Ch. 5: Processor + Memory

December 12, 2008

Processor: Datapath and Control

- Goal:
  - implement hardware to execute instructions
  - study issues of performance
- Basic instruction set:
  - `lw, sw` (memory reference)
  - `add, sub, and, or, slt` (arithmetic-logical)
  - `beq, j` (branches)

Generic Implementation

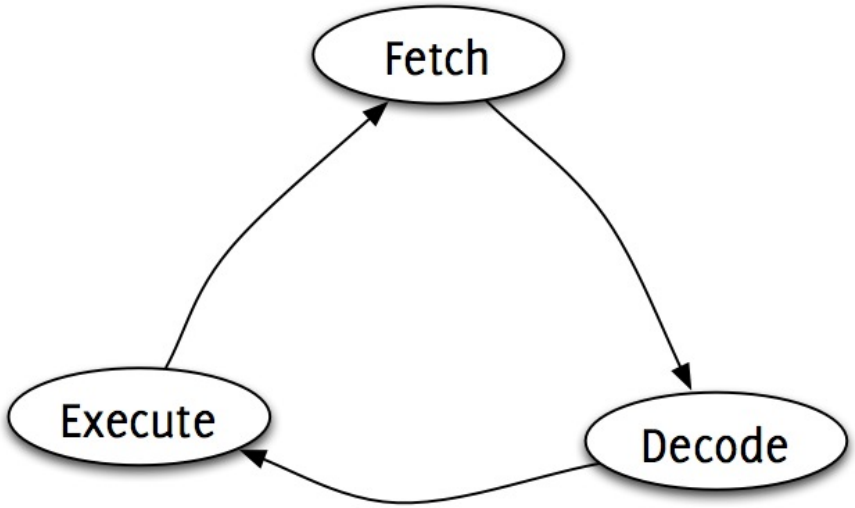


Figure: Cycle of Execution Instruction

Outline

Steps of Executing an Instruction

- **1:** All classes of instructions require initially:
  - Fetch instruction from memory
  - Read value of CPU registers (1 or 2)

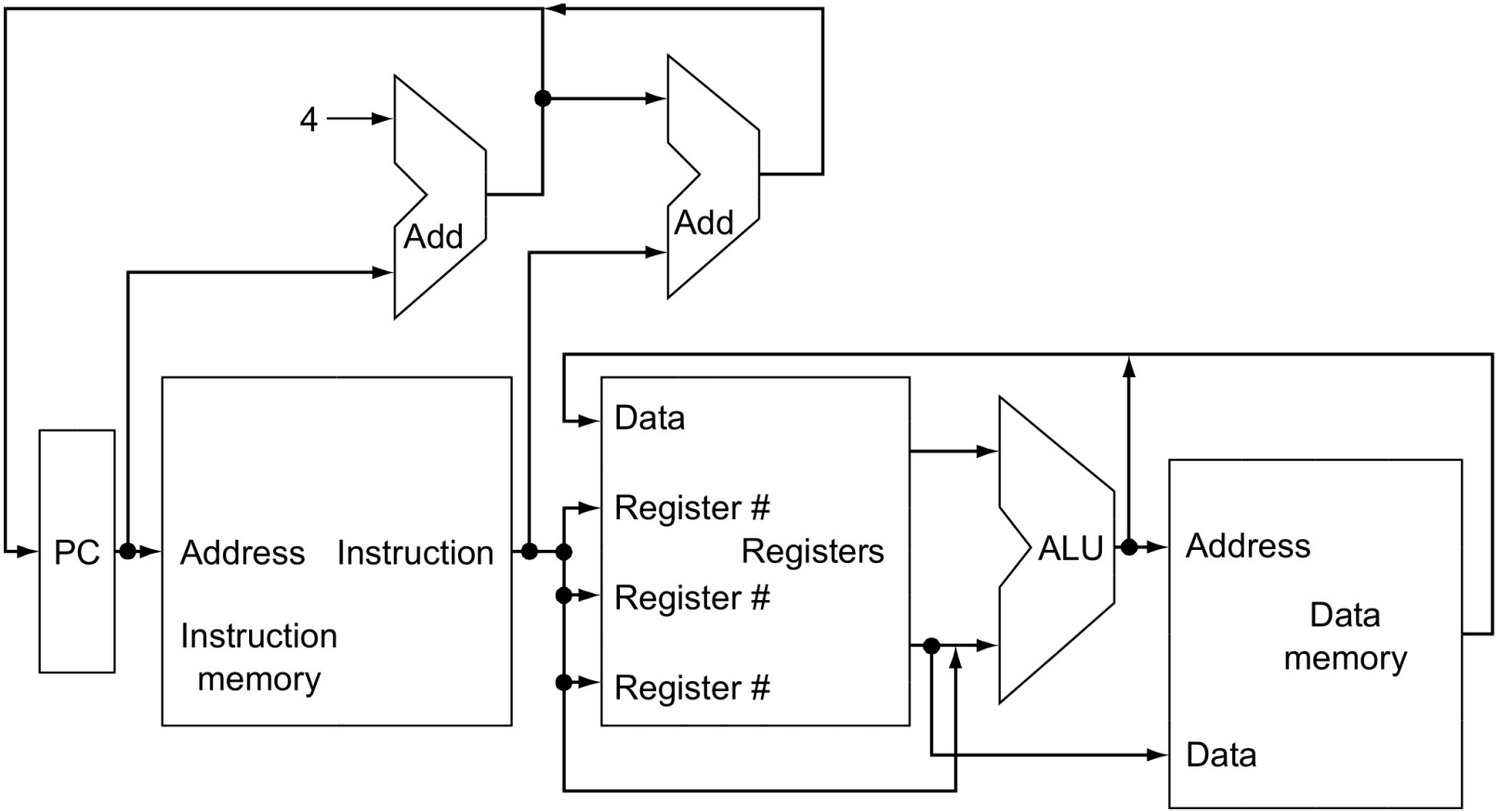
Steps of Executing an Instruction

- **2:** Common amongst all (except `j`): *after reading registers, use ALU*
  - calculate address (memory reference)
  - to add (arithmetic-logical)
  - compare (branches)

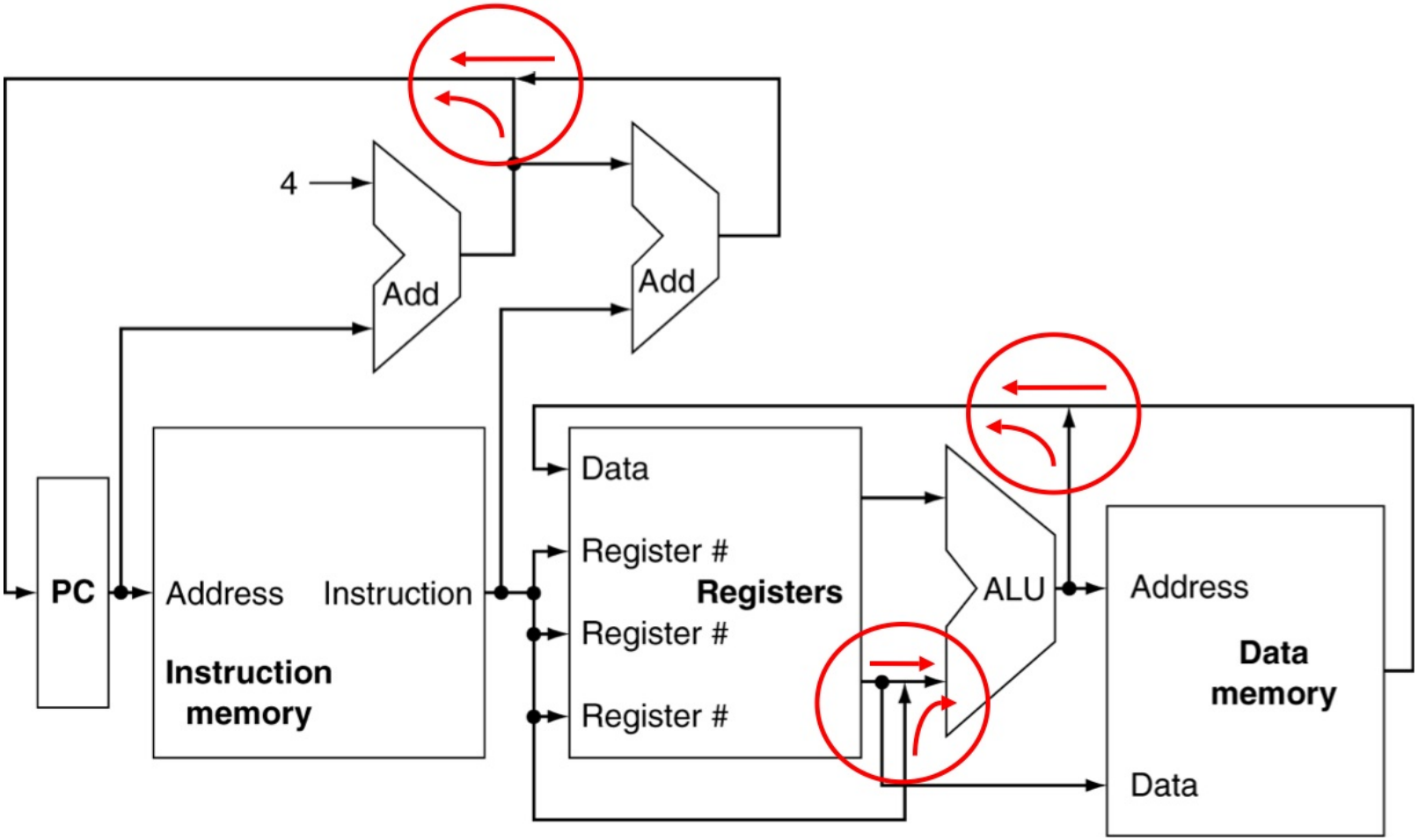
Steps of Executing an Instruction

- **3:** Different for each class
  - access memory to r/w (memory reference)
  - write from ALU to register (arithmetic-logical)
  - change PC to jump address (branch)

Abstract View of Implementation



Choice for Data



Can't just join the wires

Choice for Function

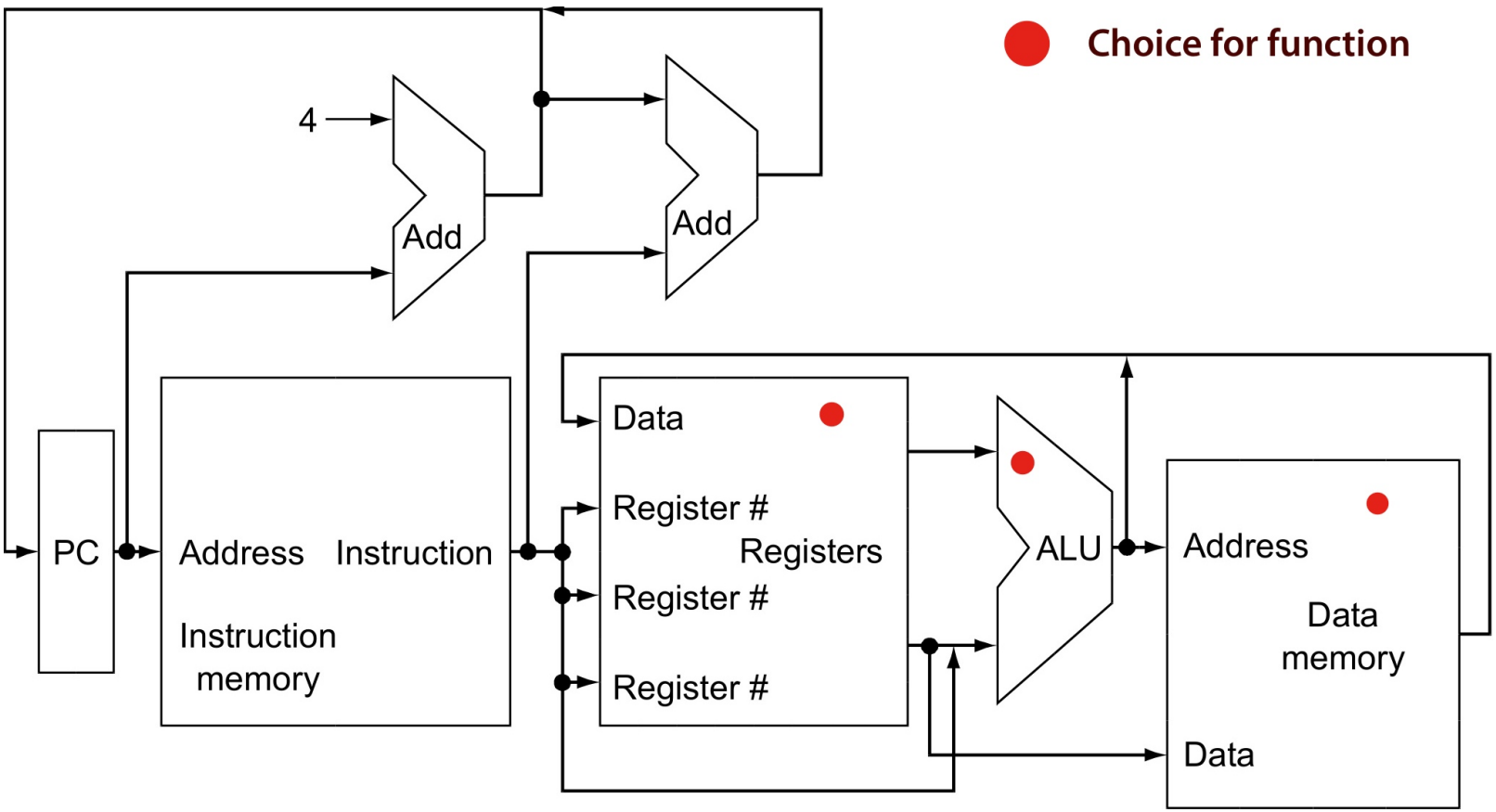


Figure: Some Units with Choice for Function

Adding details

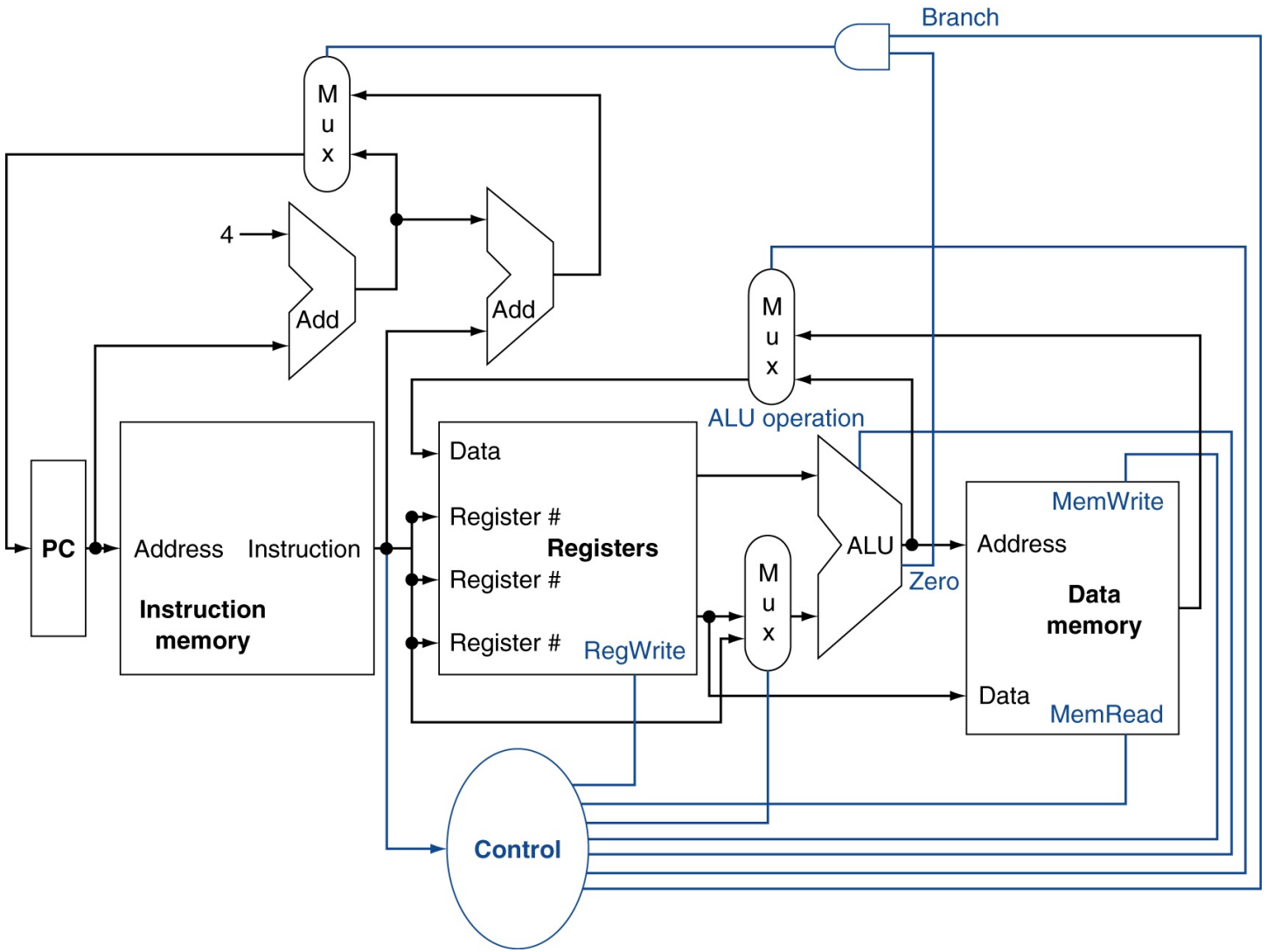


Figure: Adding multiplexors and control

Logic Elements

- Combinational elements
  - without memory; o/p depends on i/p
  - basic elements: AND, OR, AND, . . .
  - ALU, multiplexors

Logic Elements

- Combinational elements
  - without memory; o/p depends on i/p
  - basic elements: AND, OR, AND, . . .
  - ALU, multiplexors
- Sequential or state elements
  - o/p depends on i/p + current state
  - memory, registers

