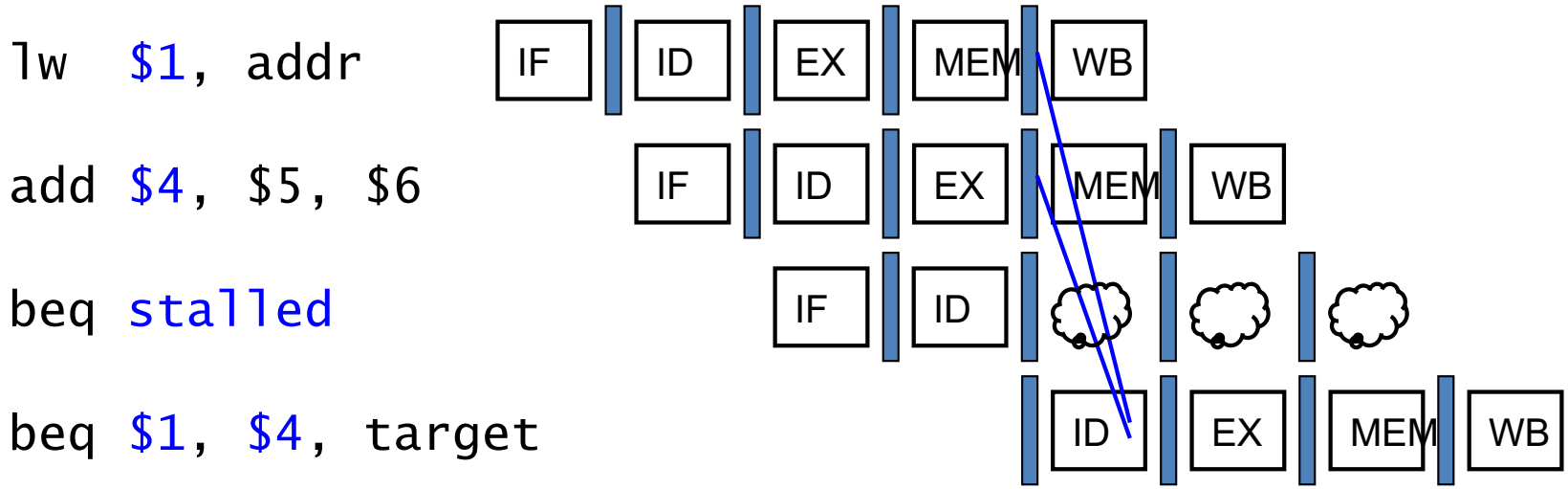
```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```



- Can resolve using forwarding

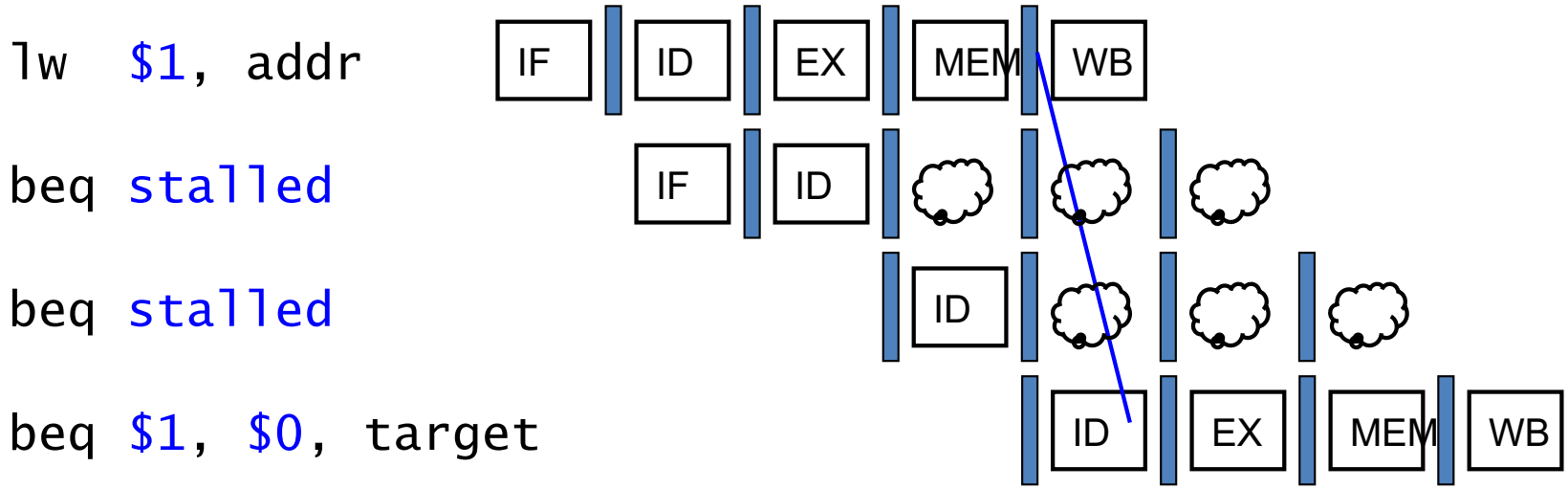
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles



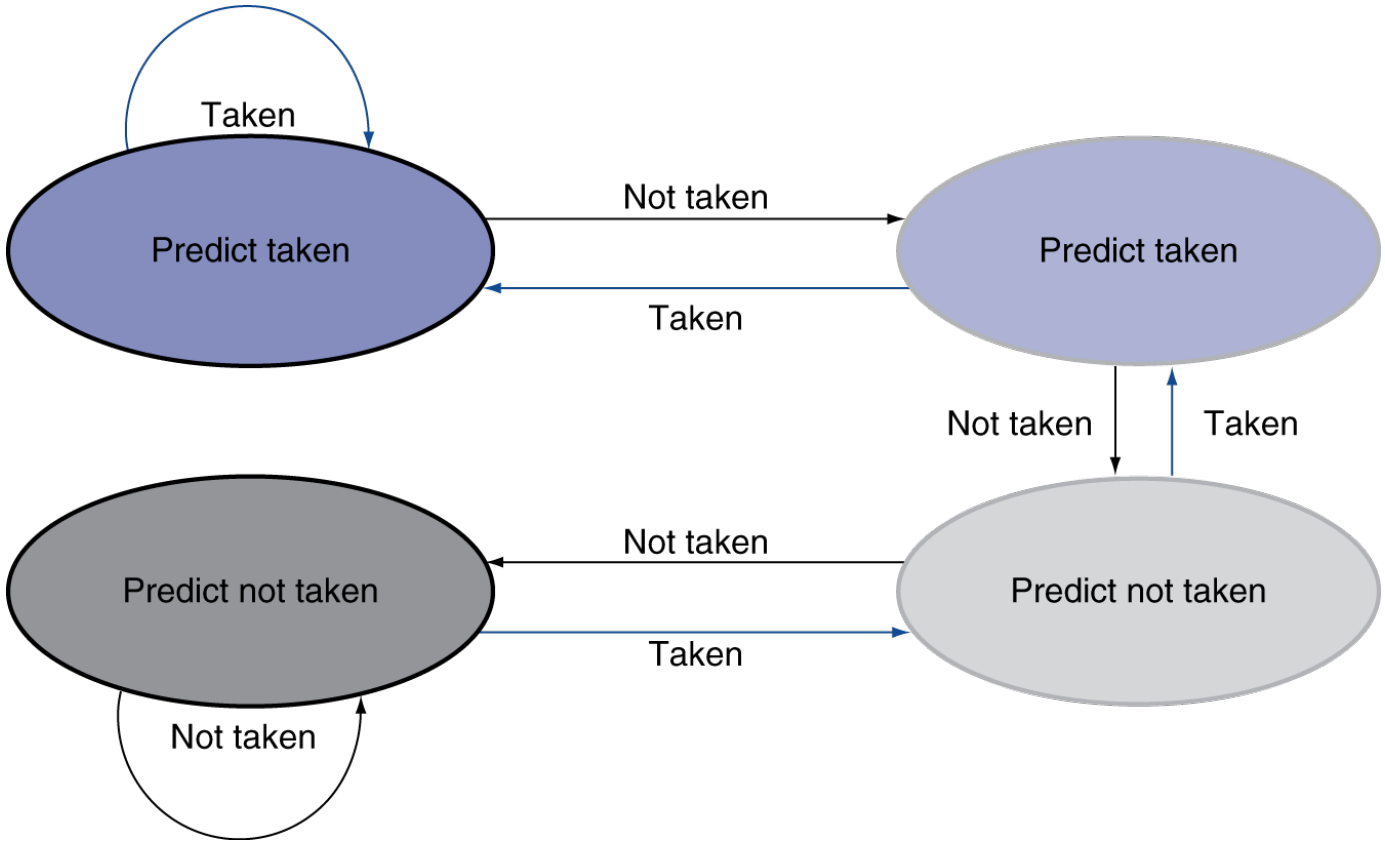
1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

Matrix Multiply

Unrolled C code

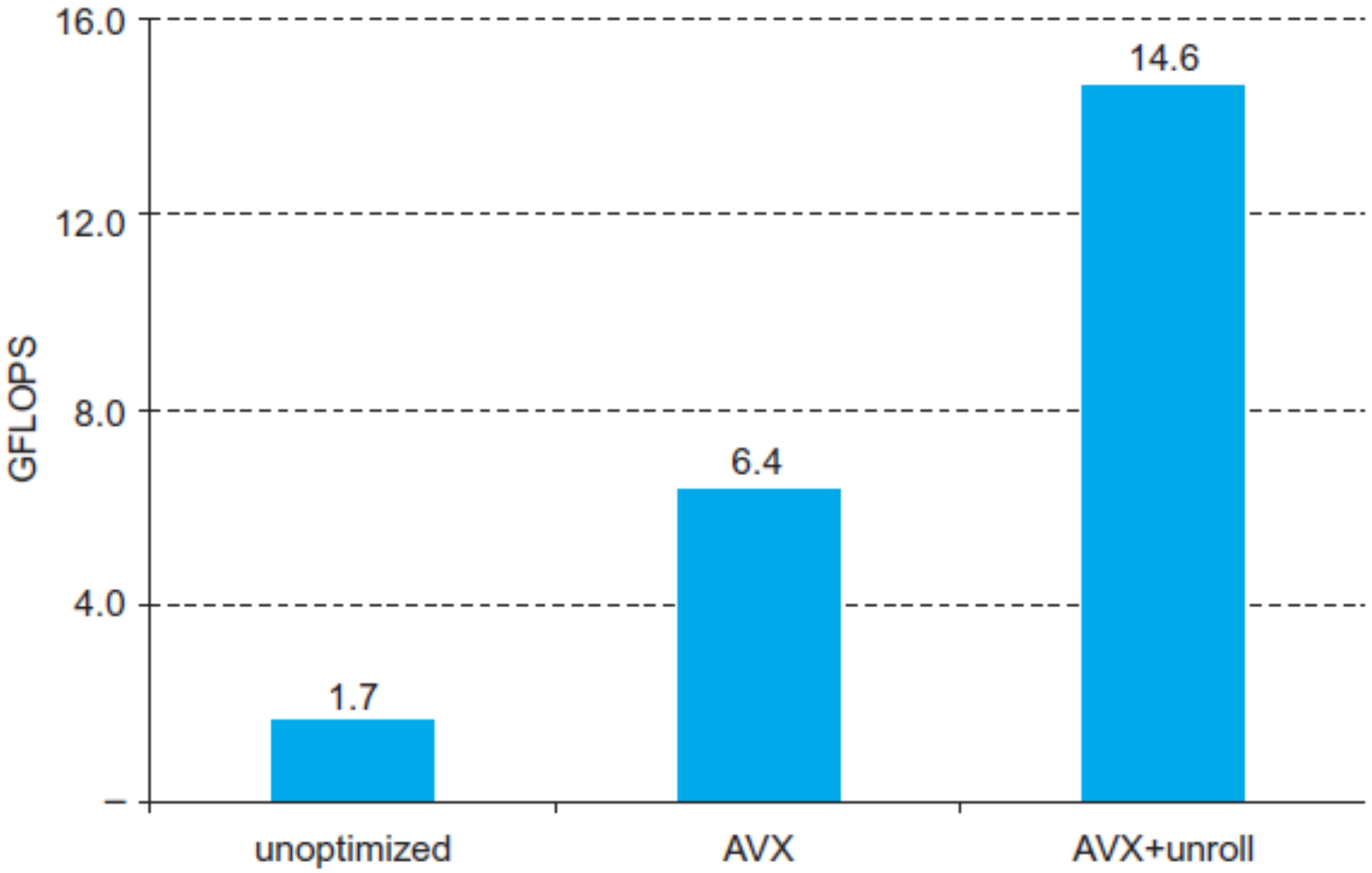
```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6  for ( int i = 0; i < n; i+=UNROLL*4 )
7   for ( int j = 0; j < n; j++ ) {
8    __m256d c[4];
9    for ( int x = 0; x < UNROLL; x++ )
10    c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12   for( int k = 0; k < n; k++ )
13   {
