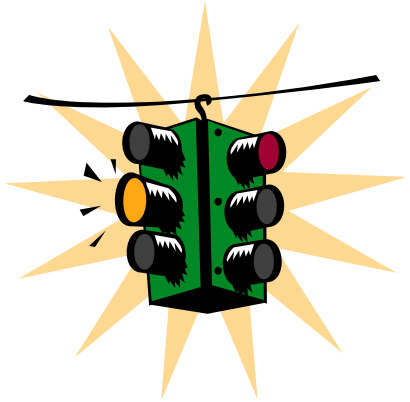
Processor Design

- Data Path
  - ¿ Data Flow path between components
- Control
  - ¿ When and where to open the path

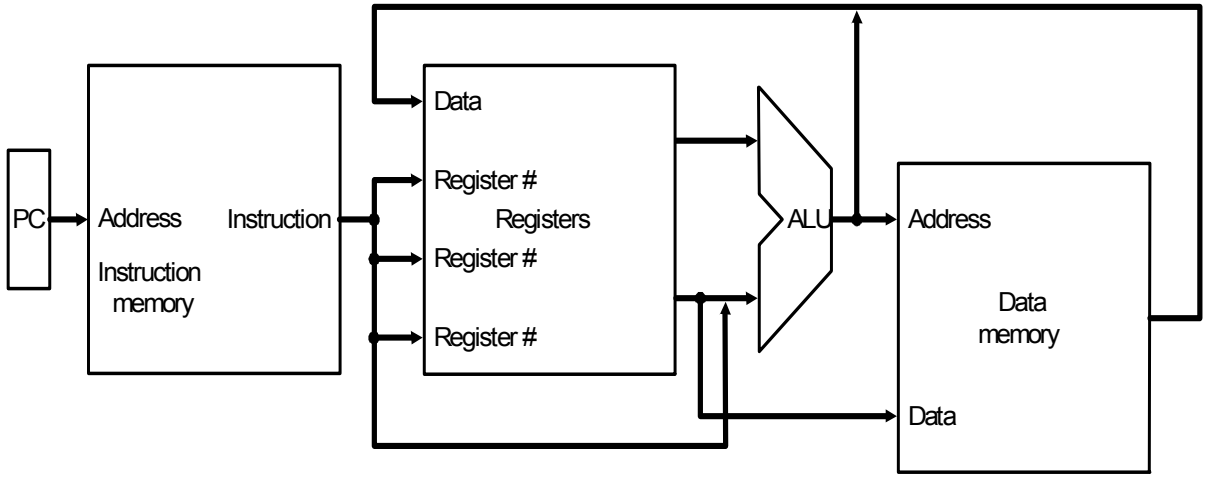


The Processor: Datapath & Control

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - ¿ memory-reference instructions: `lw, sw`
  - ¿ arithmetic-logical instructions: `add, sub, and, or, slt`
  - ¿ control flow instructions: `beq, j`
- Generic Implementation:
  - ¿ use the program counter (PC) to supply instruction address
  - ¿ get the instruction from memory
  - ¿ read registers
  - ¿ use the instruction “op-code” to decide exactly what to do

More Implementation Details

- Simplified View:



Data Path + Control on “Control Points”

- FSM with Two types of functional units:
  - ¿ elements that operate on data values (combinational)
  - ¿ elements that contain state (memory for sequential state transition)