Outline

Multiplexor

- Select one i/p out of many, based on a signal

Multiplexor

- Select one i/p out of many, based on a signal
- 1-bit Mux with 4 input lines

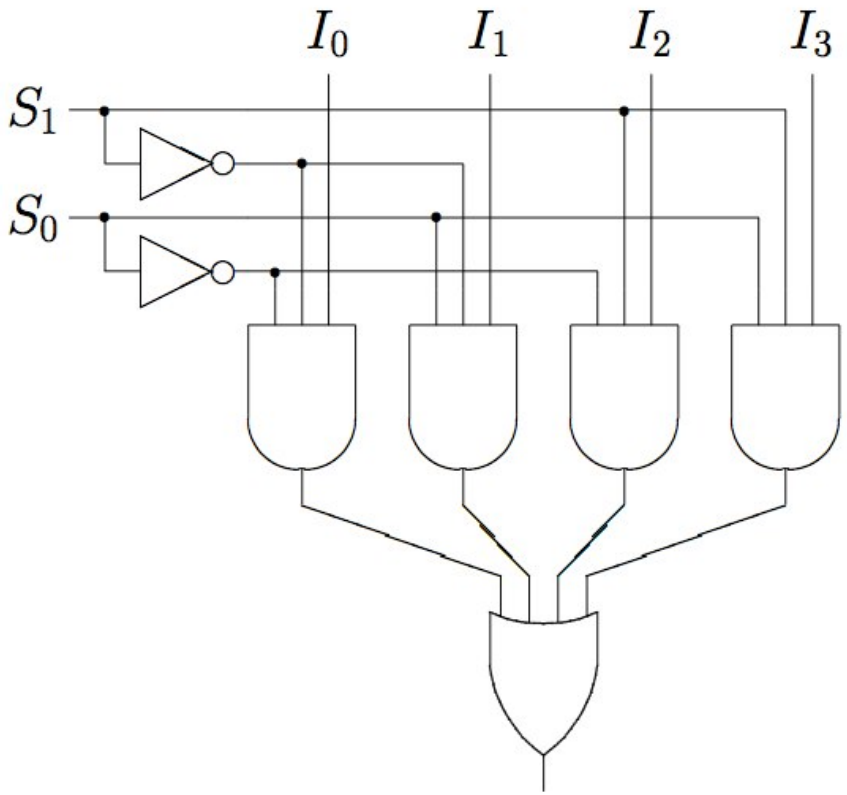


Figure: Select I_0, I_1, I_2, I_3 based on S_0, S_1

32-bit Multiplexor

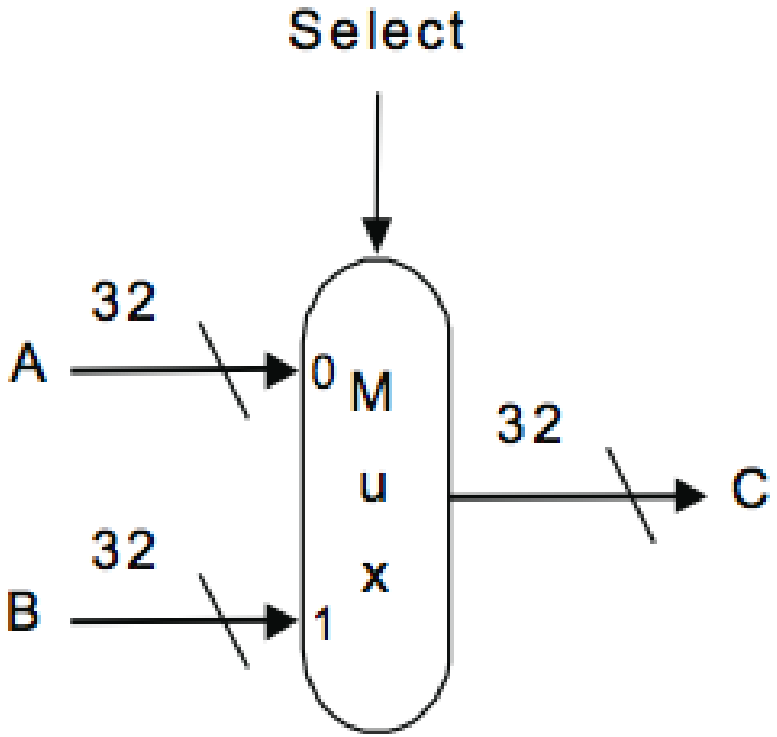


Figure: 32-bit Mux with 2 input lines

32-bit Multiplexor

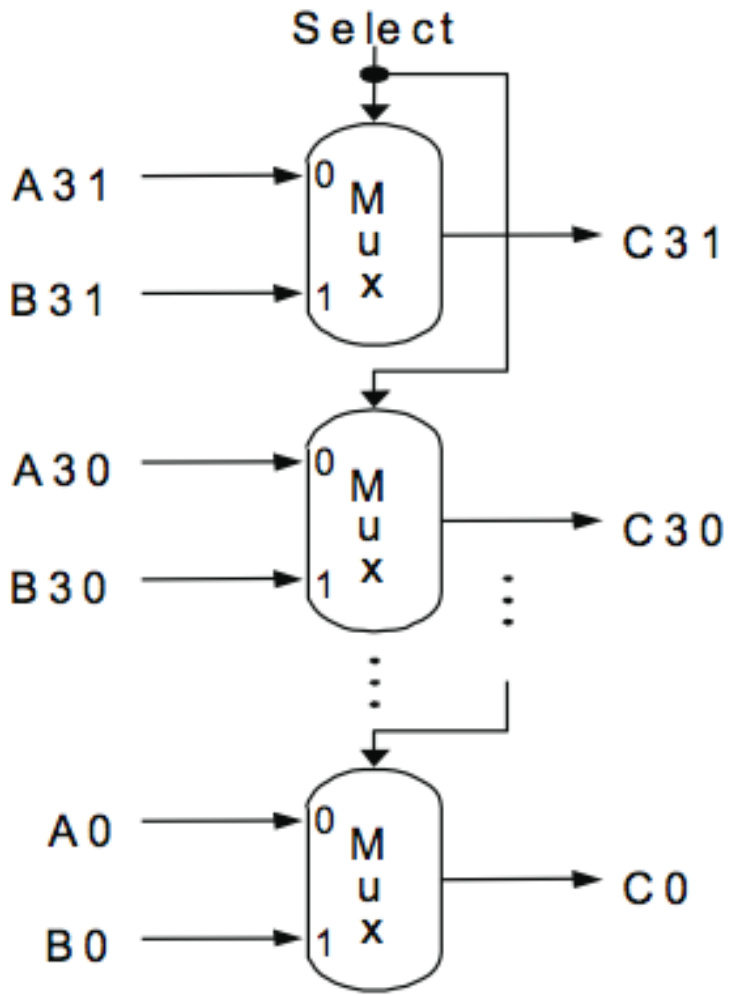


Figure: 32-bit Multiplexor construction

One-bit Adder

- Simple adder:
  - 1 bit numbers
  - carry in, carry out

One-bit Adder

- Simple adder:
  - 1 bit numbers
  - carry in, carry out

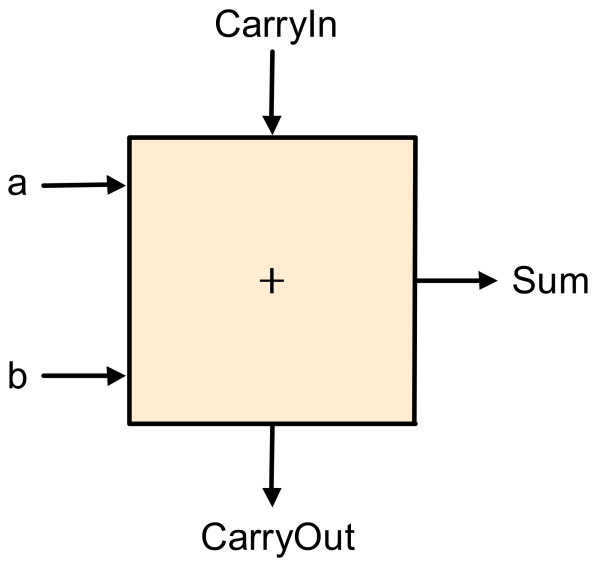


Figure: One-bit adder

One-bit Adder

- Simple adder:
  - 1 bit numbers
  - carry in, carry out

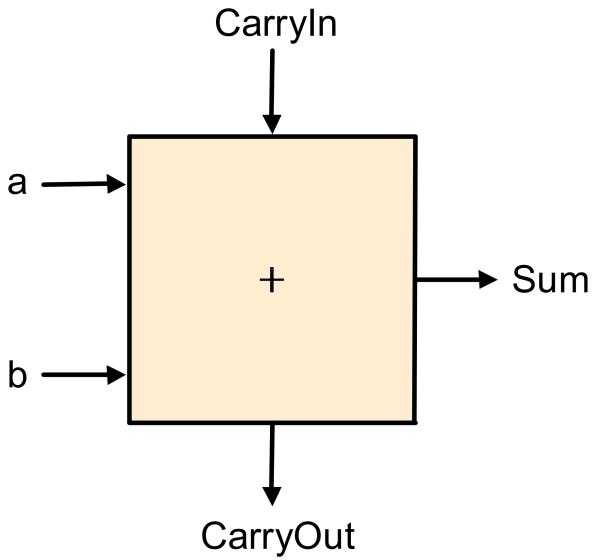


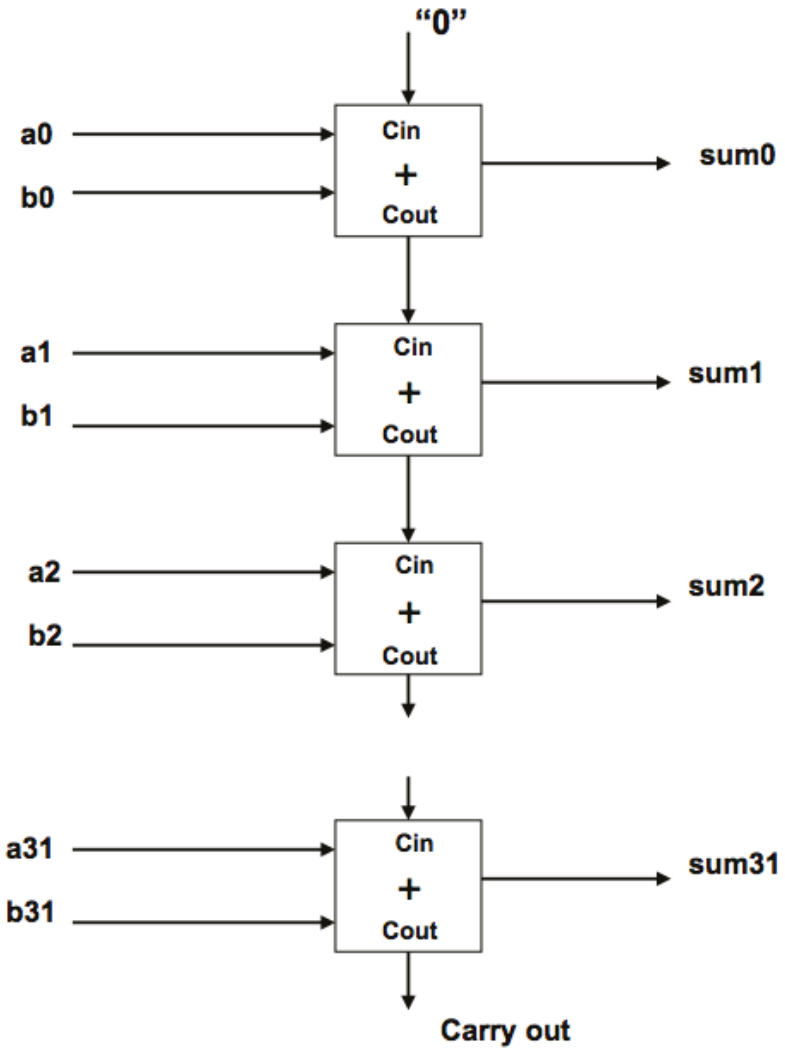
Figure: One-bit adder

$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

$$sum = a \ \textbf{xor} \ b \ \textbf{xor} \ c_{in}$$

32-bit adder

- Constructing 32-bit adder from 1-bit adders



Simple One-bit ALU

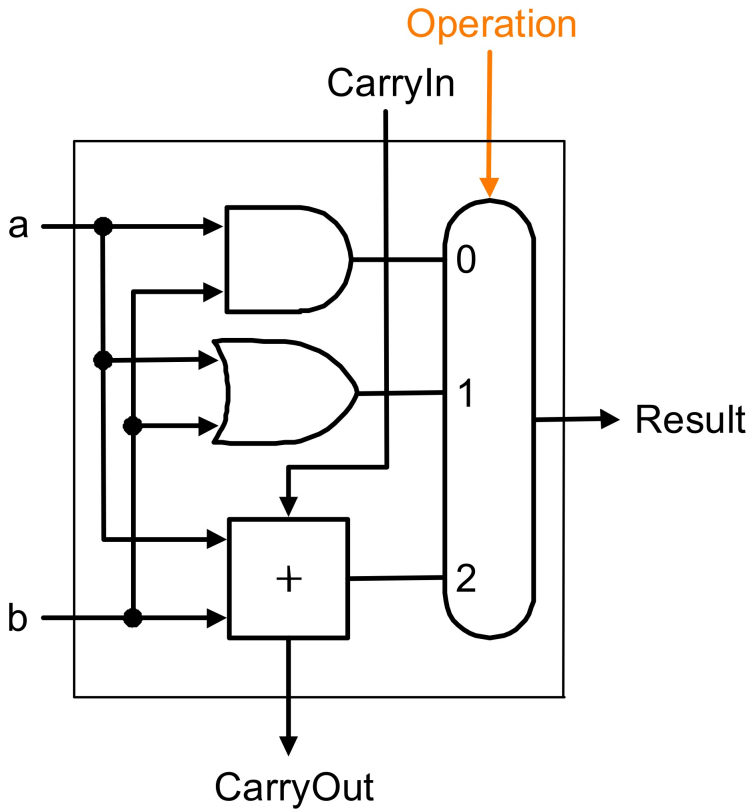


Figure: One-bit ALU: AND, OR, ADD

Simple One-bit ALU

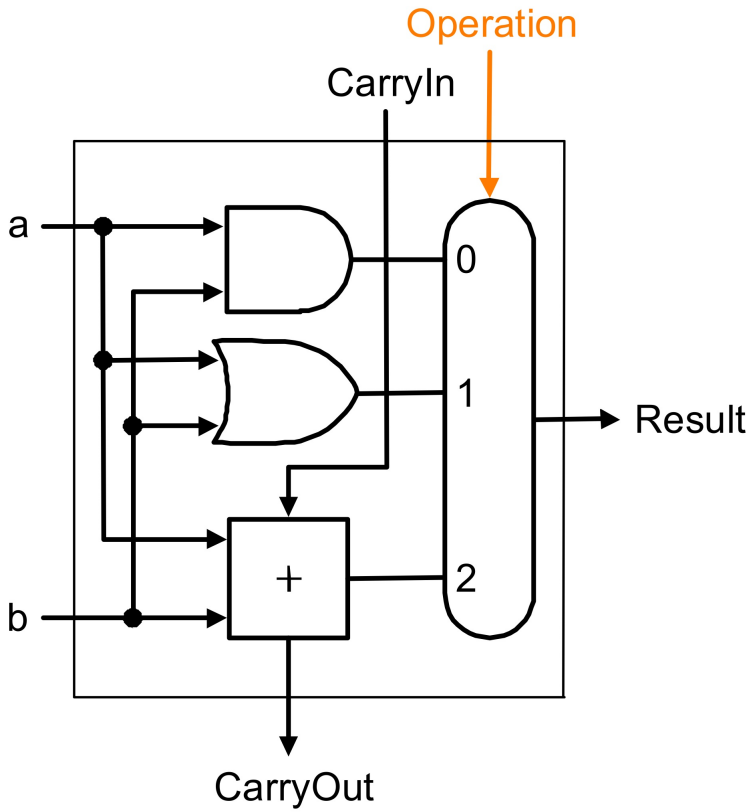


Figure: One-bit ALU: AND, OR, ADD

- 00 → AND
- 01 → OR
- 10 → ADD

Adding Subtraction

- Negate b and add

Adding Subtraction

- Negate b and add

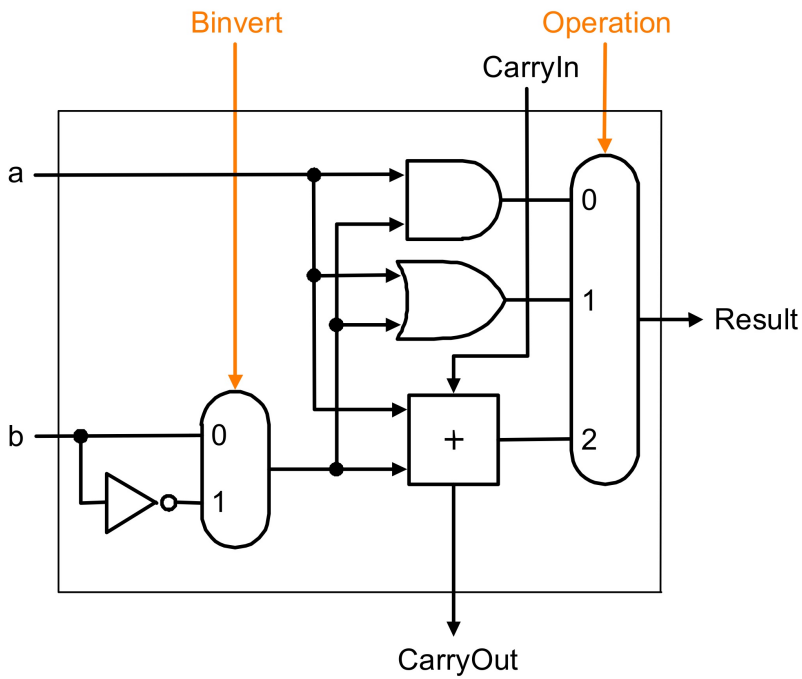


Figure: Adding subtraction

Adding Subtraction

- Negate b and add

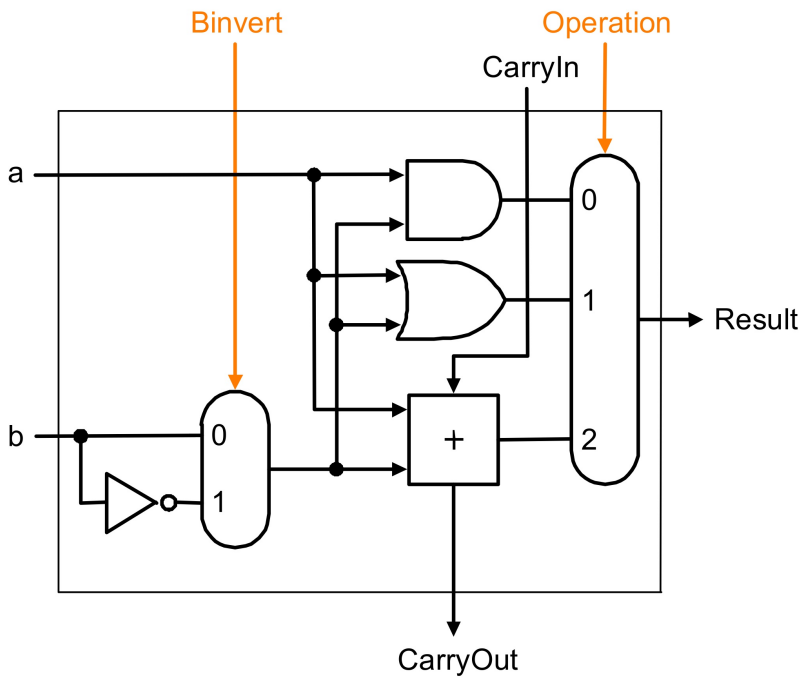
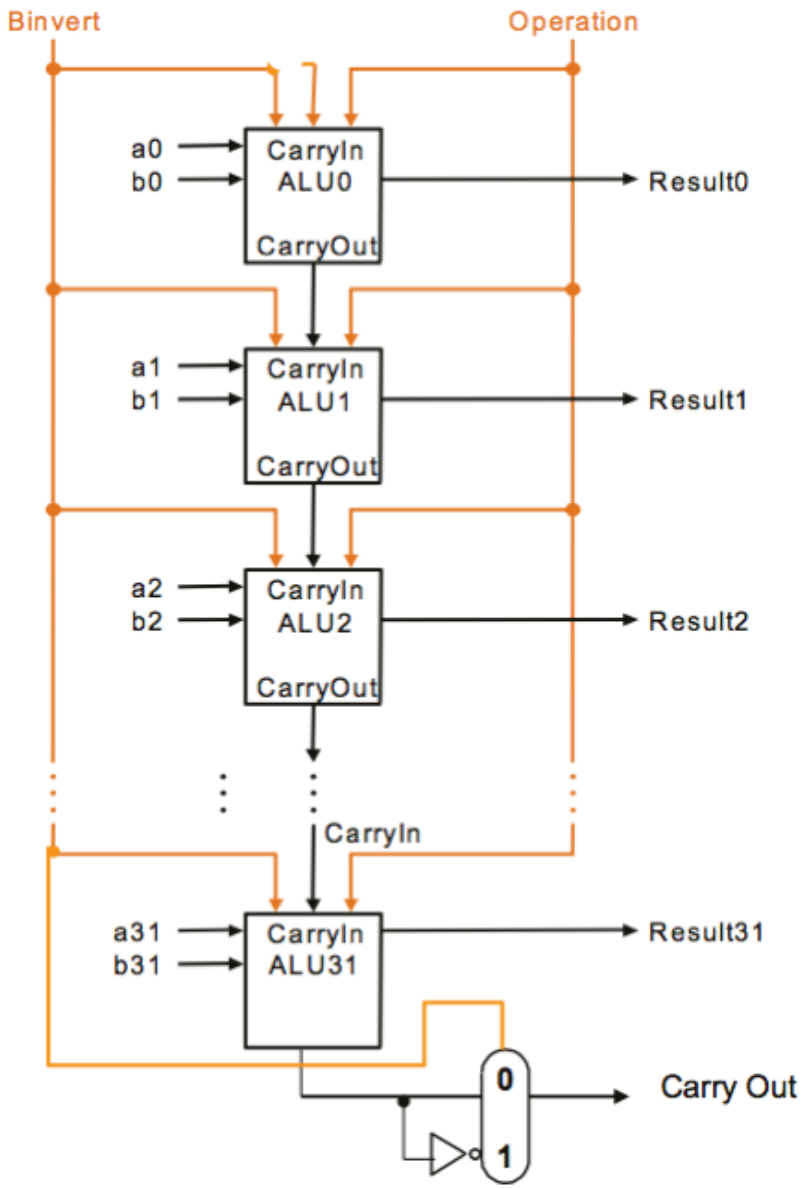


Figure: Adding subtraction

	Control lines	
Function	**Binvert** (1 line)	**Operation** (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10

Figure: Control signals

32-bit ALU


Binvert
Operation
a0
b0
CarryIn
ALU0
CarryOut
Result0
a1
b1
CarryIn
ALU1
CarryOut
Result1
a2
b2
CarryIn
ALU2
CarryOut
Result2
CarryIn
a31
b31
CarryIn
ALU31
Result31
0
1
Carry Out

Adding set-less-than

- Can be implemented with subtraction
- For all except LSB: output is 0
- For LSB: output is $\texttt{sign}(\texttt{A} - \texttt{B})$

Adding set-less-than

- Can be implemented with subtraction
- For all except LSB: output is 0
- For LSB: output is `sign`(`A` − `B`)
- Strategy:
  - Input of all ALU's except LSB: 0 (passes through)
  - Input of LSB: `Set` output of MSB

ALU Designs with SLT

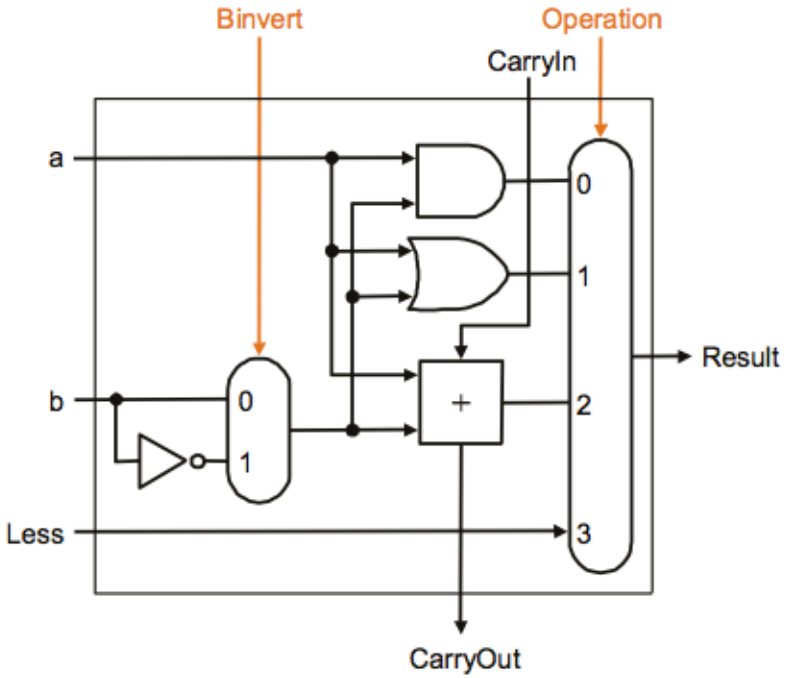


Figure: ALU for non-MSB bits with 5 functions

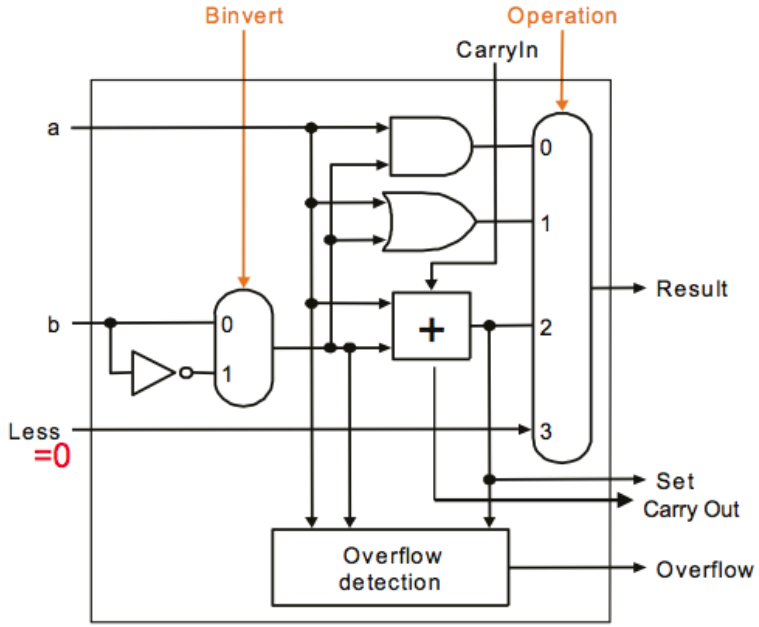
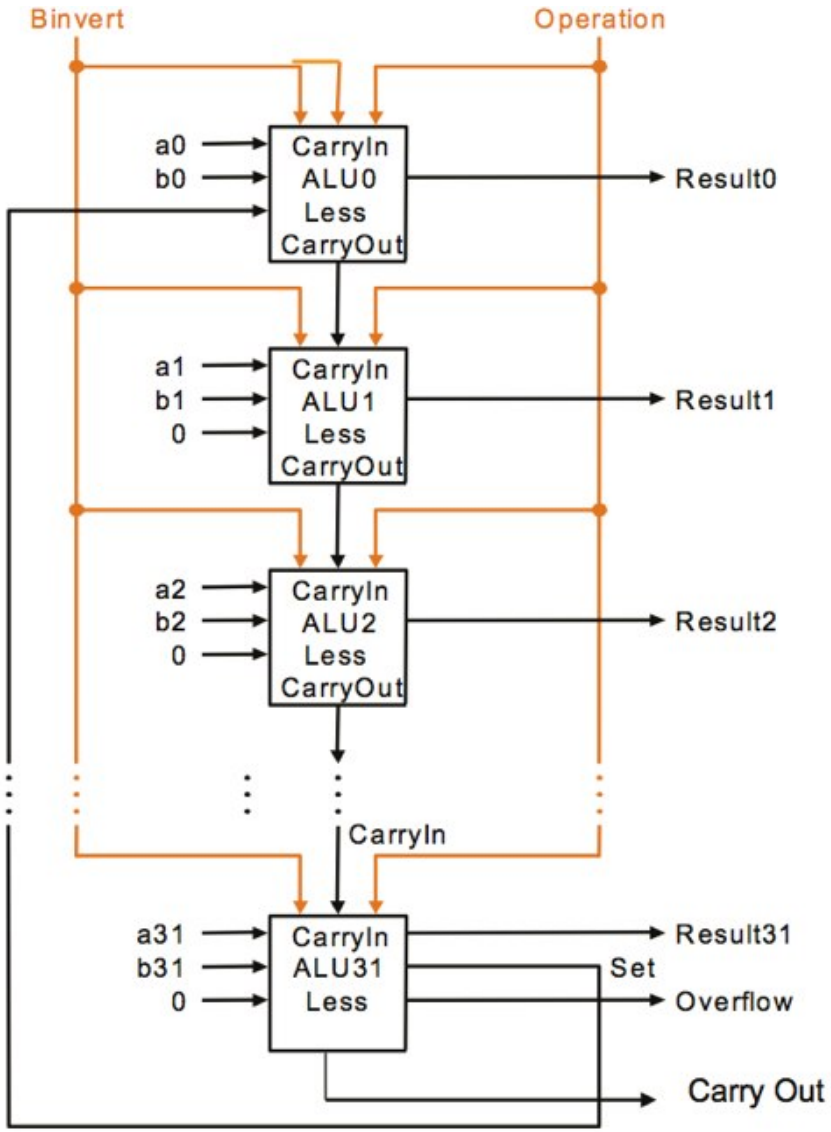


Figure: ALU for MSB bit (with shiny overflow detection!)