14    __m256d b = _mm256_broadcast_sd(B+k+j*n);
15    for (int x = 0; x < UNROLL; x++)
16    c[x] = _mm256_add_pd(c[x],
17                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18   }
19
20    for ( int x = 0; x < UNROLL; x++ )
21     _mm256_store_pd(C+i+x*4+j*n, c[x]);
22  }
23 }
```

Matrix Multiply

- Assembly code:

```
1 vmovapd (%r11),%ymm4                  # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                         # register %rax = %rbx
3 xor %ecx,%ecx                         # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3              # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2              # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1              # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0       # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5             # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4             # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3             # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0        # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                         # register %rax = %rax + %r8
16 cmp %r10,%rcx                        # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2             # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1             # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>                  # jump if not %r8 != %rax
20 add $0x1,%esi                        # register % esi = % esi + 1
21 vmovapd %ymm4,(%r11)                 # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)             # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)             # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)             # Store %ymm1 into 4 C elements
```

Performance Impact



Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

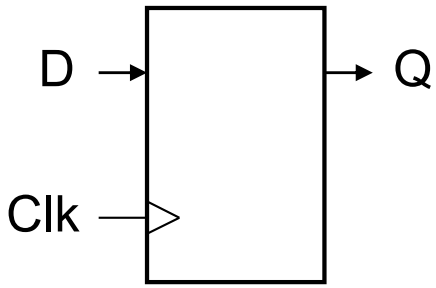
Pitfalls

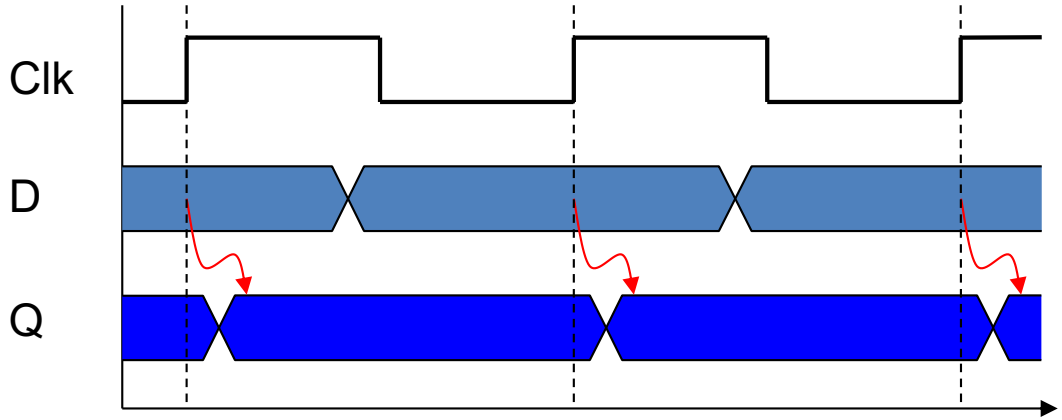
- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

End of Chapter 4

Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1





Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later