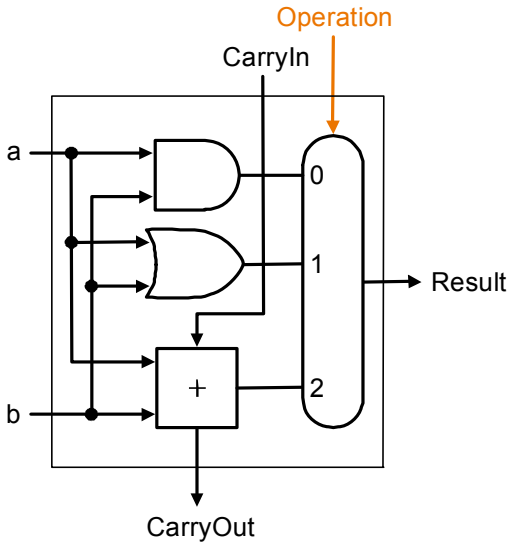
MIPS ALU

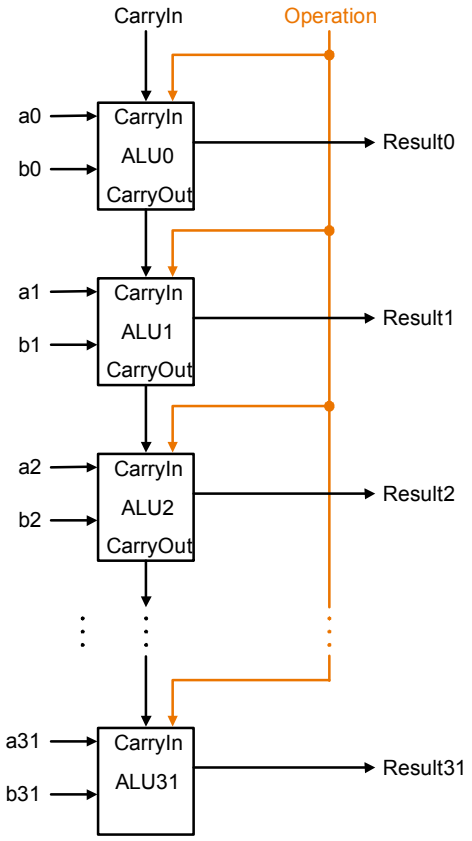
- We can build an ALU to support the MIPS instruction set
  - ¿ key idea: use multiplexor to select the output we want
  - ¿ we can efficiently perform subtraction using two's complement
  - ¿ we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - ¿ all of the gates are always working
  - ¿ the speed of a gate is affected by the number of inputs to the gate
  - ¿ the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - ¿ Clever changes to organization can improve performance (similar to using better algorithms in software)
  - ¿ we'll look at examples for addition and multiplication

Building a 32 bit ALU

Support AND, OR, Add/Sub

Key: Mux





Tailoring the ALU to the MIPS

overview

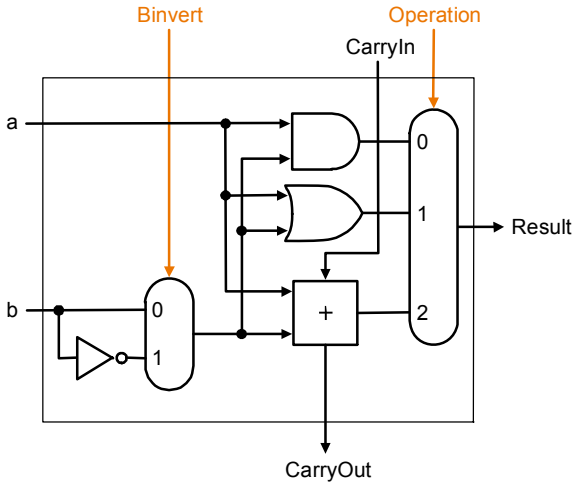
- Adding more and select through Mux per opcode
  - ¿ Need to support test for equality (beq $t5, $t6, 0x100)
    - Ø use subtraction: (a-b) = 0 implies a = b

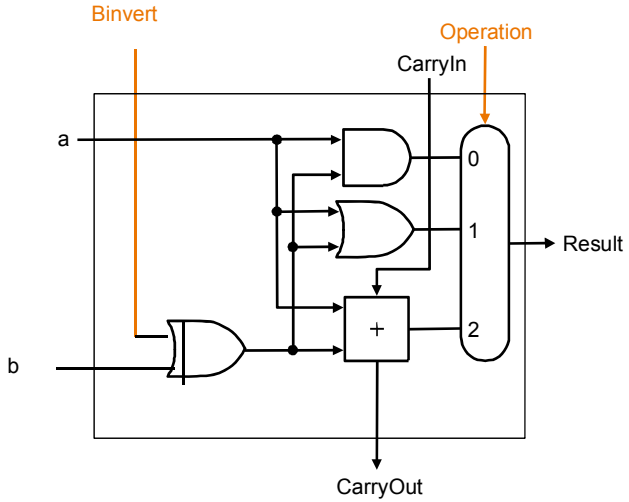
Name	Format	Example						Comments
		6bits	5bits	5 bits	5 bits	5bits	6bits	
add	R	0	2	3	1	0	32	add $1,$2,$3
sub	R	0	2	3	1	0	34	sub $1,$2,$3
addi	I	8	2	1	100			addi $1,$2,100
addu	R	0	2	3	1	0	35	subu $1,$2, $3
and	R	0	2	3	1	0	36	and $1,$2, $3
or	R	0	2	3	1	0	37	or $1, $2, $3
lw	I	35	2	1	100			lw $1,100($2)
sw	I	43	2	1	100			sw $1, 100($2)
beq	I	4	1	2	25			beq $1, $2, 100
j	J	2	2500					j 10000