- `Operation`: 3, `BInvert`: 1

32-bit ALU with SLT



Outline

Clock and Clocking Methodology

- When should a sequential element's state be written/read
  - in a *synchronous* system

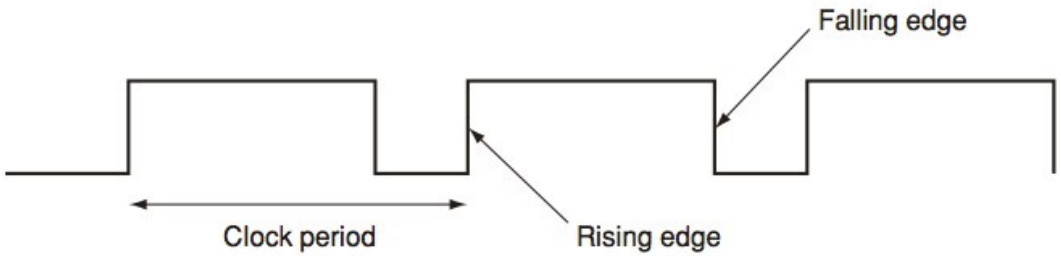


Figure: Clock Signal

Clock and Clocking Methodology

- When should a sequential element's state be written/read
  - in a *synchronous* system

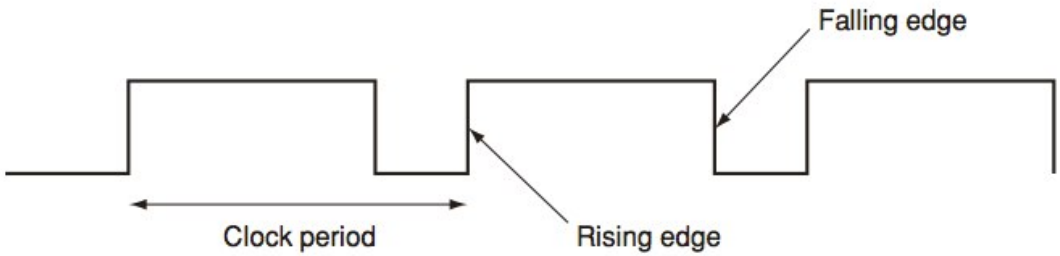


Figure: Clock Signal

- Edge-triggered clocking
  - fall or rise is *active*
  - causes state change
  - usually *rise*

Clock

- I/p, O/p to combinational elements → sequential elements

Clock

- I/p, O/p to combinational elements → sequential elements
  - ∵ they don't store anything

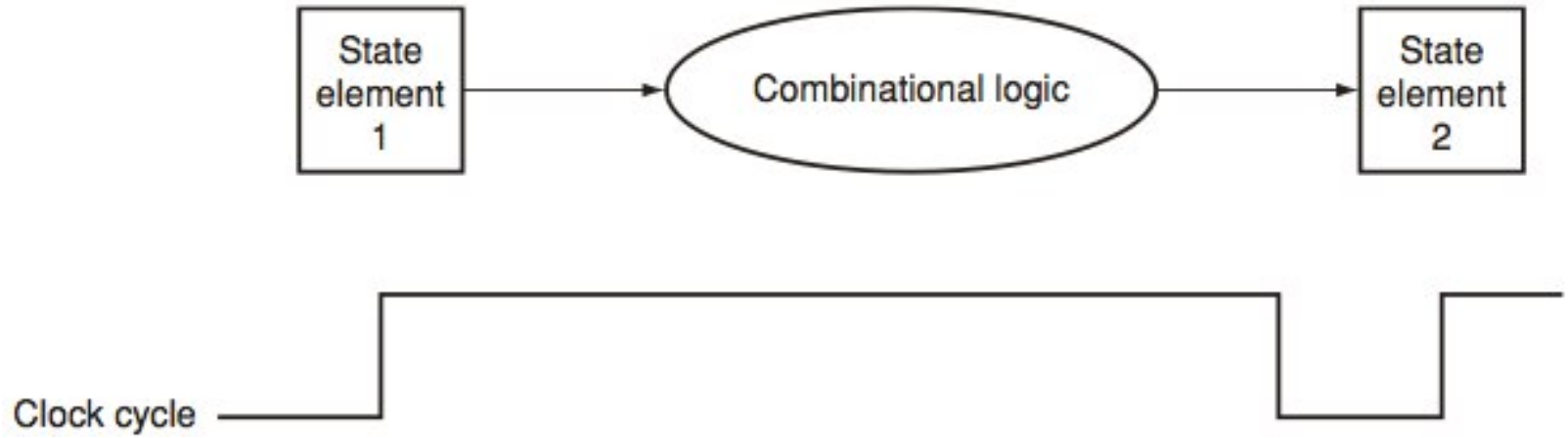


Figure: Combinational Logic behavior with Clock

S-R Latch

- Simplest memory element
- Output = stored value
- Change stored value: set-reset
- Unclocked

S-R Latch

- Simplest memory element
- Output = stored value
- Change stored value: set-reset
- Unclocked

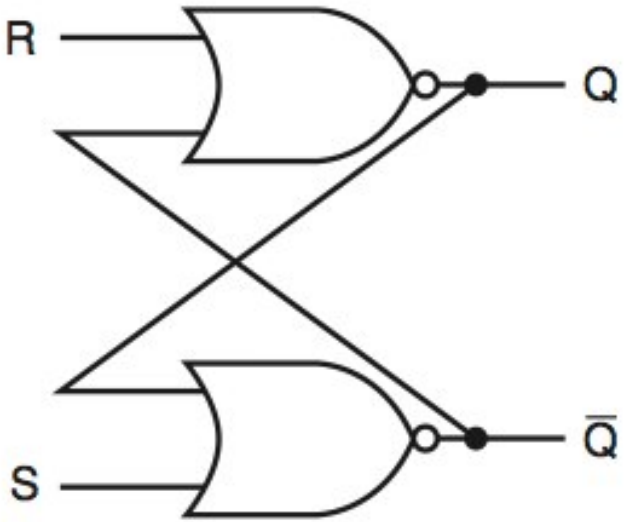


Figure: S-R Latch

SR Latch		
S	**R**	**Action**
0	0	Keep previous state
0	1	Q = 0
1	0	Q = 1
1	1	Illegal operation

Figure: Operation of the S-R Latch

D Latch

- Output = stored value
- Stored value changes based on clock (enable)

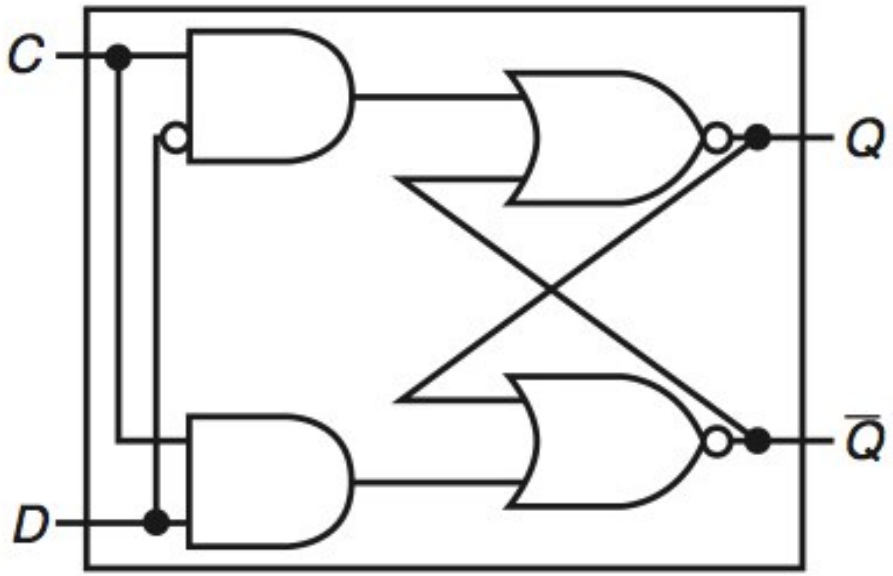


Figure: D Latch

D Flip-Flop		
C	**D**	**Q**
0	X	Previous value
1	0	0 (Reset)
1	1	1 (Set)

Figure: D Latch Operation

D Latch

- Output = stored value
- Stored value changes based on clock (enable)

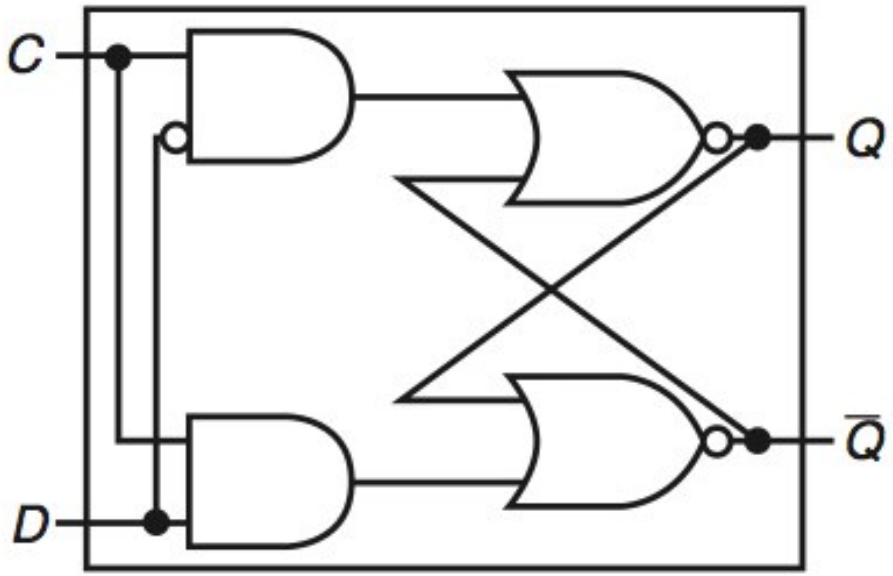


Figure: D Latch

D Flip-Flop		
C	D	Q
0	X	Previous value
1	0	0 (Reset)
1	1	1 (Set)

Figure: D Latch Operation

- Functioning:
  - $C = 1 \Rightarrow Q = D$
  - $C = 0 \Rightarrow Q = D_{previous}$

D Flip-Flop

- Slightly more complicated than the latch

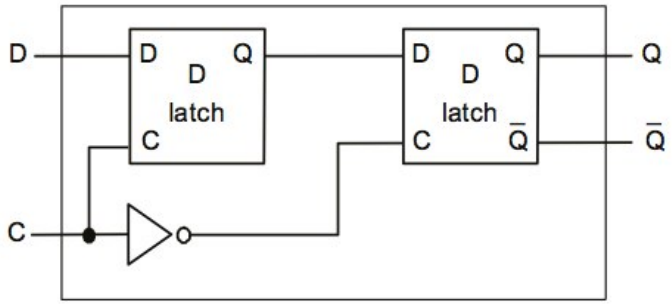


Figure: D Flip-Flop

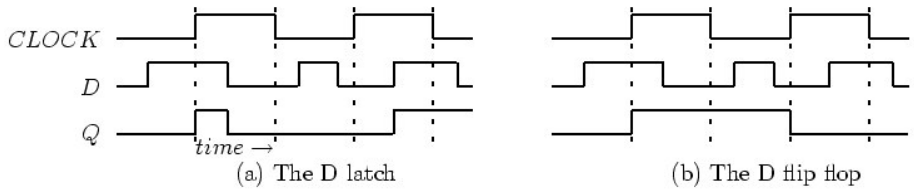


Figure: D Flip-Flop and Latch

D Flip-Flop

- Slightly more complicated than the latch

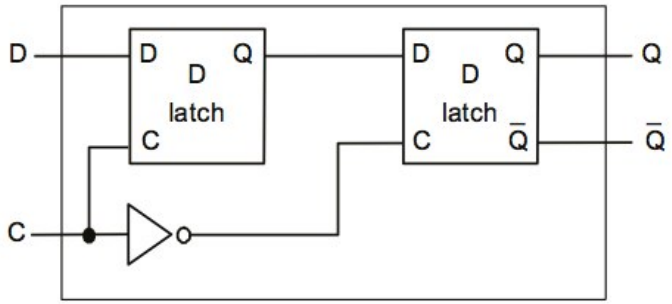


Figure: D Flip-Flop

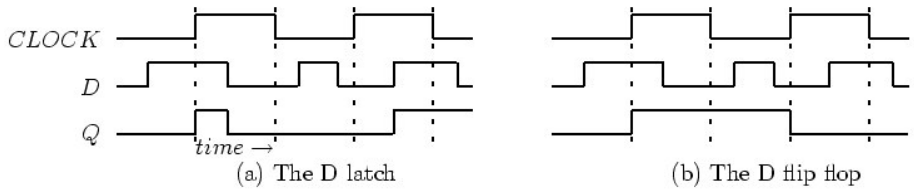


Figure: D Flip-Flop and Latch

- Latch: state changes with change in input and clock being asserted (=1)
- Flip-flop: state changes only on clock rise

D Flip-Flop

- Slightly more complicated than the latch

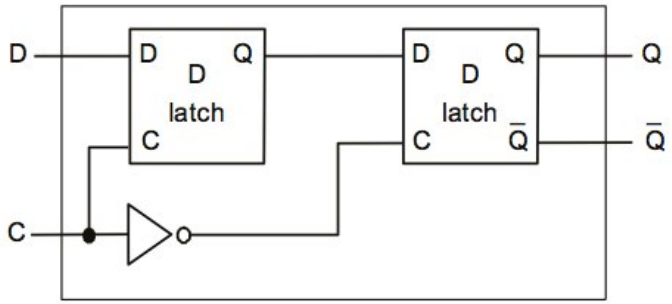


Figure: D Flip-Flop

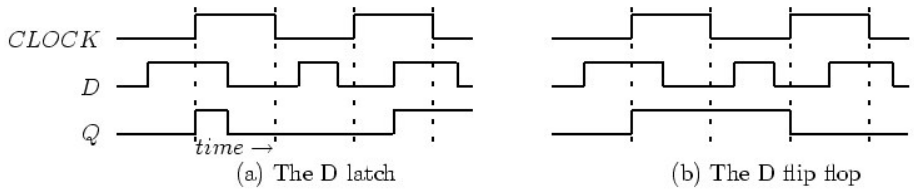


Figure: D Flip-Flop and Latch

- Latch: state changes with change in input and clock being asserted (=1)
- Flip-flop: state changes only on clock rise
  - not as transparent as latch
  - Only flip-flops since using edge-triggered clocking

Registers

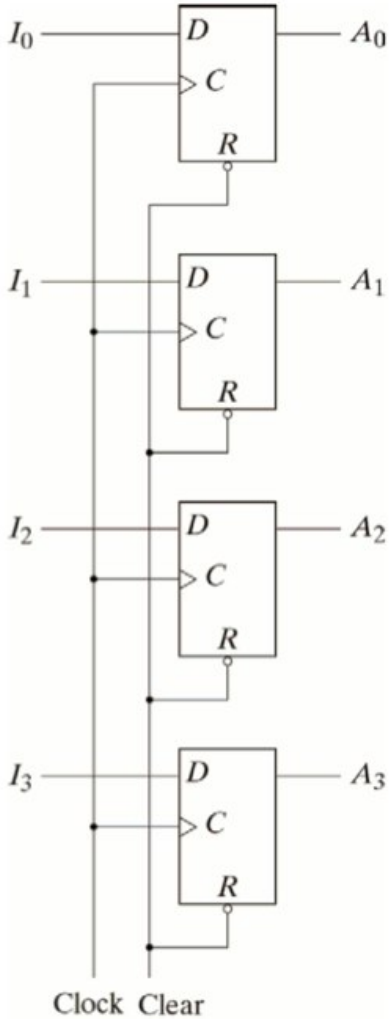


Figure: Simple register made with flip-flop

Register File Design

- Set of registers
  - Read/write by supplying register no. (and data)
  - Uses D Flip-Flop as building block

Register File Design

- Set of registers
  - Read/write by supplying register no. (and data)
  - Uses D Flip-Flop as building block

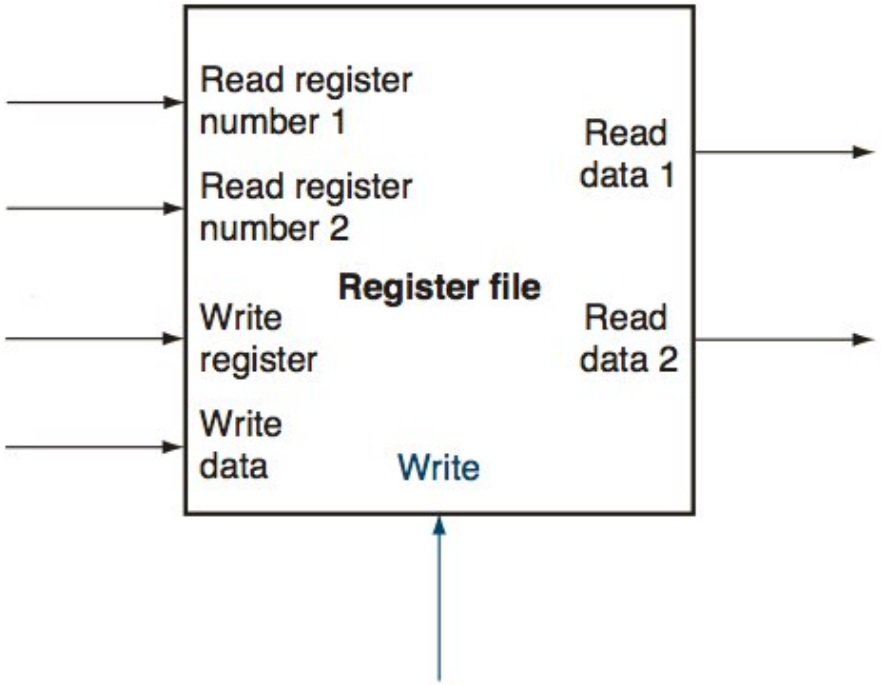


Figure: Register File Design

- 2 read ports, 1 write port

MIPS Register File

- 32, 32-bit registers

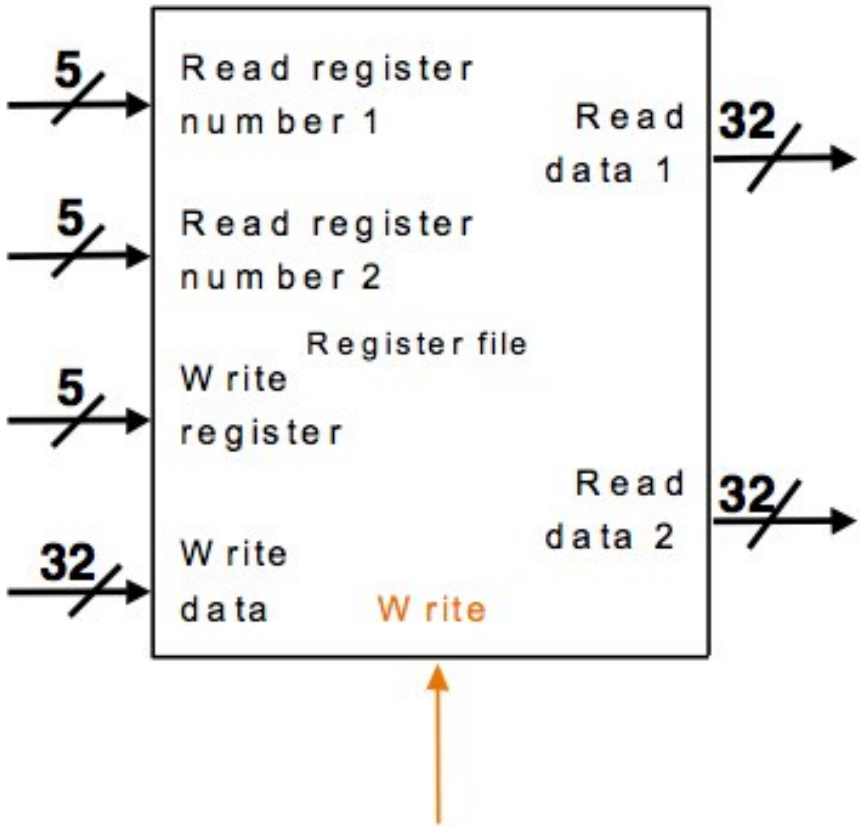


Figure: MIPS registers

- 5-bit read and write lines
- 32-bit data lines

Reading from Register Files

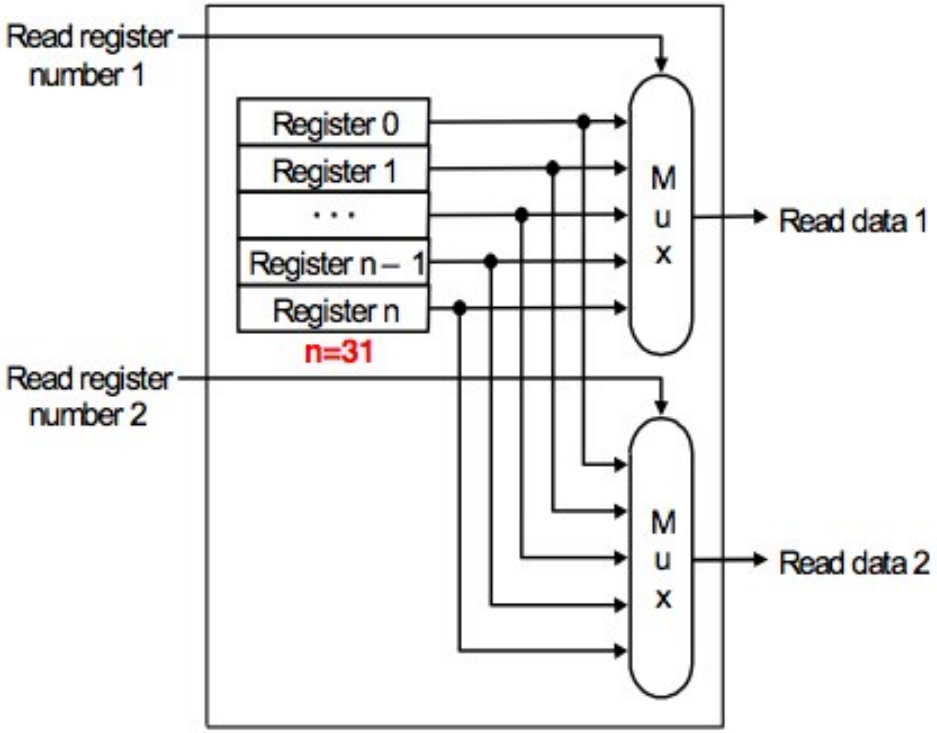


Figure: Reading from RF - internals

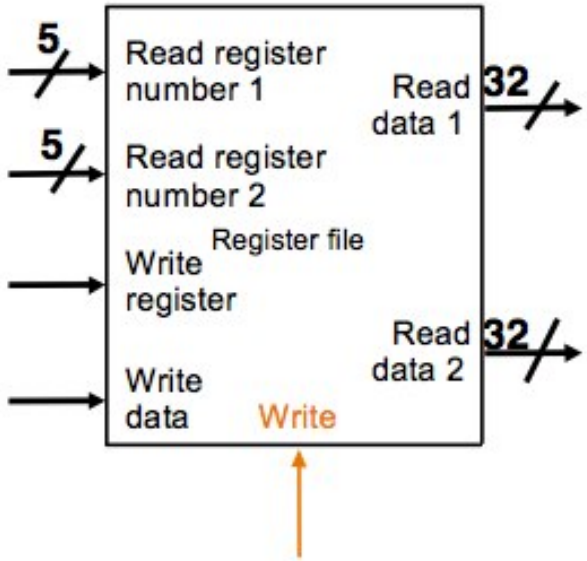


Figure: Reading from RF - the lines

Writing to Register Files

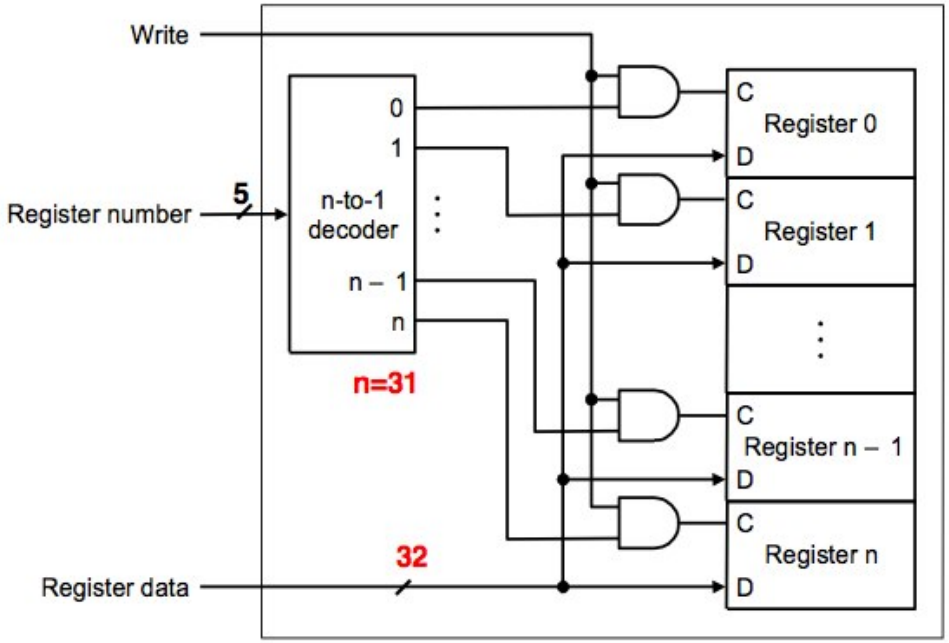


Figure: Writing to RF - internals

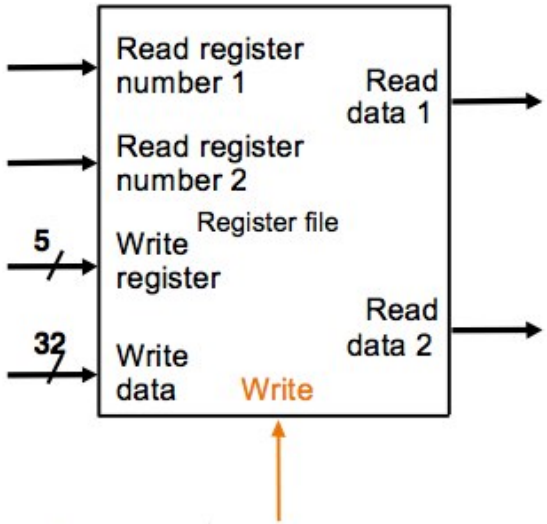


Figure: Writing to RF - the lines

Introduction

- Two technologies
  - **S**tatic **R**andom **A**ccess **M**emory
  - **D**ynamic **R**andom **A**ccess **M**emory
  - both volatile
- Constructed from smaller chips
- Each chip has a configuration:
  - **128M*1**: 128M addressable locations of 1-bit each
  - **16M*8**: 16M addressable locations of 8-bit each

Memory Chip Design

- read/write: 8 bits wide
- 32K addressable locations
- Functioning:
  - CS (Chip Select): 1 for r/w
  - R=0, W=0 ⇒ chip not being accessed
  - R=0, W=1 ⇒ write data in $\texttt{D}_{\texttt{in}}$ at `Address` location
  - R=1, W=0 ⇒ read into $\texttt{D}_{\texttt{out}}$ the value from chip at location `Address`

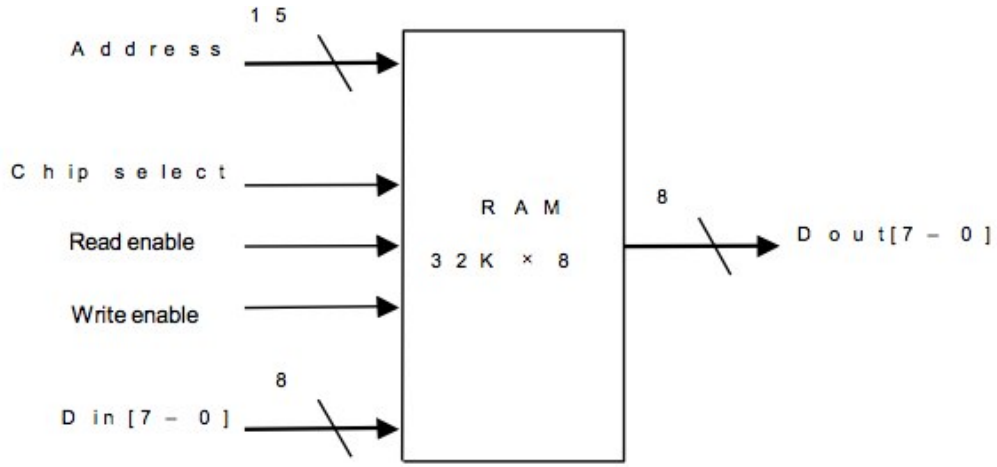


Figure: A 32K*8 RAM

SRAM Structure

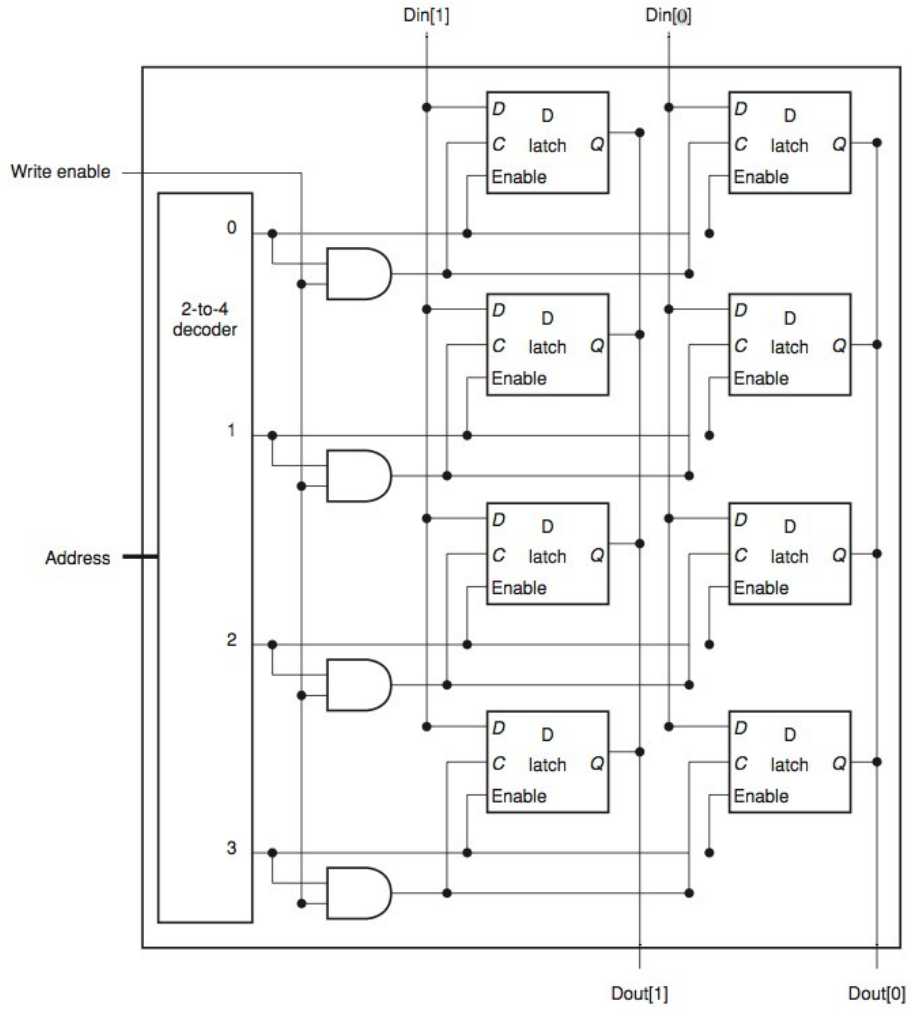


Figure: Bsic Structural Design of 4X2 SRAM Chip

Outline

Introduction

- Implementation of MIPS System
  - **Datapath**: MIPS Processor + Memory
  - also control unit
- **Single cycle design**: all instructions take one clock-cycle

Introduction

- Implementation of MIPS System
  - **Datapath**: MIPS Processor + Memory
  - also control unit
- **Single cycle design**: all instructions take one clock-cycle
  - long clock period (accomodate slowest instruction)
  - cannot reuse resources in a single cycle ⇒ duplication

Datapath Elements

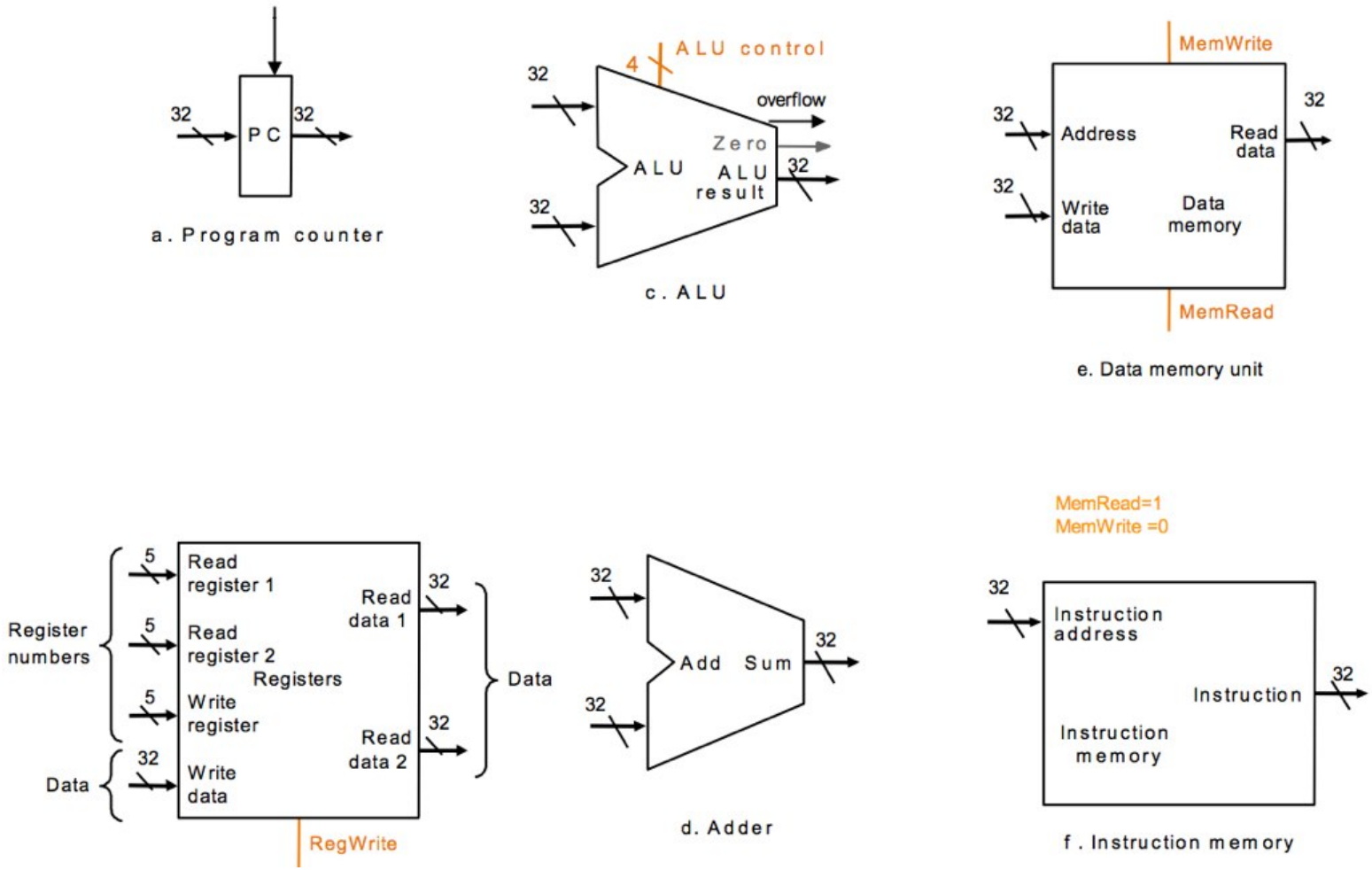


Figure: Elements used in MIPS Datapath

Edge-triggered Methodology

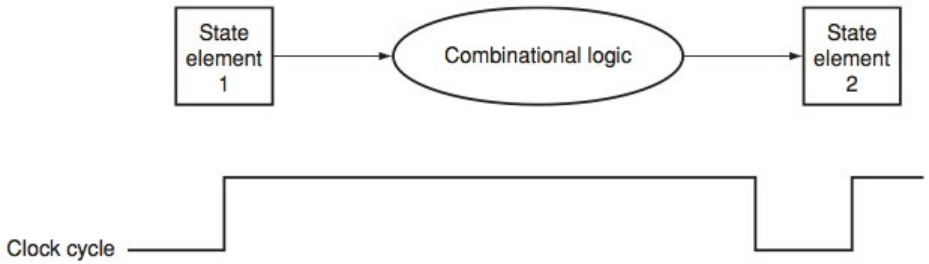


Figure: Edge-triggered Methodology

Edge-triggered Methodology

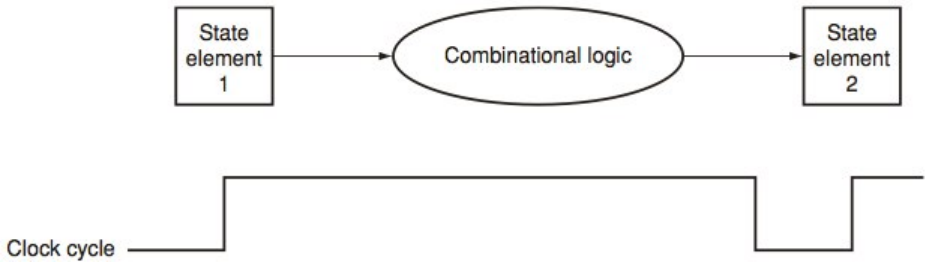


Figure: Edge-triggered Methodology

- Execution strategy:
  - read content of a state element at *beginning* of clock cycle
  - send values through a combinational element
  - write result to a state element at *end* of clock cycle
- edge-triggered ⇒ read **and** write in same cycle

PC Increment

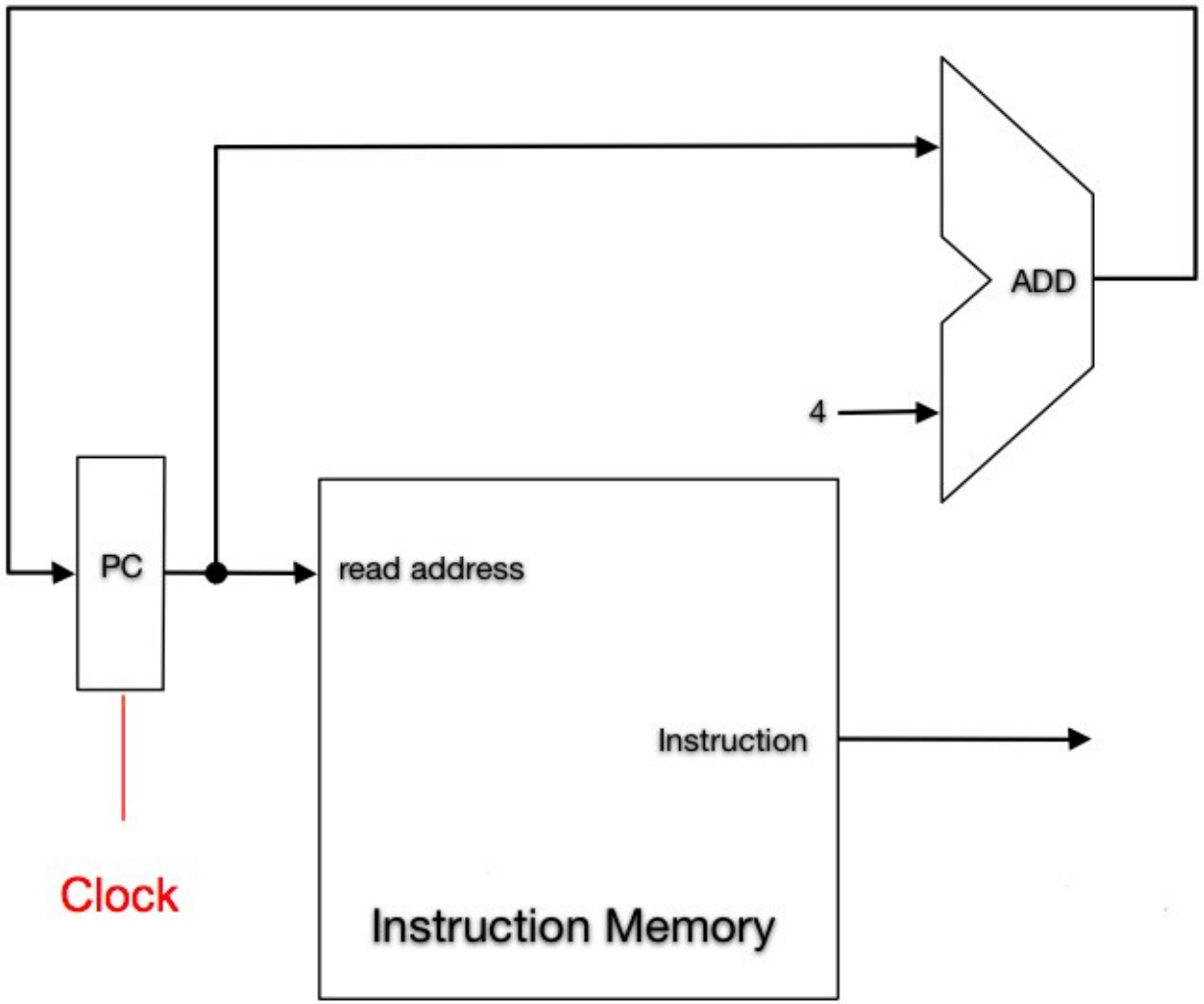
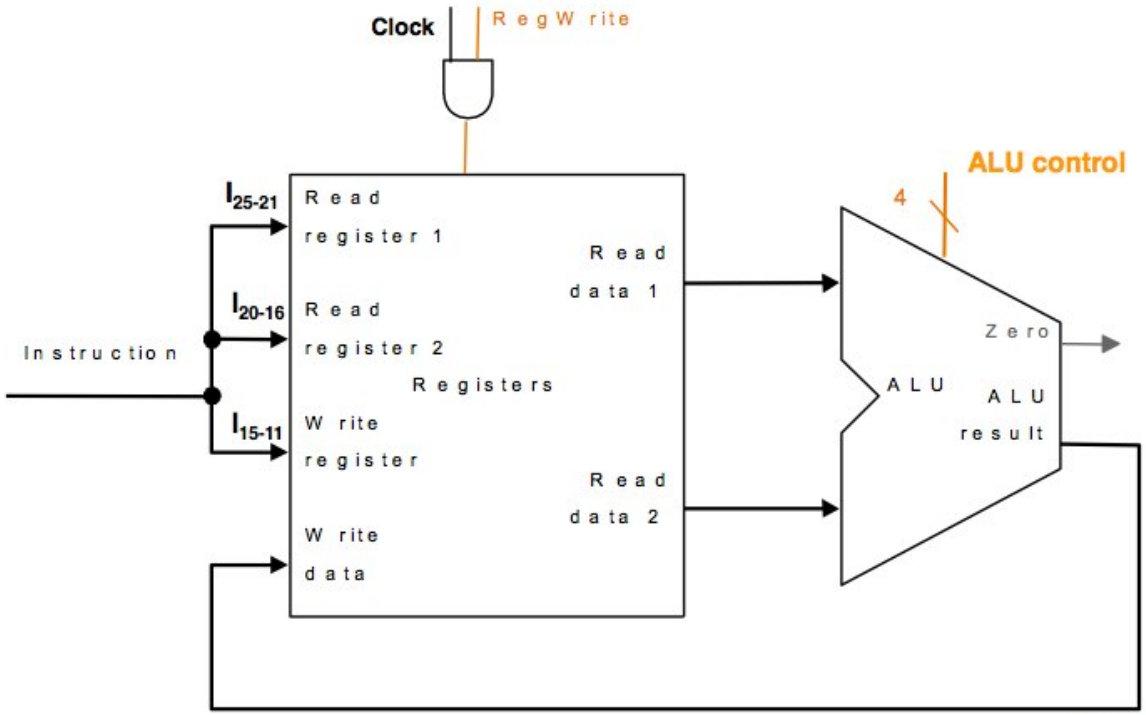