What about subtraction (a – b) ?

- Two's complement approach: just negate b and add.
- A clever solution: negate+1 = 2's complement

  negate signal = carry-in for least significant bit position





B xor 1 = not B

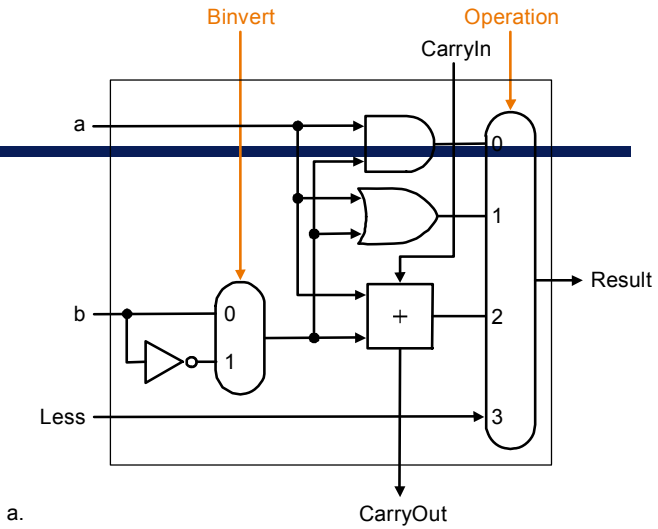
Tailoring the ALU to the MIPS

- Need to support test for equality (beq $t5, $t6, L)
  - ¿ use subtraction: (a-b) = 0 implies a = b
- to support the set-on-less-than instruction

  (slt $r1, $r2, $r3)
  - ¿ remember: slt is an arithmetic instruction
  - ¿ produces a 1 if rs < rt and 0 otherwise
  - ¿ use subtraction: (a-b) < 0 implies a < b

Supporting slt

- Can we figure out the idea?



a.