Figure: PC Increment Circuit

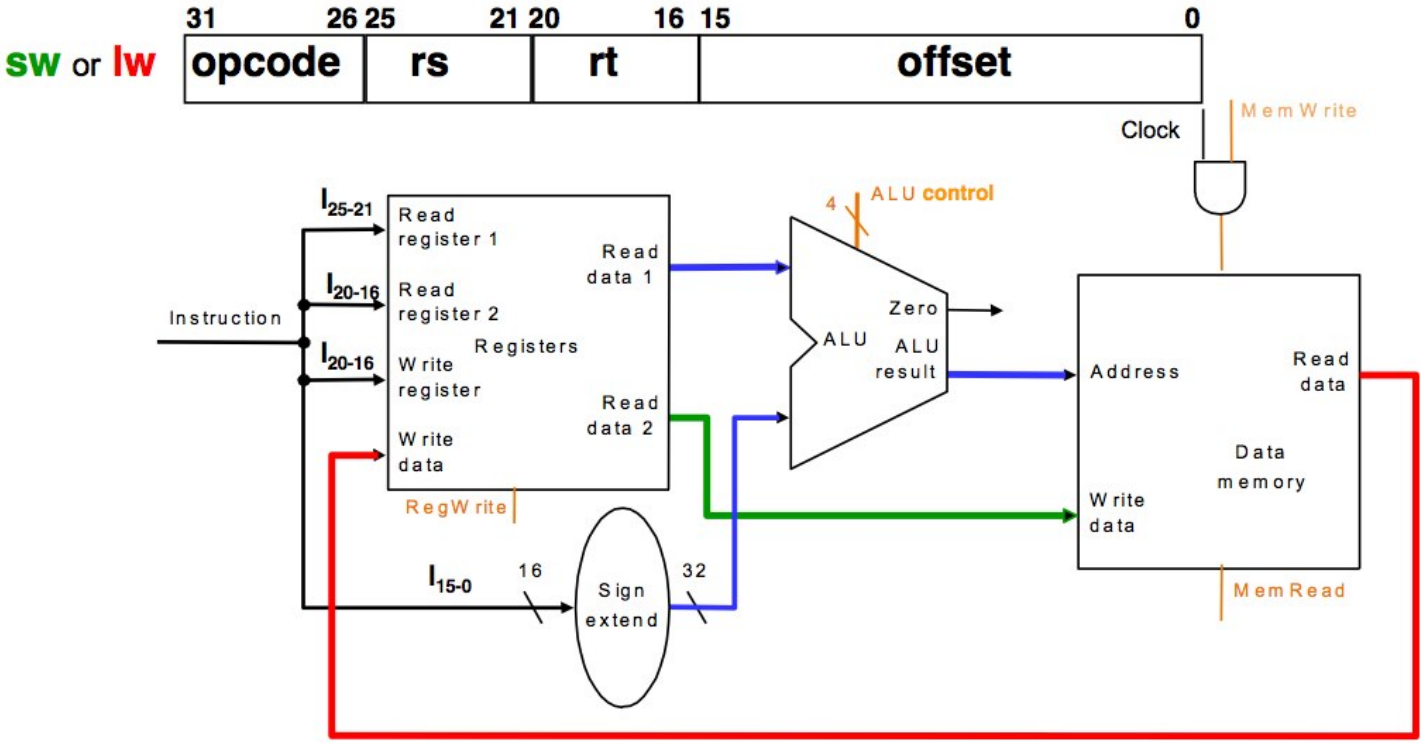
Datapath for R-type Instructions



R-type

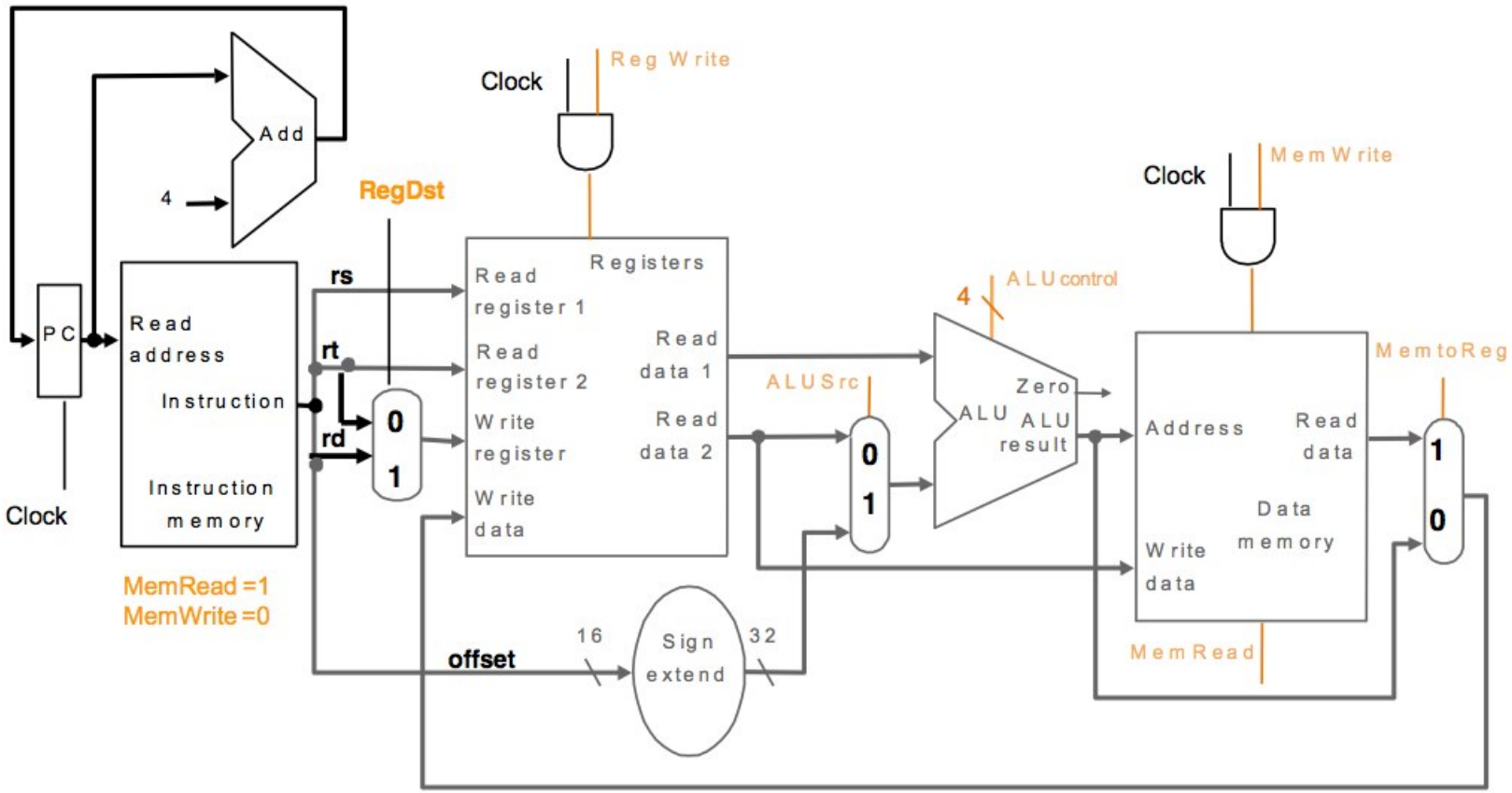
add = 32
sub = 34
slt = 42
and = 36
or = 37
nor = 39

Datapath for load/save Instructions

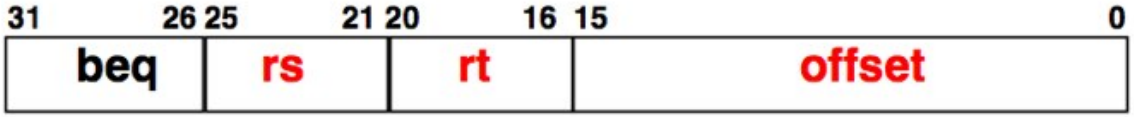


- Control signals:
  - **lw, sw**: ALU control=0010 (for address calculations)
  - **sw**: Memread=0, Memwrite=1, Regwrite=0
  - **lw**: Memread=1, Memwrite=0, Regwrite=1

Datapath for lw, sw and R-type Instructions



Datapath for beq Instruction



Branch target = [PC] + 4 + 4×offset

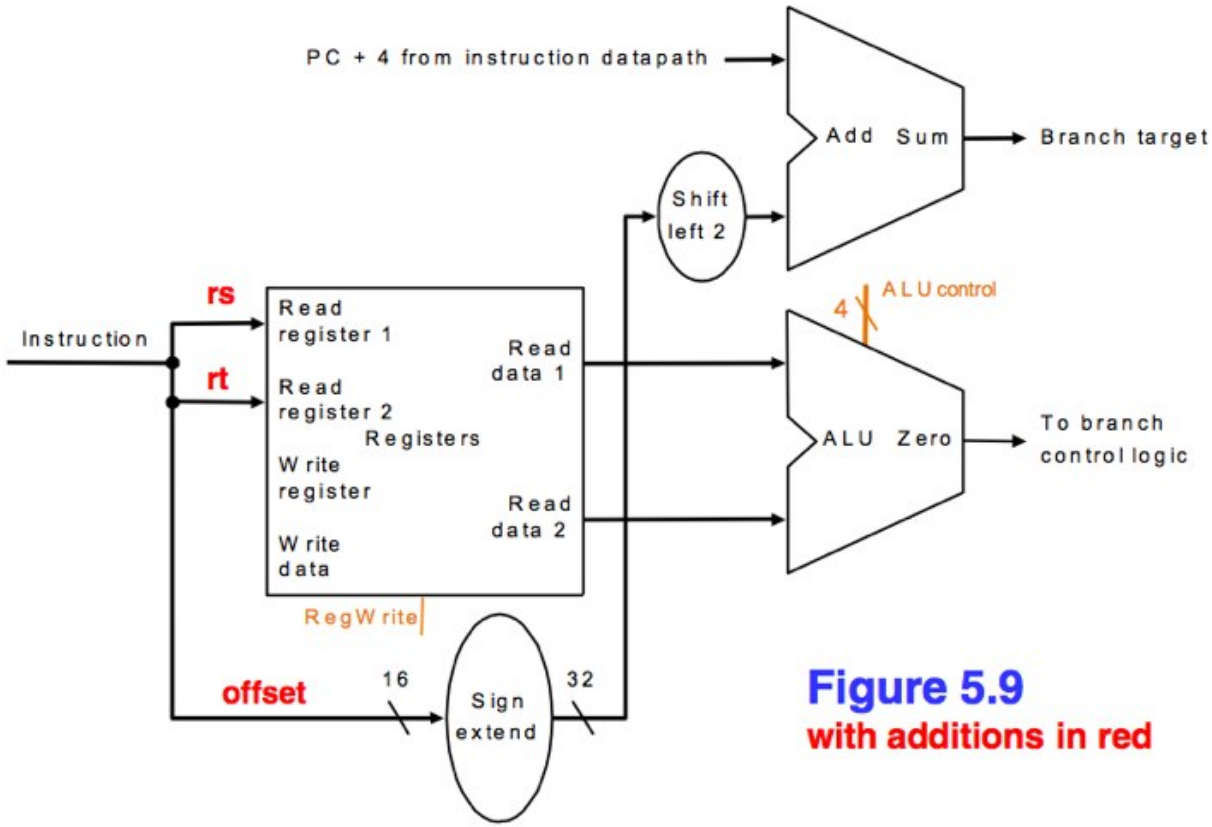
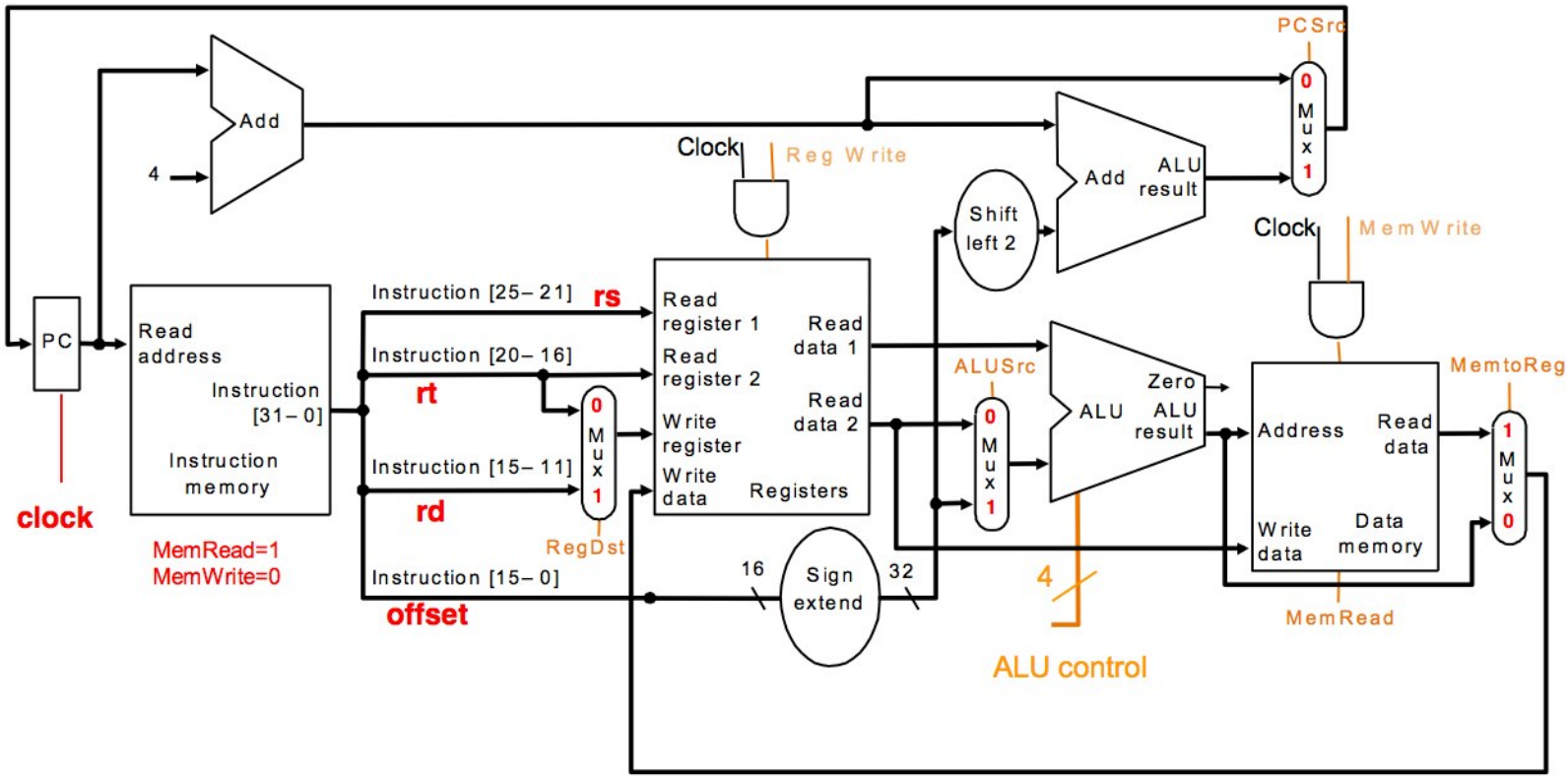
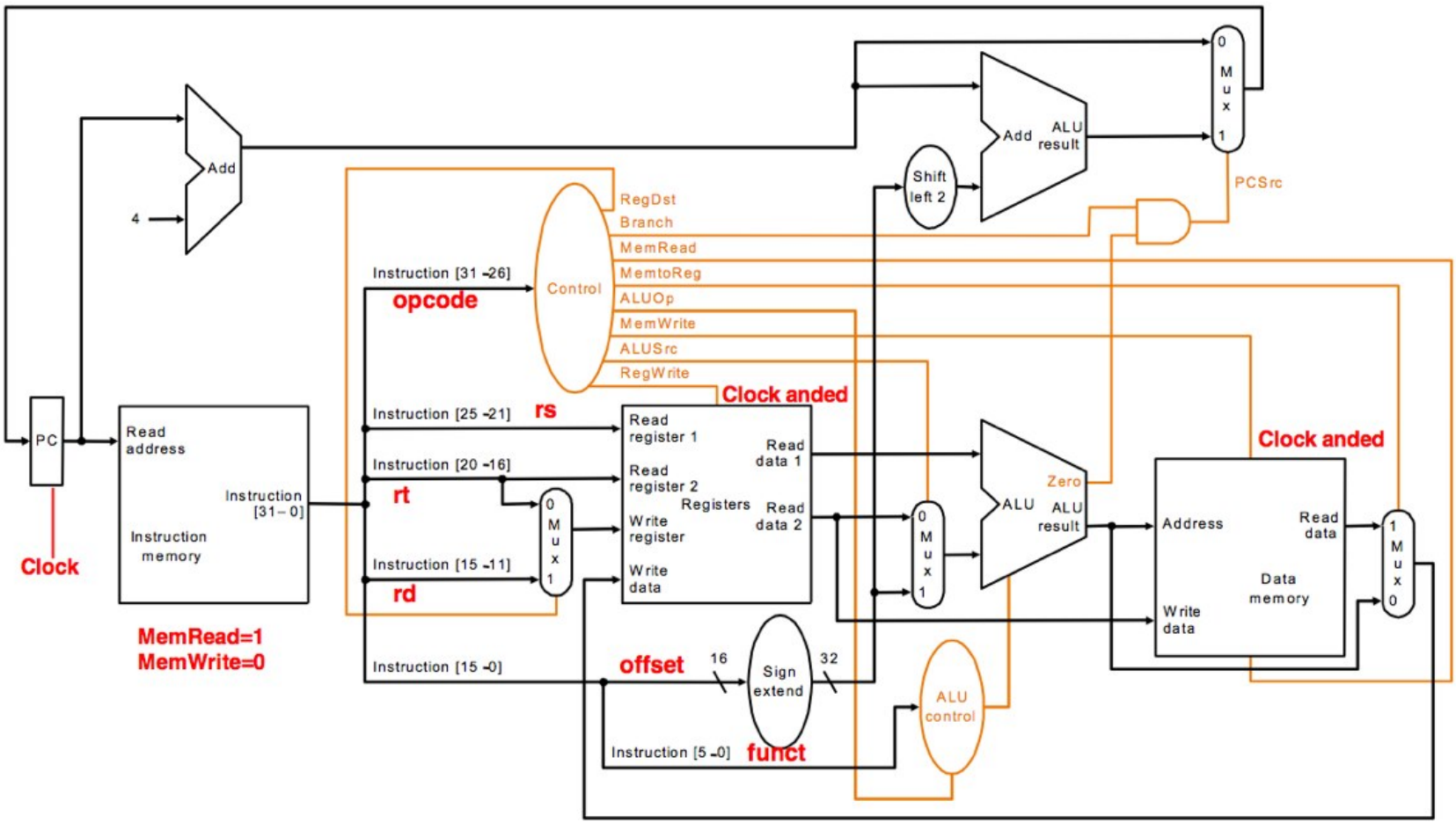


Figure 5.9
with additions in red

Datapath for R-type, lw/sw, beq Instructions



Datapath and Control Circuit

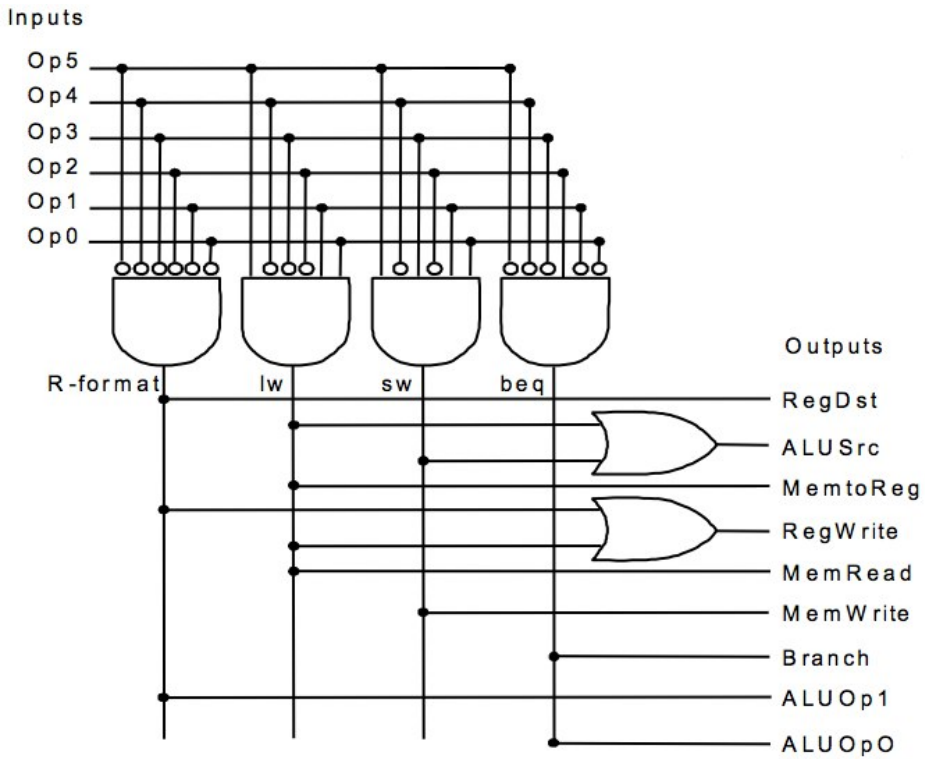


Control Signals and Opcode

	Input	Output							
	Op-code	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-type	000000	1	0	0	1	0	0	0	1	0
lw	100011	0	1	1	1	1	0	0	0	0
sw	101011	0	1	0	0	0	1	0	0	0
beq	000100	0	0	0	0	0	0	1	0	1

Figure: Control signals depend on the Opcode

Control Signal Generation



Datapath for R-type, lw/sw, beq, j Instructions

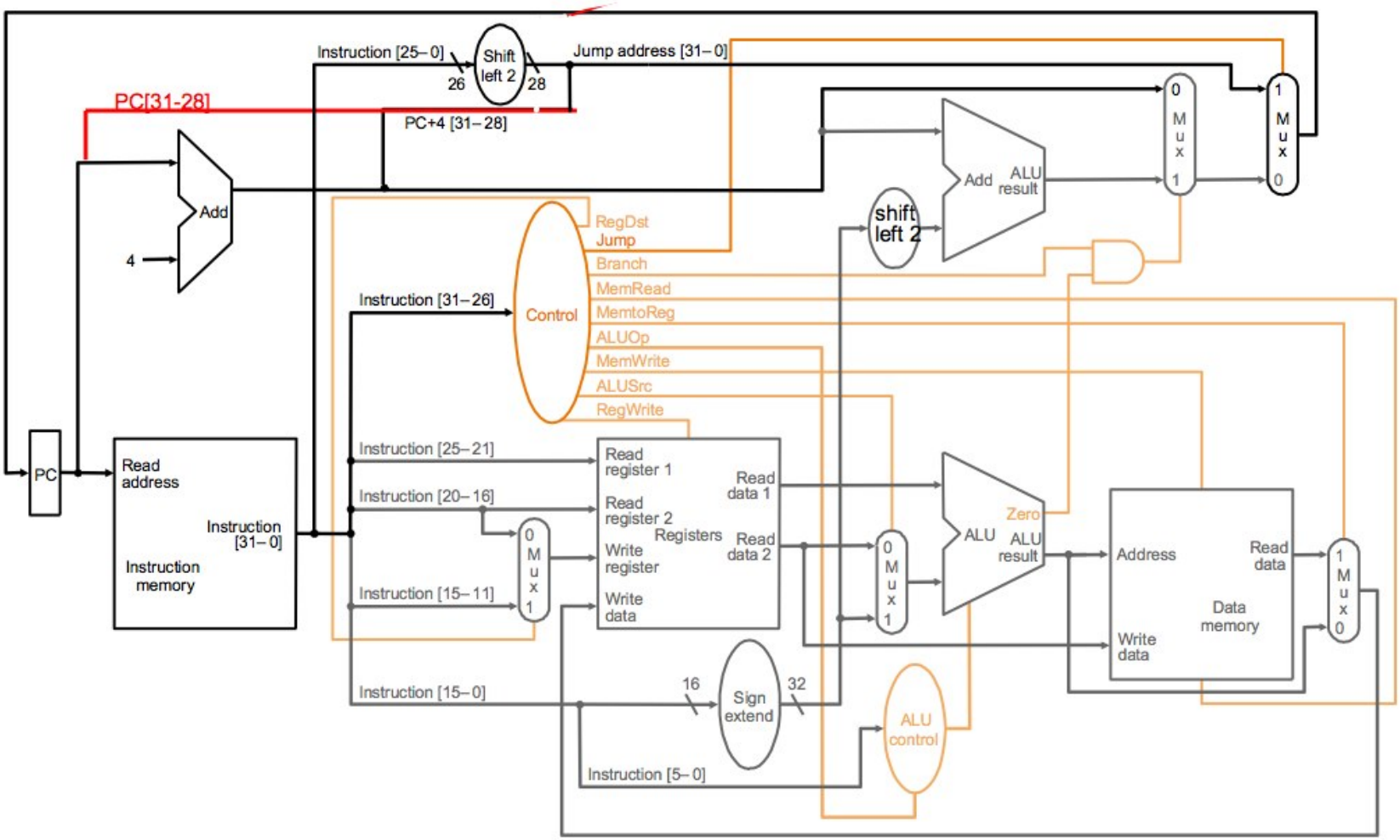
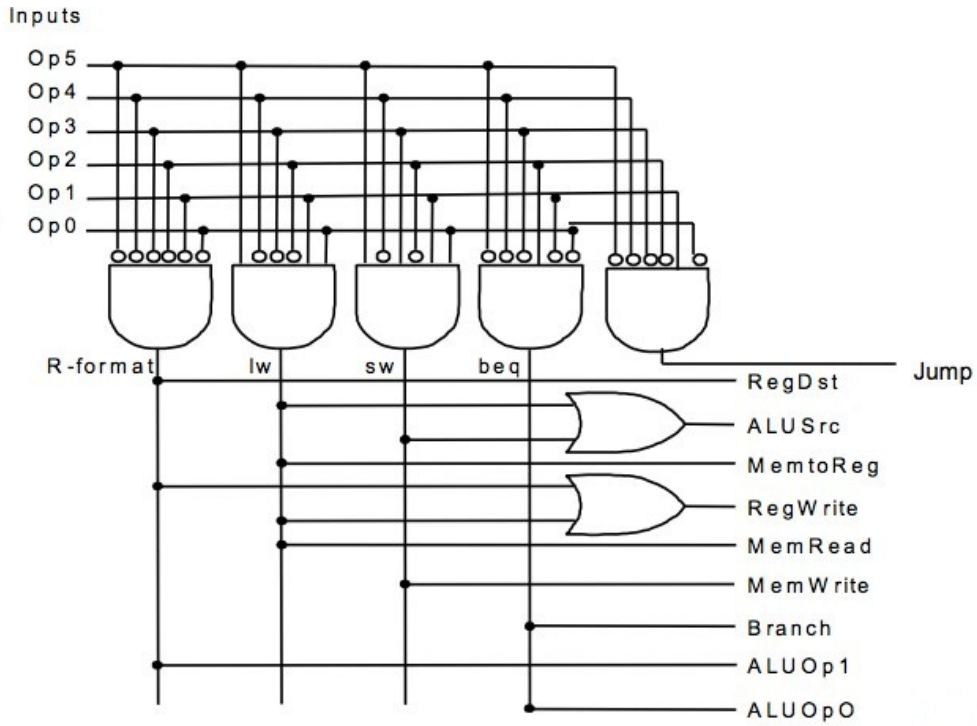


Figure: The complete datapath

Control Circuit with Jump Included

$$\text{Jump} = \overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}\,Op_1\,\overline{Op_0}$$



About Single-cycle design

- Inefficient design
- Single-cycle design ⇒ CPI = ?

About Single-cycle design

- Inefficient design
- Single-cycle design ⇒ CPI = ?
  - CPI = 1
- Slowest instruction ≡ Longest possible machine path

About Single-cycle design

- Inefficient design
- Single-cycle design ⇒ CPI = ?
  - CPI = 1
- Slowest instruction ≡ Longest possible machine path
  - the `load` instruction; uses 5 functional units:
    - instruction memory
    - register file
    - ALU
    - data memory
    - register file

About Single-cycle design

- Inefficient design
- Single-cycle design ⇒ CPI = ?
  - CPI = 1
- Slowest instruction ≡ Longest possible machine path
  - the `load` instruction; uses 5 functional units:
    - instruction memory
    - register file
    - ALU
    - data memory
    - register file
- Fastest?

About Single-cycle design

- Inefficient design
- Single-cycle design ⇒ CPI = ?
  - CPI = 1
- Slowest instruction ≡ Longest possible machine path
  - the `load` instruction; uses 5 functional units:
    - instruction memory
    - register file
    - ALU
    - data memory
    - register file
- Fastest?
  - probably `jump`

About Single-cycle design

- Acceptable if fewer instructions

About Single-cycle design

- Acceptable if fewer instructions
  - used in older, simpler ISA implementations

About Single-cycle design

- Acceptable if fewer instructions
  - used in older, simpler ISA implementations
- terrible for ISA with complex instructions, such as *floating point* operations

About Single-cycle design

- Acceptable if fewer instructions
  - used in older, simpler ISA implementations
- terrible for ISA with complex instructions, such as *floating point* operations
- Dual problems:
  - violates *"make the common-case faster"* principle (performance)
  - need to duplicate hardware (cost)

Improvements over Single-cycle Design

- Two ways to improve (performance and cost):
  - **multi-cycle design**:
    - some instructions run faster than others

Improvements over Single-cycle Design

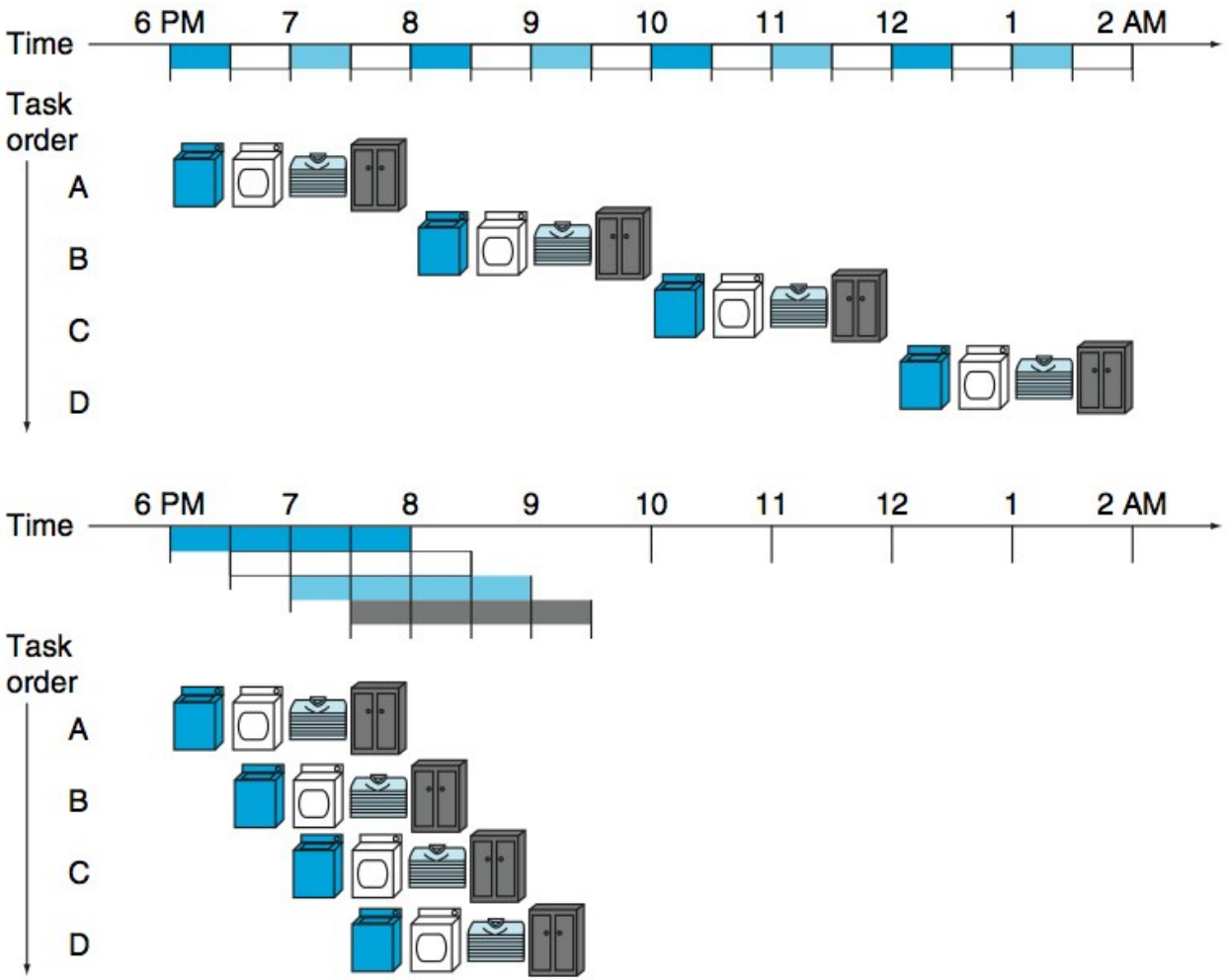
- Two ways to improve (performance and cost):
  - **multi-cycle design**:
    - some instructions run faster than others
  - **Pipelining**:
    - Overlap execution of instructions

Pipelining

Pipelining: Introduction

- Run multiple instructions in parallel
- Improves performance and hardware utilization
- similar to *assembly line, laundry cleaning*

Example of Pipelining



Introduction

- First: identify steps in instruction execution

Introduction

- First: identify steps in instruction execution
- Five steps in any MIPS instruction:
  - **Fetch instruction**
  - **Read registers** (while simultaneously decoding)
  - **Execute operation / calculate address**
  - **Access data memory**
  - **Write results to register**

Execution Time Improvement with Pipelining

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Figure: Total time for each instruction

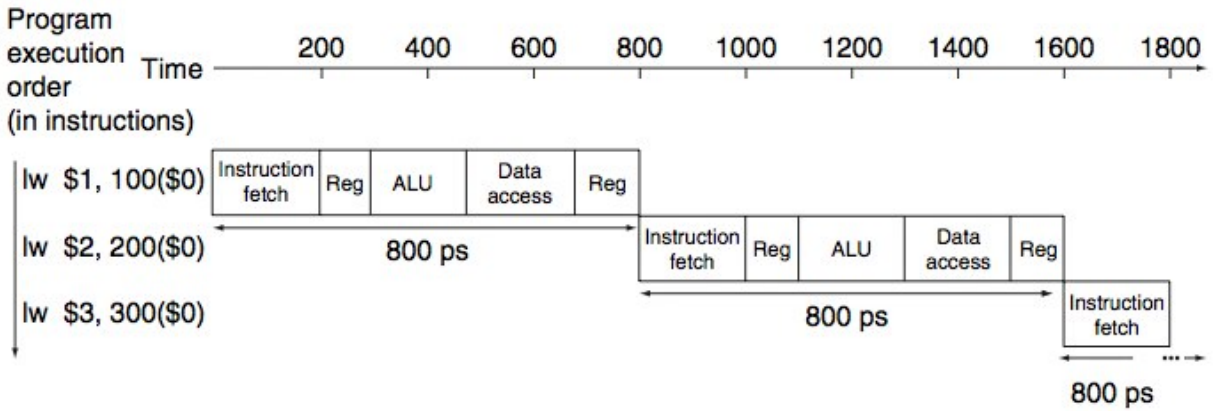


Figure: Single-cycle non-pipelined execution

Execution Time Improvement with Pipelining

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Figure: Total time for each instruction

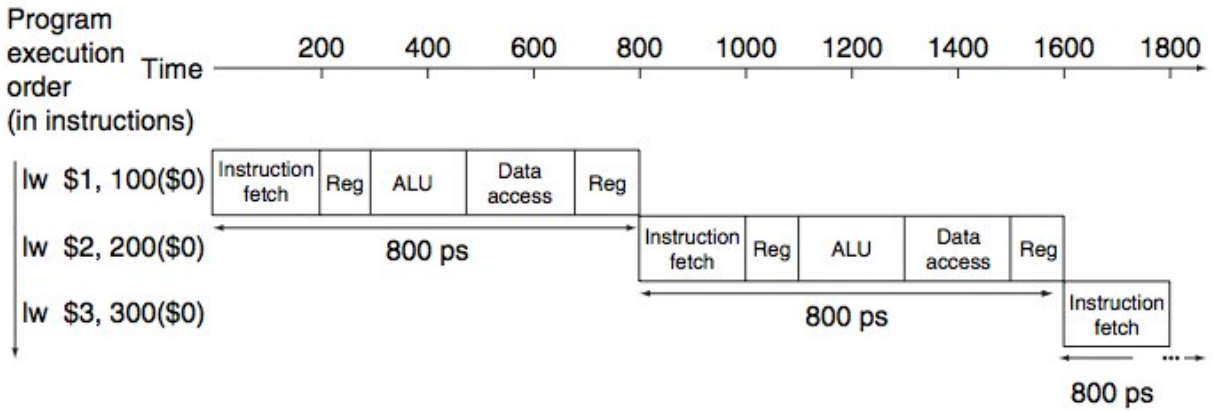


Figure: Single-cycle non-pipelined execution

- Total execution time: 800 * 3 = 2400 ps

Execution Time Improvement with Pipelining

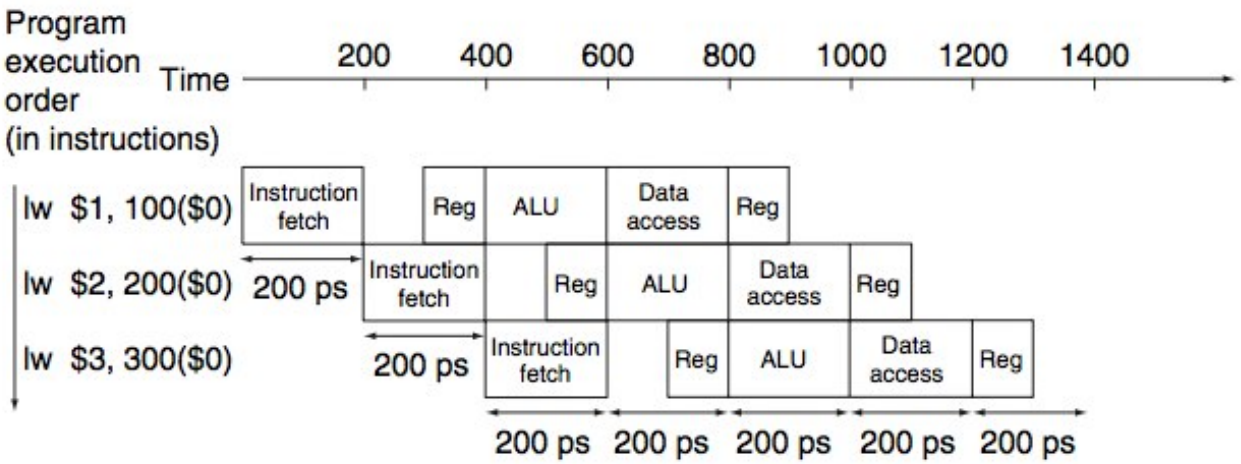


Figure: Pipelined execution

- Total execution time: << 2400 ps

Pipelined Execution

- Clock cycle relates to single operation, rather than an instruction
- Accomodate the slowest *operation*: 200 ps
- Read and write to register can happen in different halves of same cycle

Pipelining the Datapath

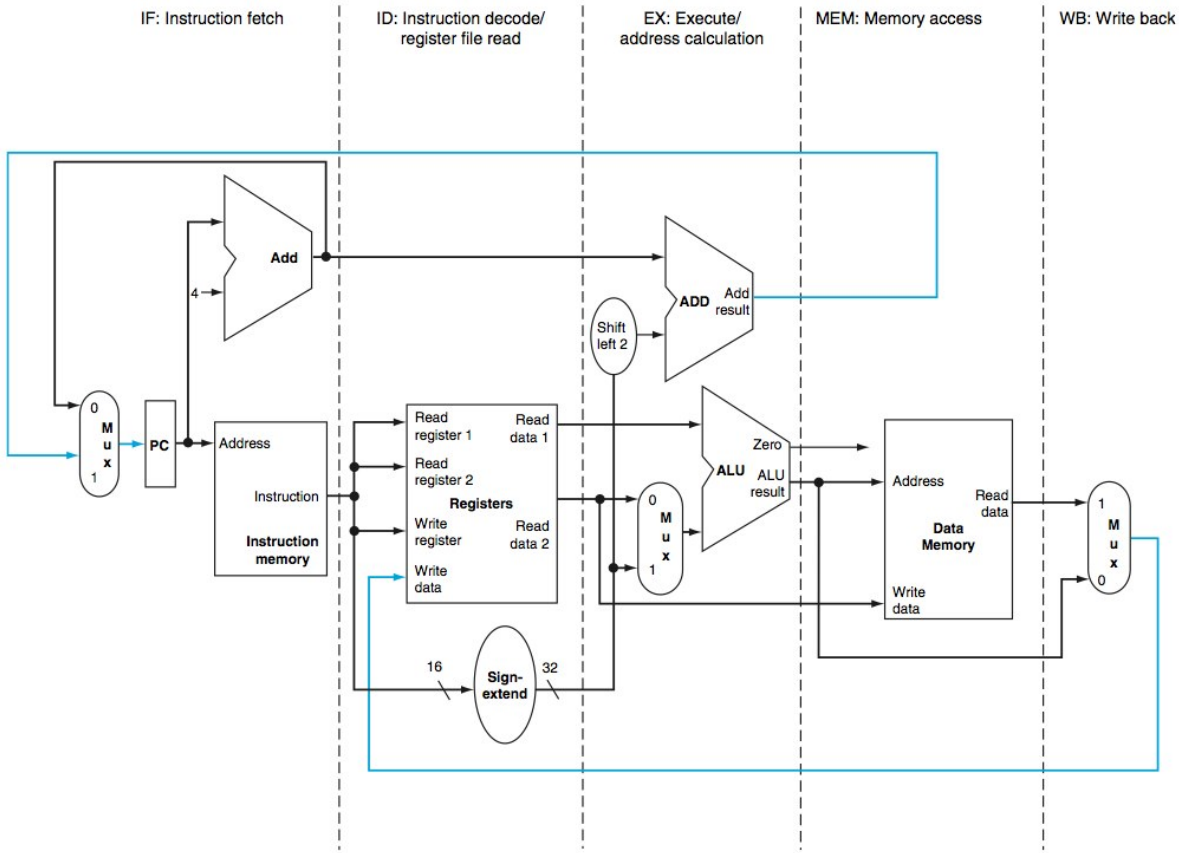


Figure: Datapath without pipelining

- Data flows left-to-right through stages, except
  - *write to register* and *write to PC*
  - does not affect current instruction

Instruction Execution in Pipeline

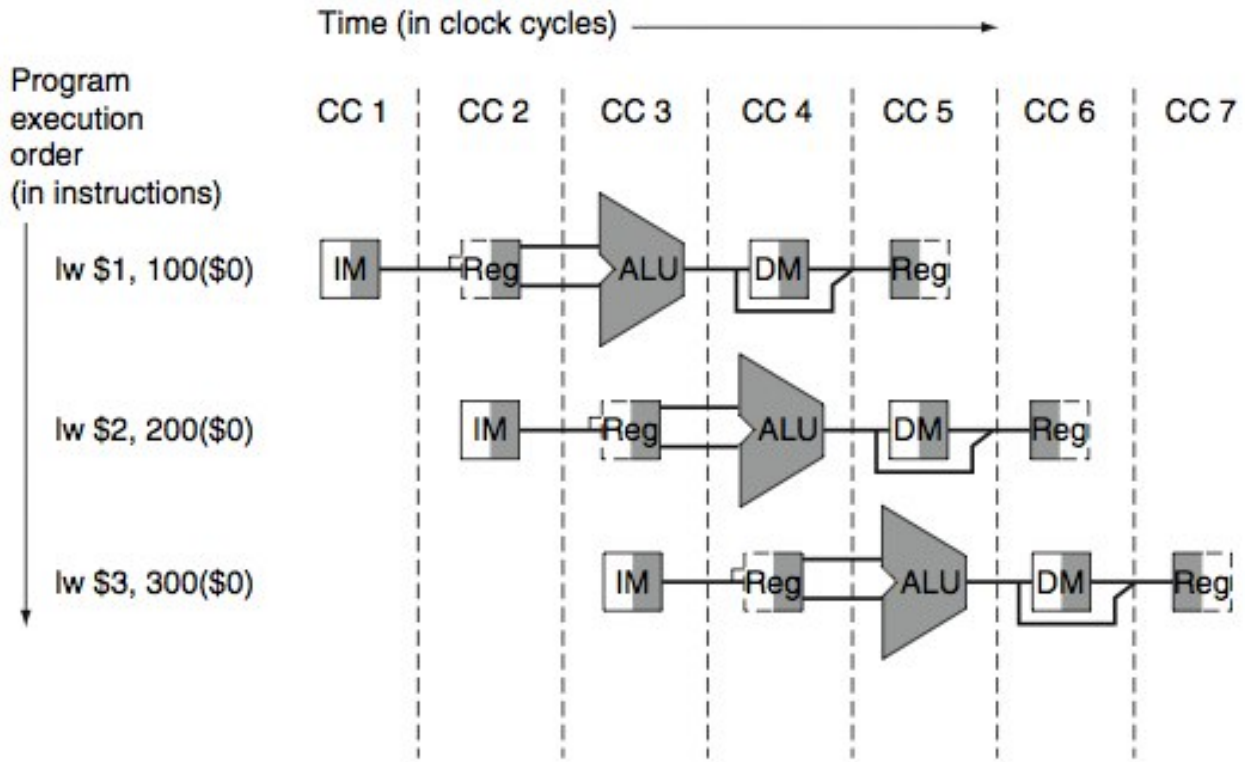


Figure: Instruction execution in single-cycle datapath with pipeline

- *Virtually* every instruction has its own datapath, but *staggered*

Pipelinig the Datapath

- Need to store data for an instruction as it passes through the datapath
  - for e.g., the value read from IM must be stored so it's available for later stages
  - $\Rightarrow$ add registers at every stage
- Each instruction advances to next stage on clock cycle