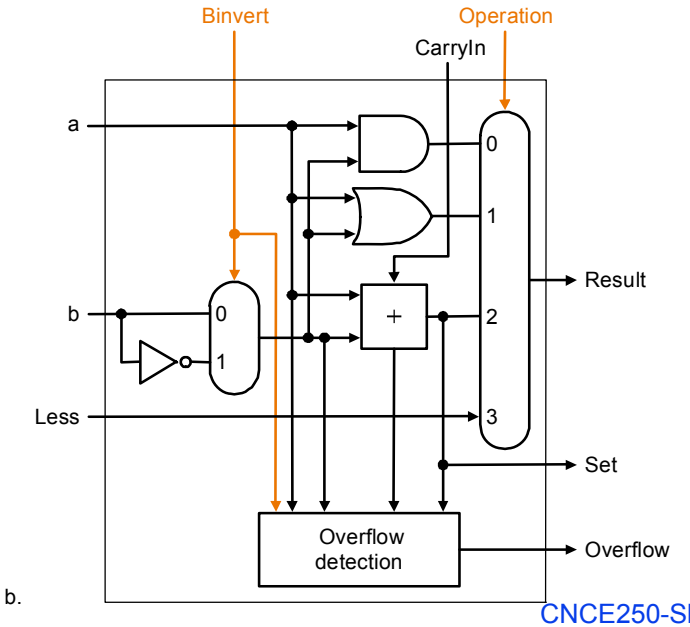
- If a<b then (a-b)<0

So, if after subtraction, the result is Negative, i.e. MSB=1

Then

LSB of Result =MSB

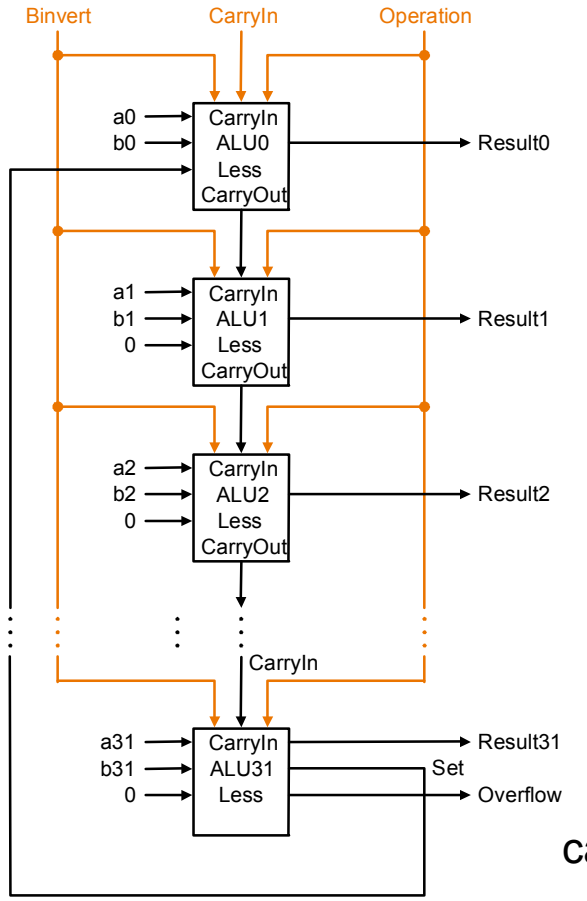
and other bits of Result = 0



b.