Pipelining the Datapath

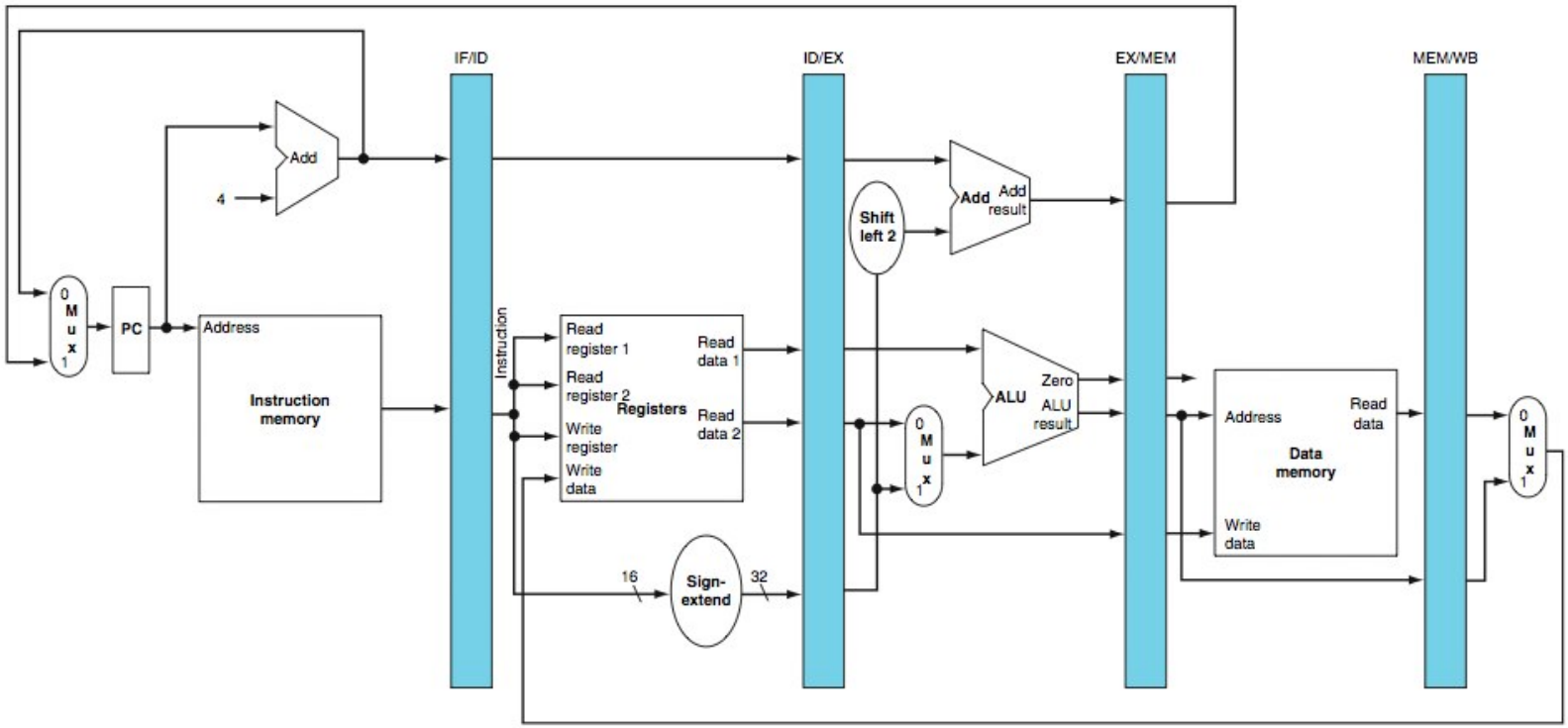


Figure: Pipelined Datapath

Example for load instruction

First Stage

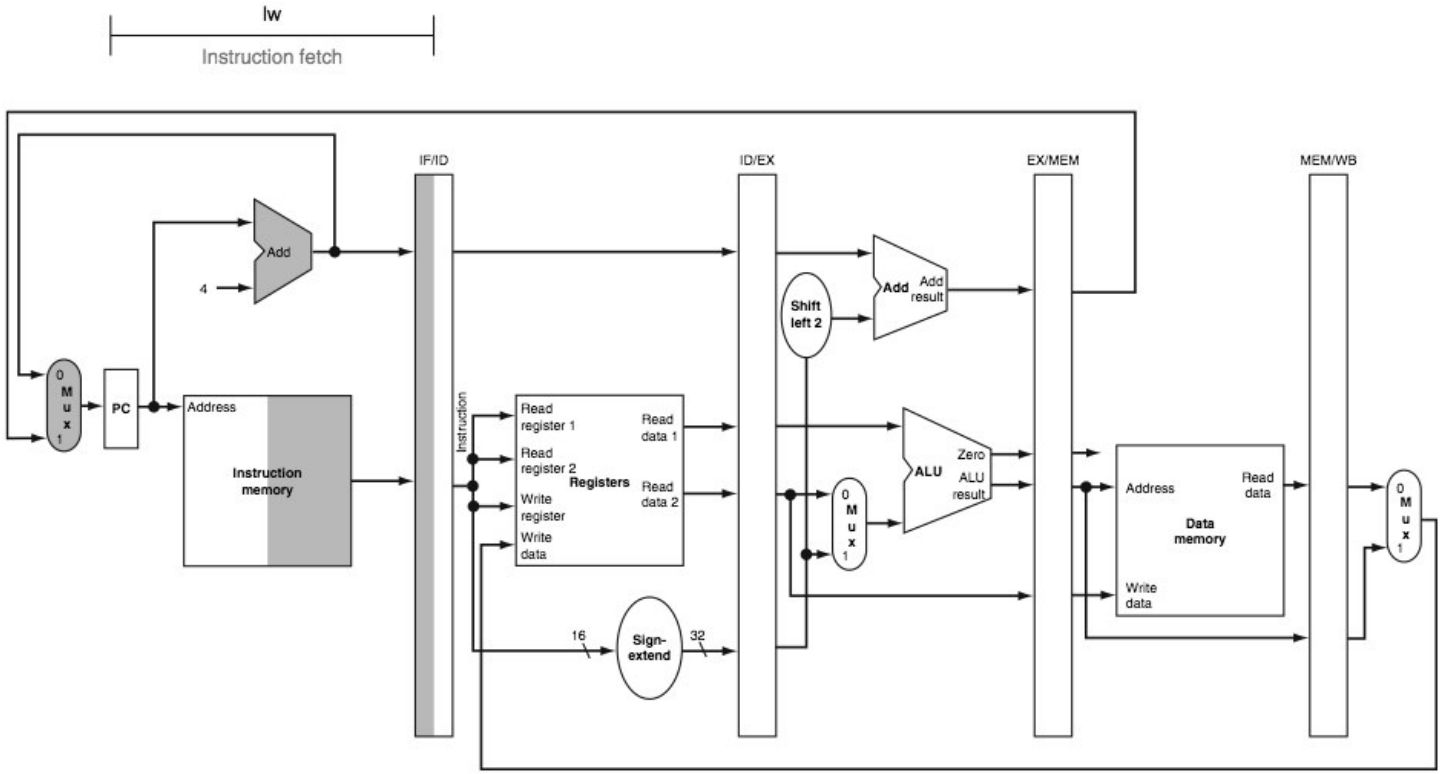


Figure: Instruction Fetch

First Stage

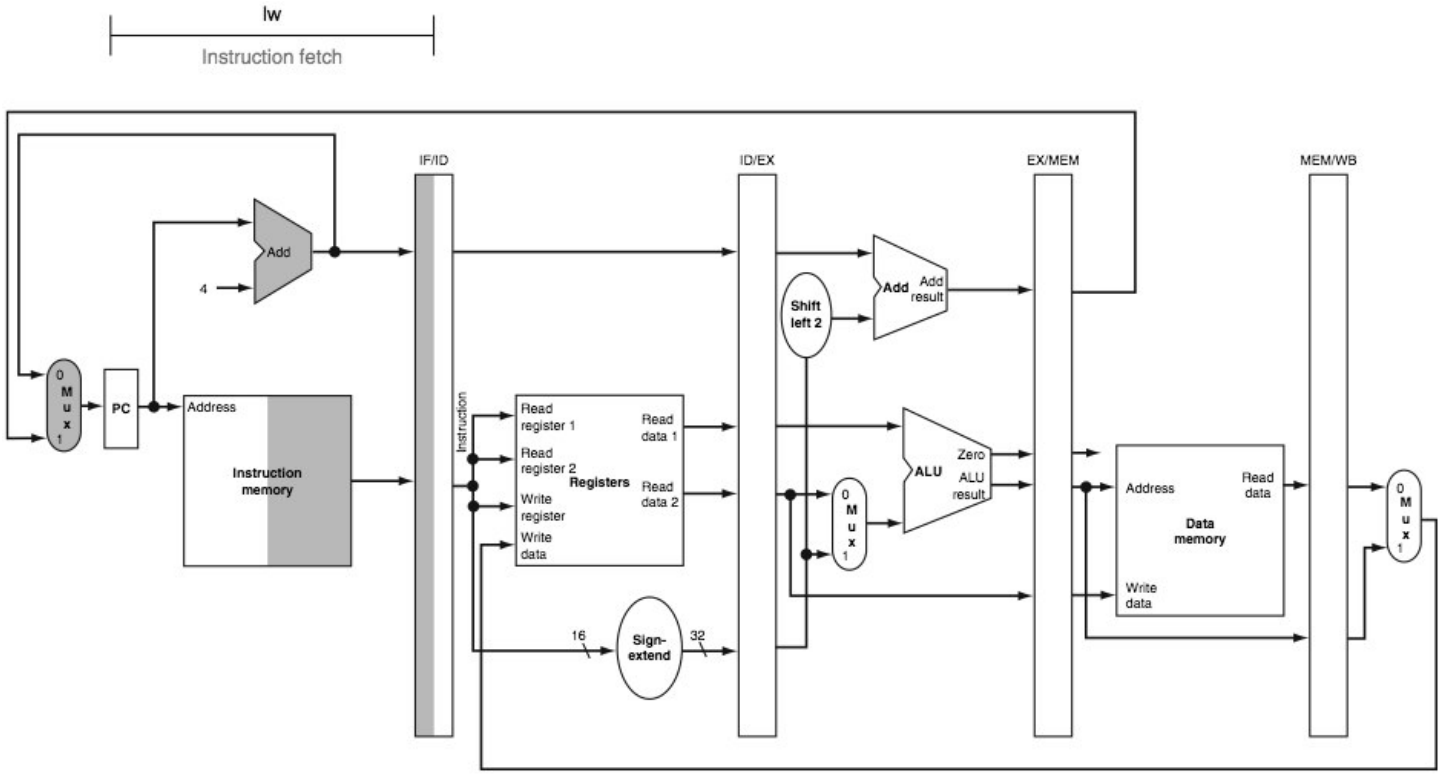


Figure: Instruction Fetch

- Fetch instruction; Increment PC (save in IF/ID register also)

Second Stage

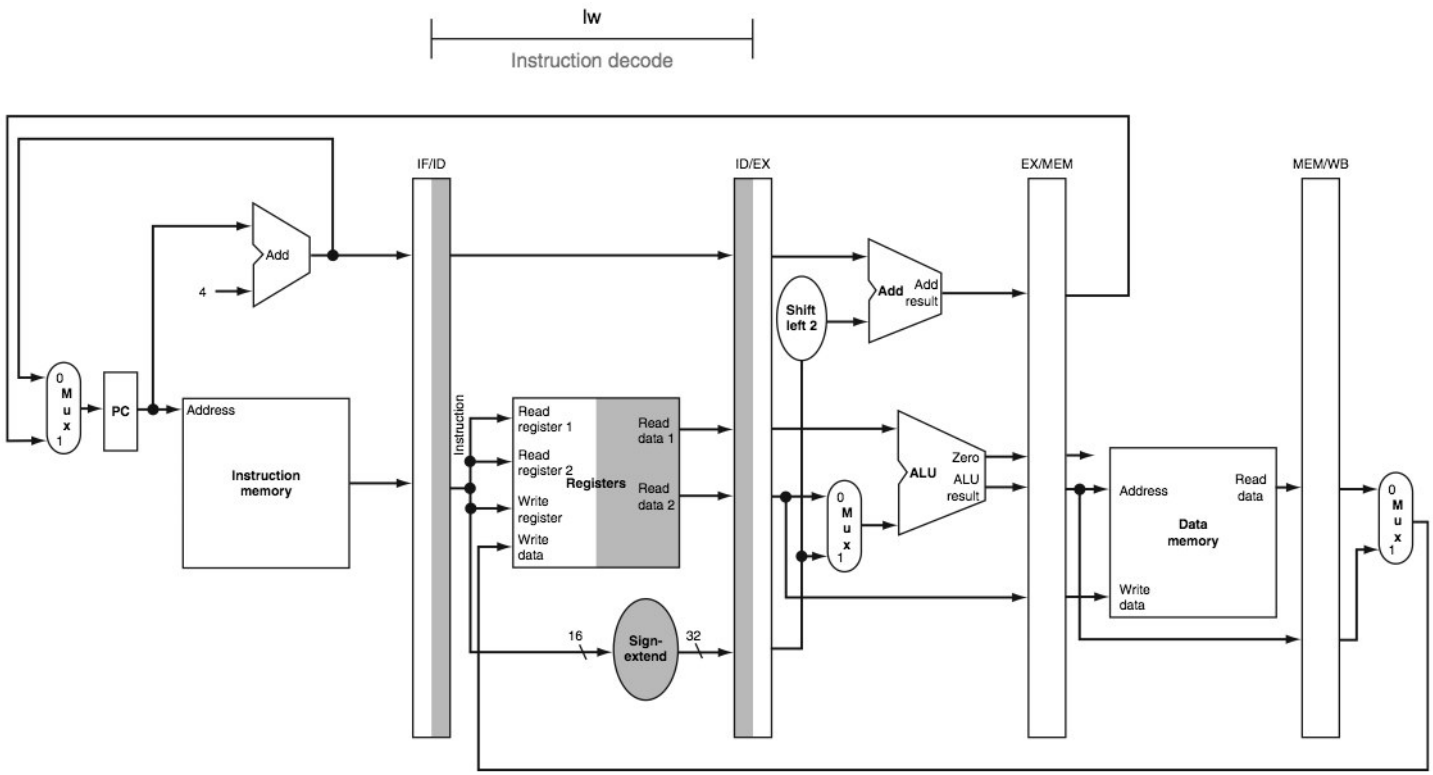


Figure: Instruction decode and register read

Second Stage

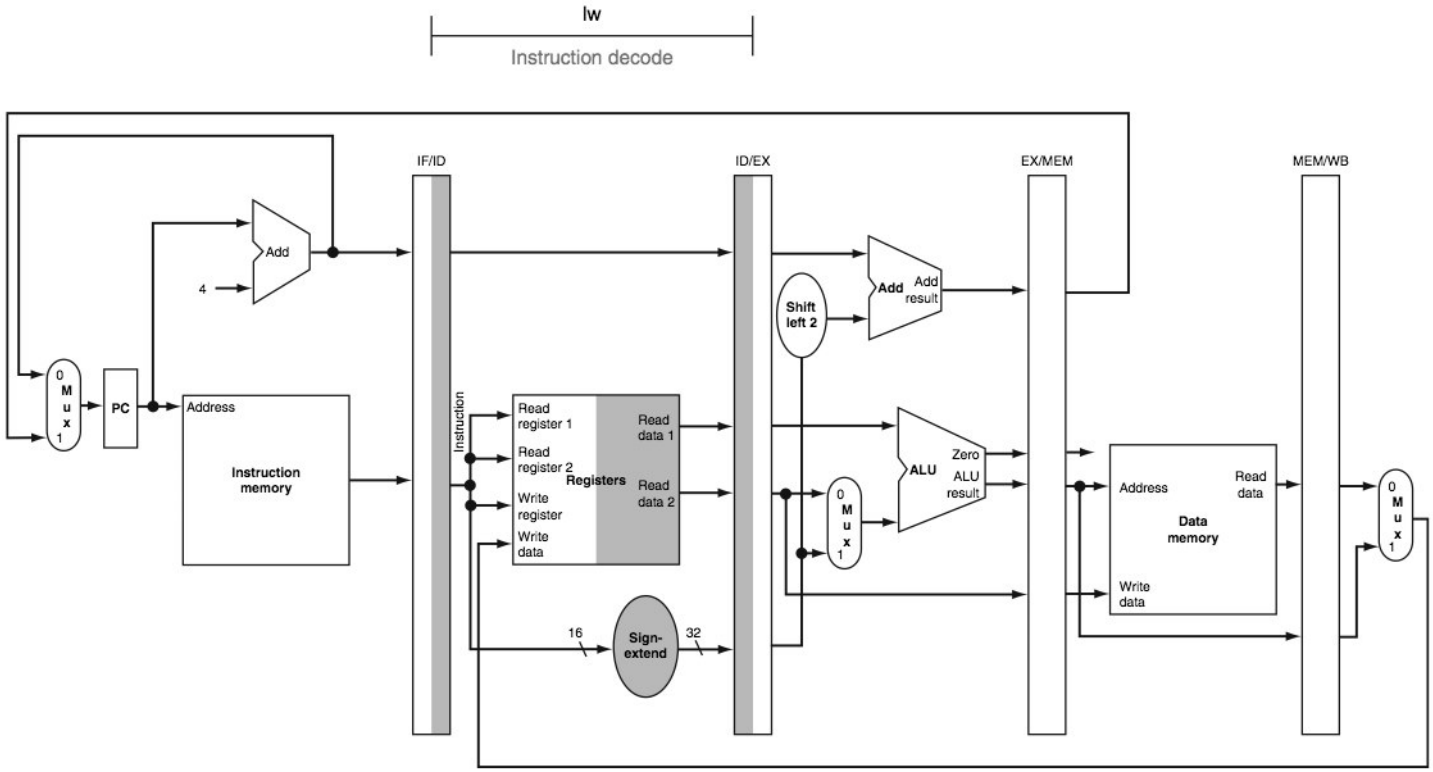


Figure: Instruction decode and register read

- Store in ID/EX registers:
  - Incremented PC address
  - 2 register values
  - sign extended offset

Third Stage

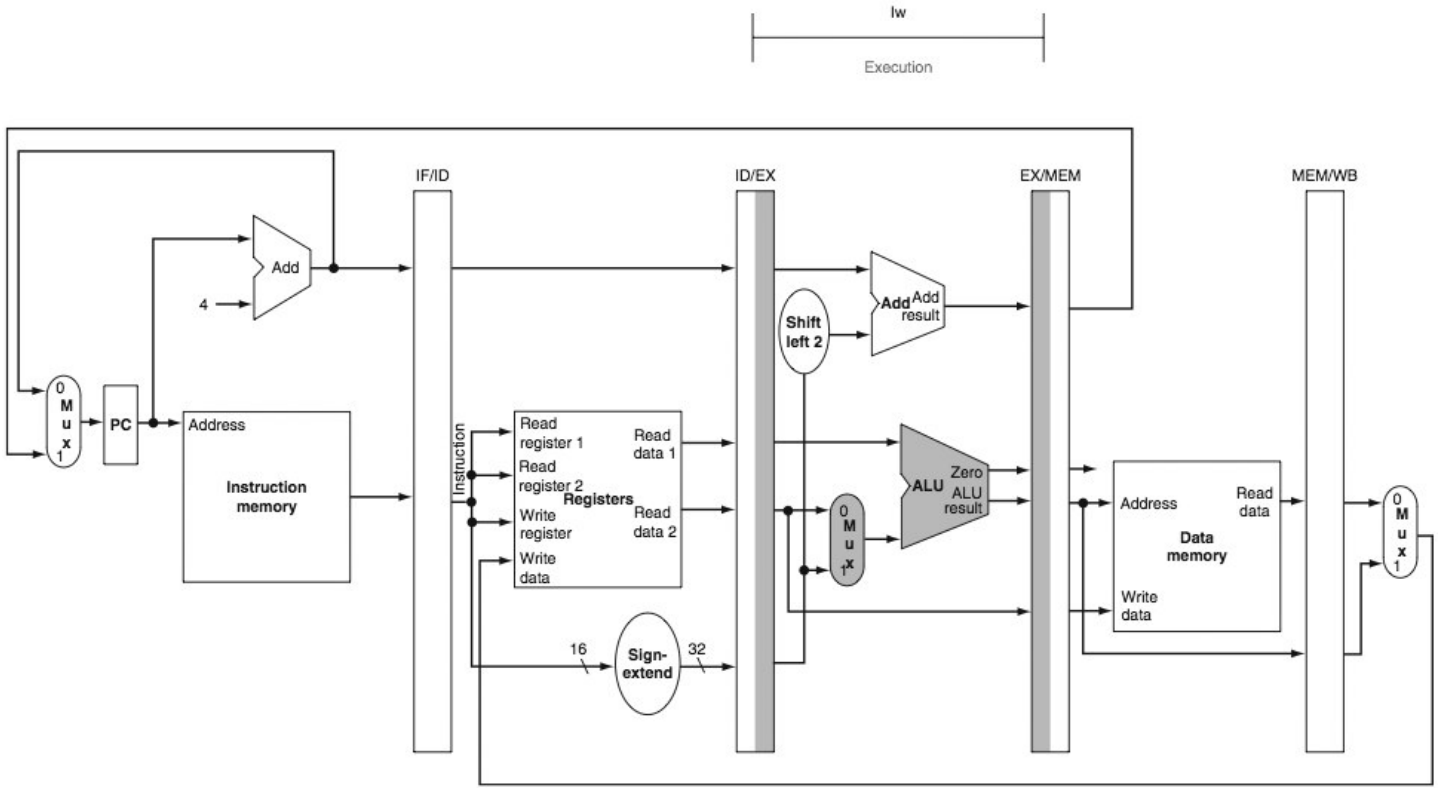


Figure: Execute or Address Calculation

Third Stage

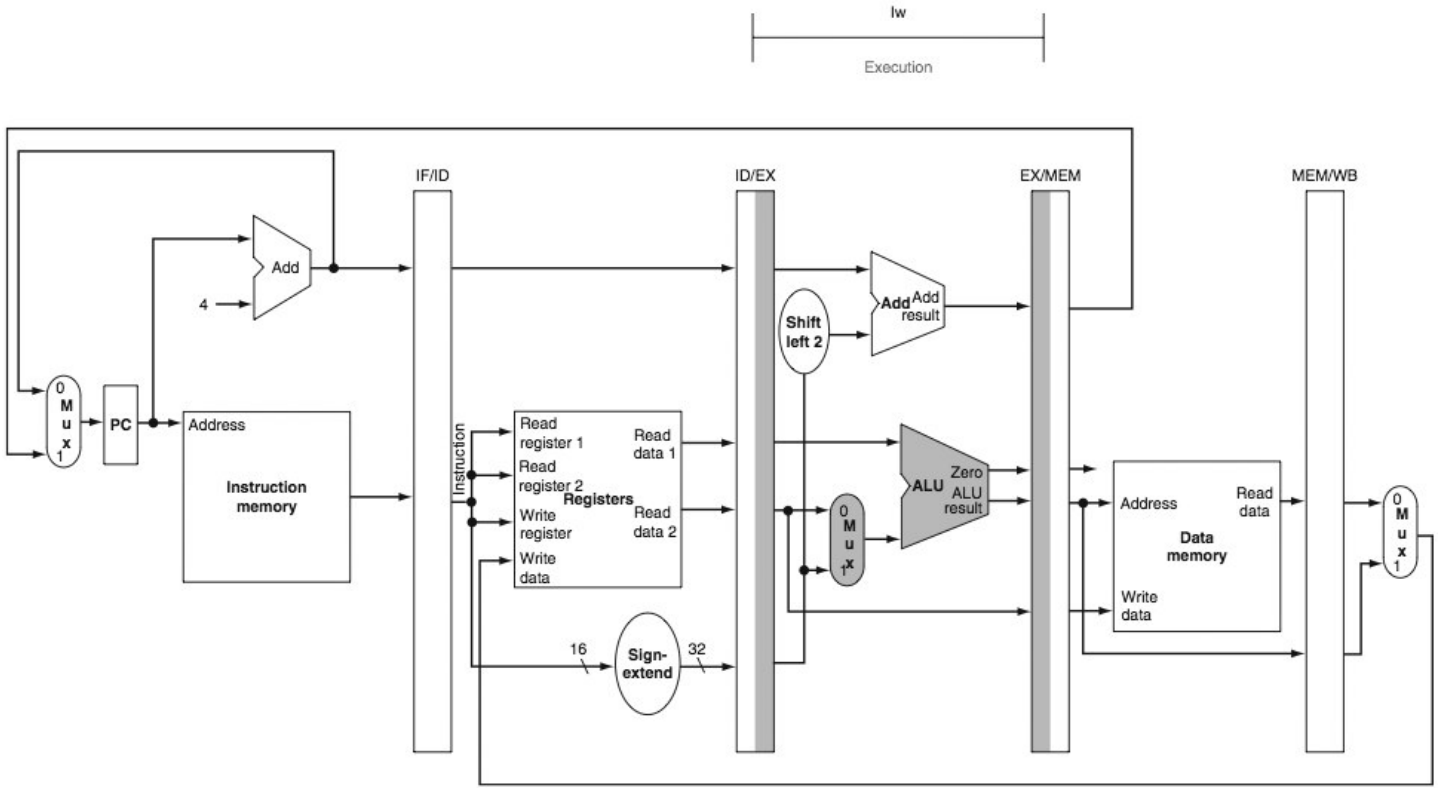


Figure: Execute or Address Calculation

- add register 1 to offset
- sum place in EX/MEM register

Fourth Stage

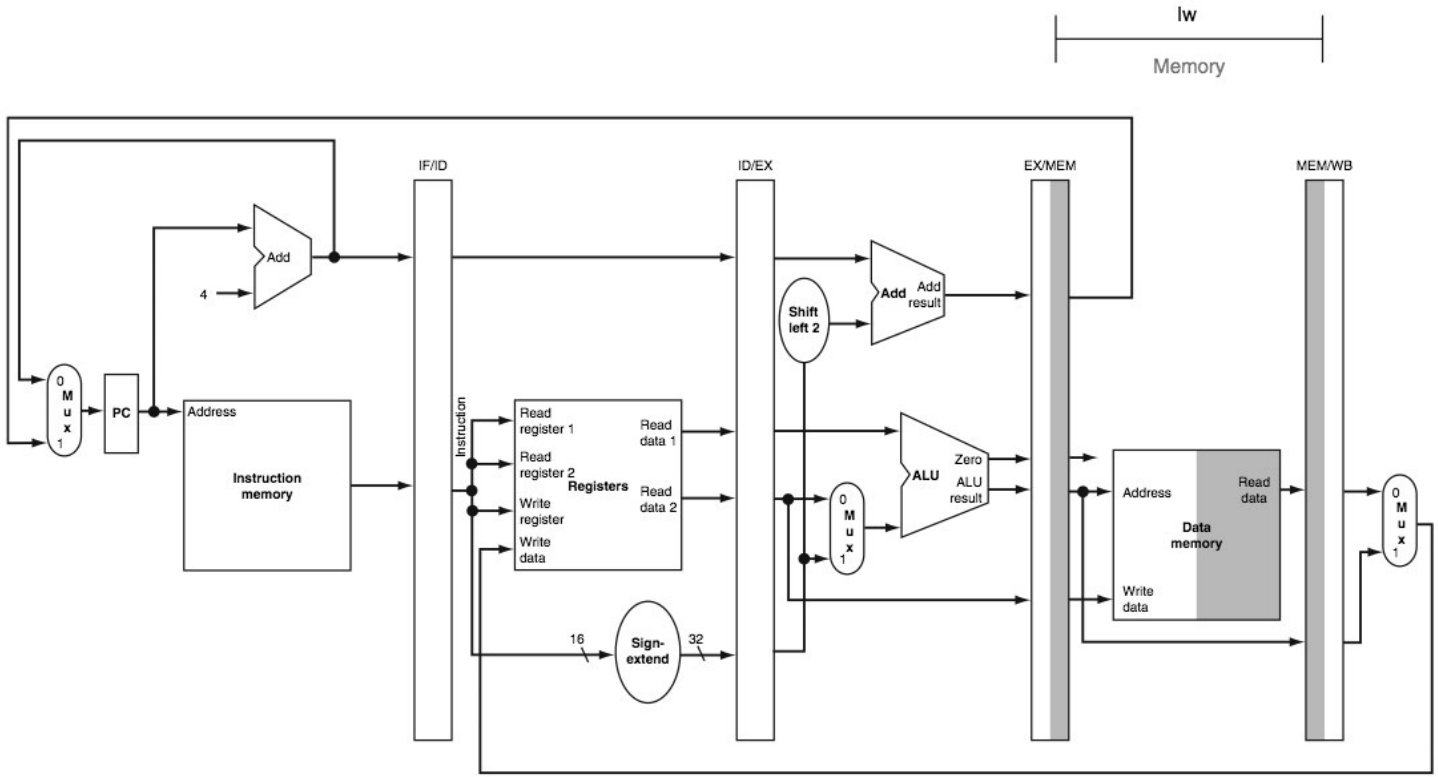


Figure: Memory Access

Fourth Stage

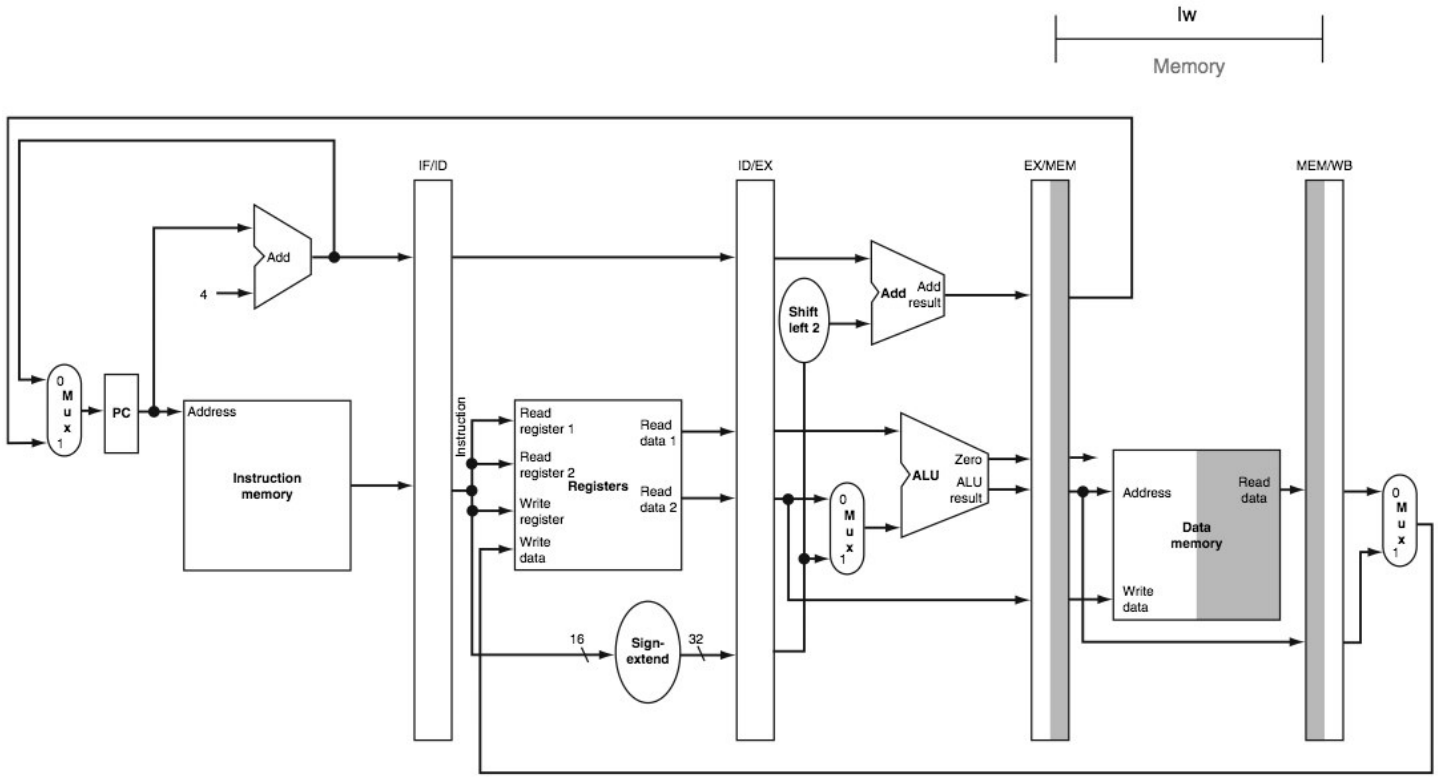


Figure: Memory Access

- load the memory data into MEM/WB register

Fifth Stage

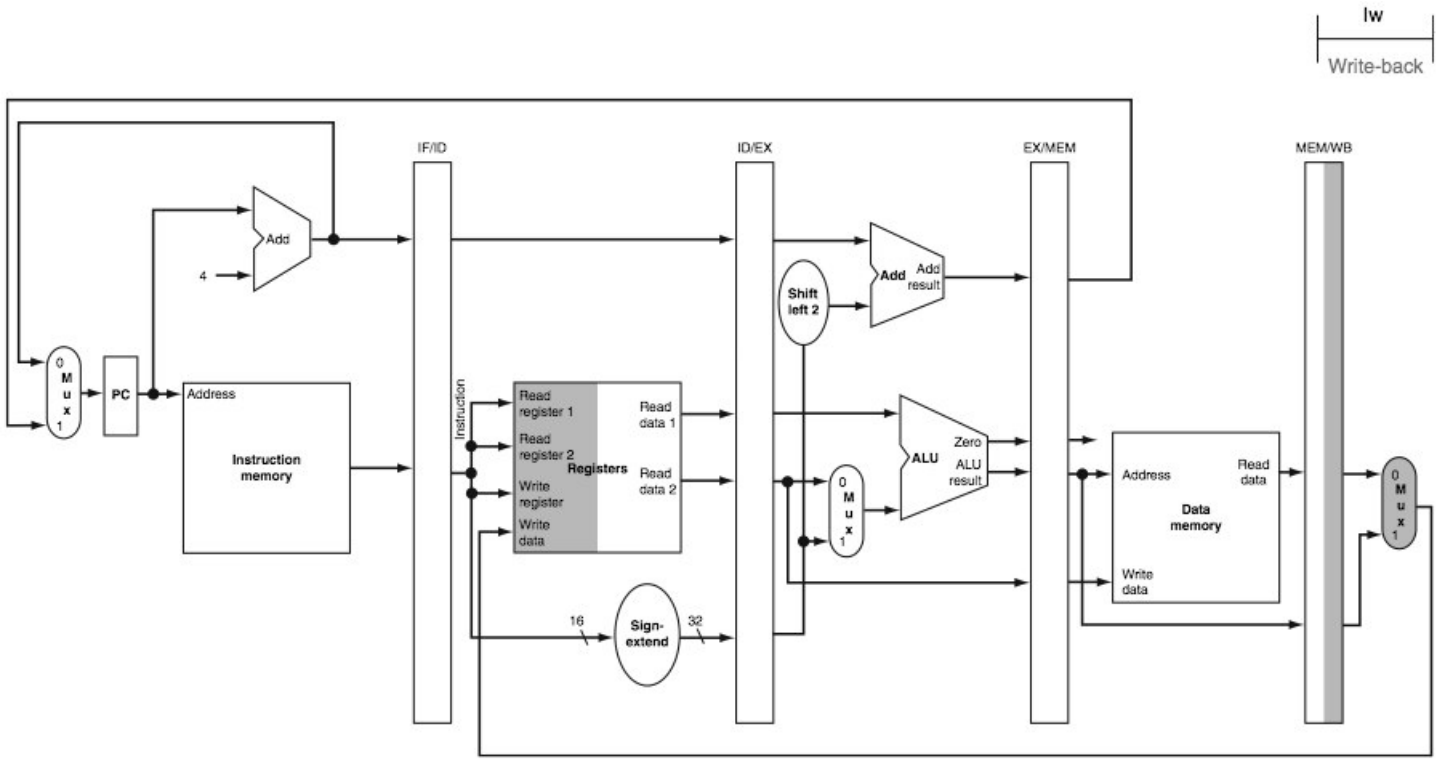


Figure: Write Back

Fifth Stage

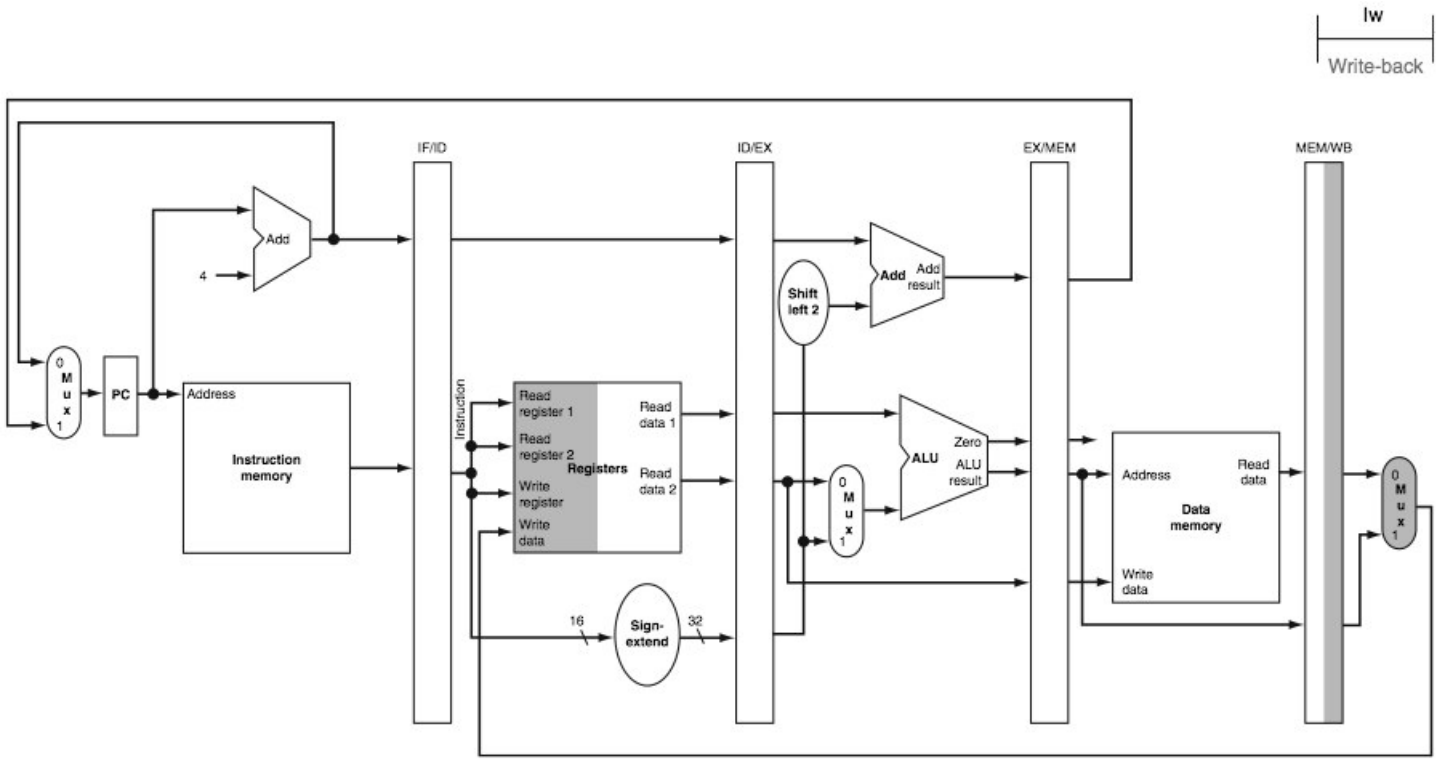


Figure: Write Back

- Read from MEM/WB register into register file
- **Error?**

Fifth Stage

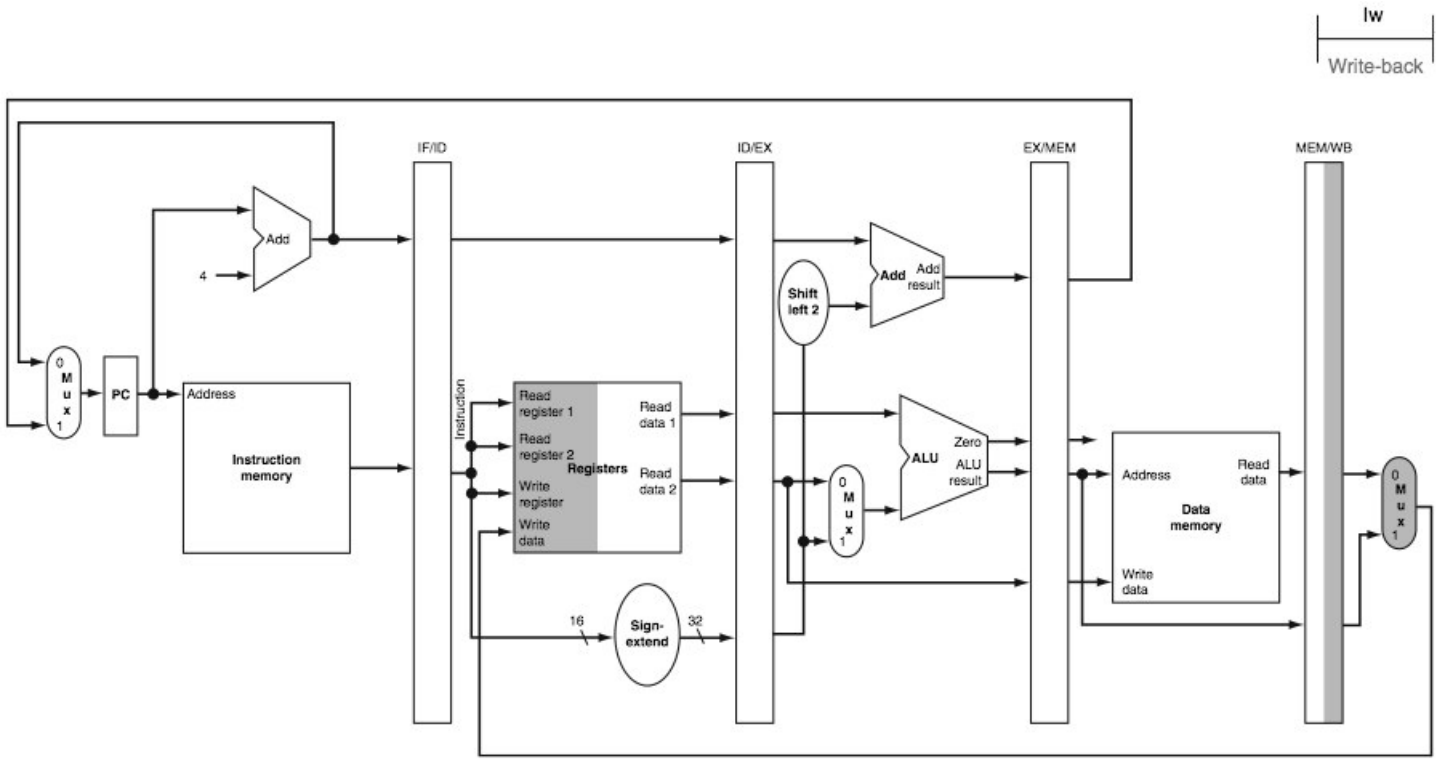


Figure: Write Back

- Read from MEM/WB register into register file
- **Error?**
  - save address for register file is not preserved

Corrected Pipeline Datapath

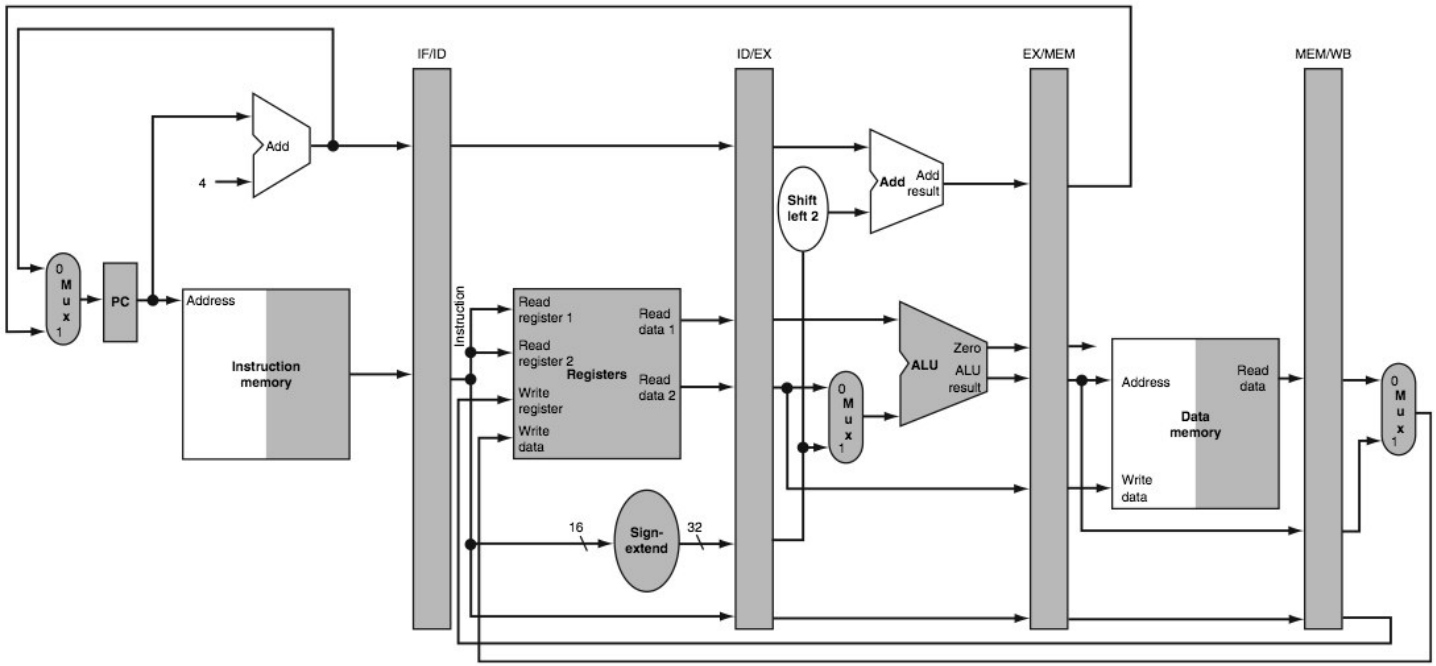


Figure: Correction made for load instruction

Control Lines

- Store and pass Control signals as well

Control Lines

- Store and pass Control signals as well

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

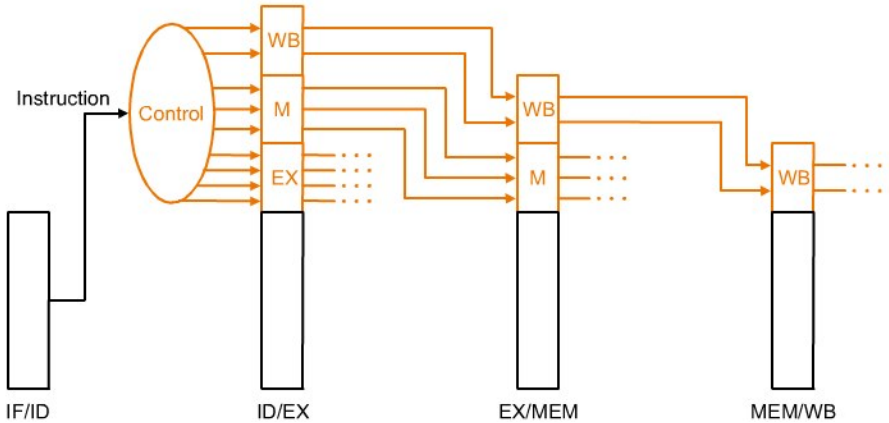


Figure: Including control signals

Complete Pipelined Architecture

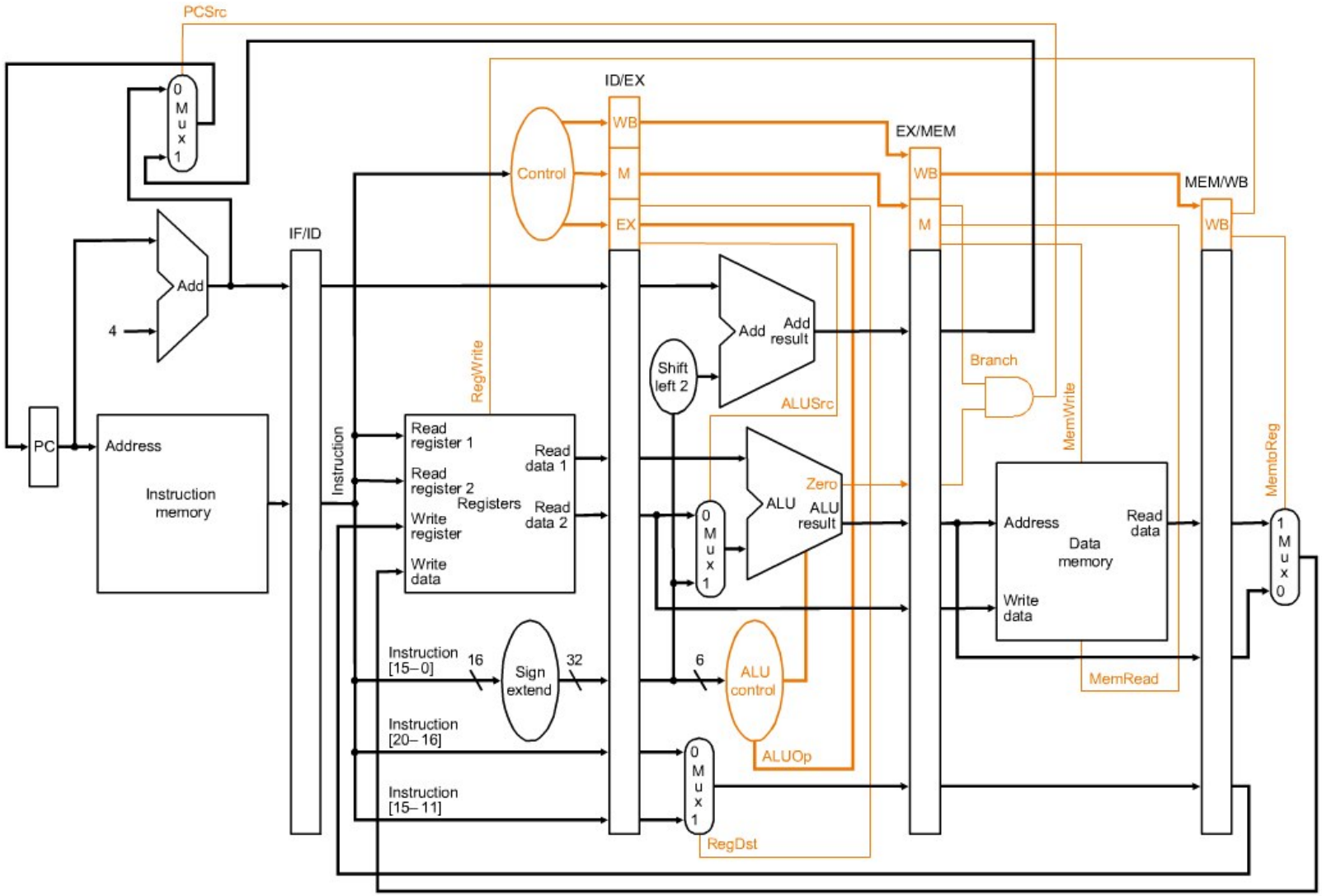


Figure: Datapath and Control of Pipelined MIPS