Supporting slt & overflow

for sub for 2's compl. to select a particular op



carry-in ≠ carry-out

Sign-bit, MSB goes to LSB for slt

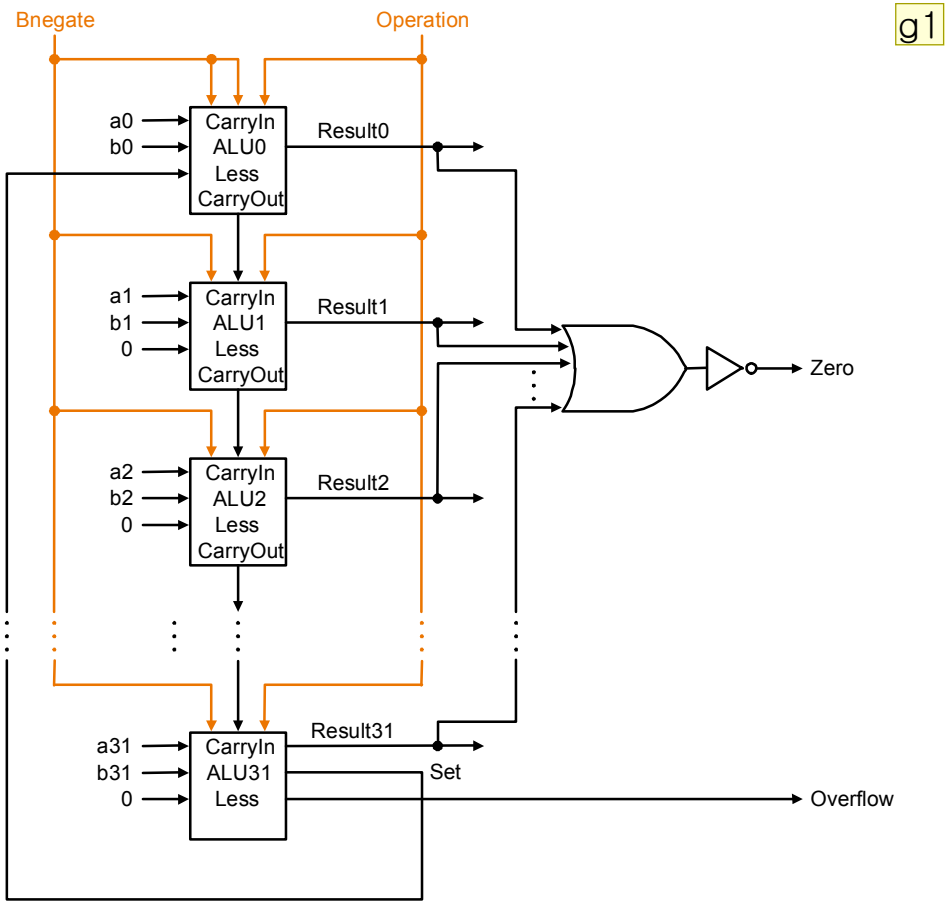
Test for equality

- ***Note: zero is 1 when the result is zero!***

- Notice control lines:

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
```

(see Sec. 4.4. p.247)



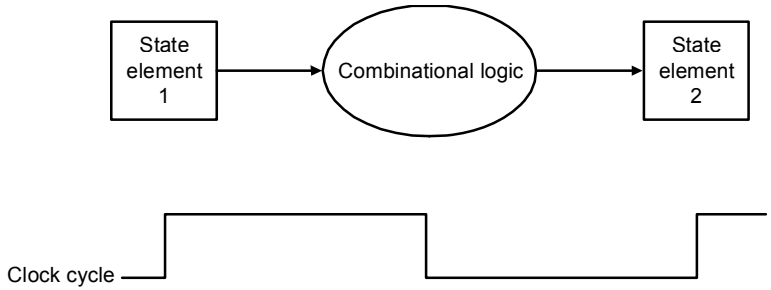
g1

for beq, subtraction yields 0. all bit positions will be o. so negating them makes 1.
for sub and slt we need subtraction, so Bnegate will be 1 and it will feed to the lsb as a carry-in to have two's comlement.
ghlee, 2015-03-31

Our Implementation

review

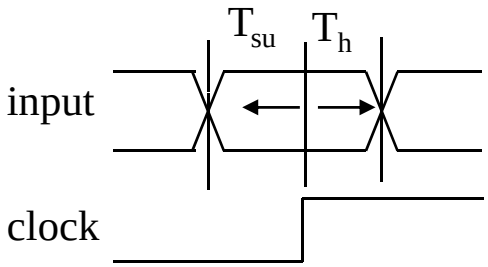
- An edge triggered methodology
- Typical execution:
  - ¿ read contents of some state elements (registers),
  - ¿ send values through some combinational logic,
  - ¿ write results to one or more state elements (registers)



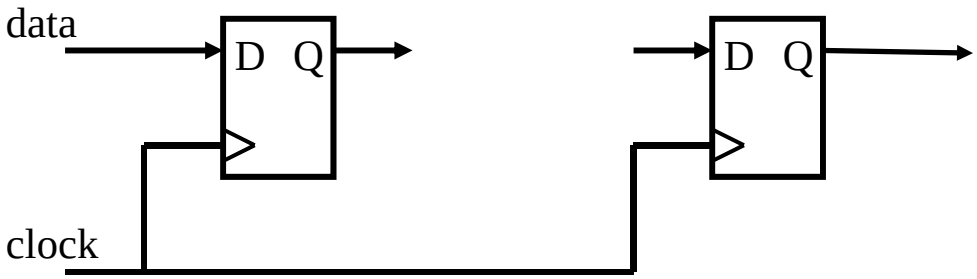
Timing and Control Point

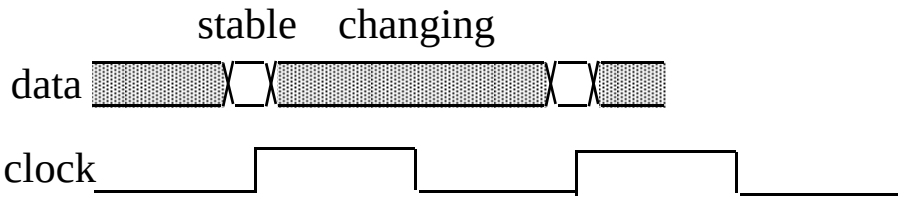
review

- Definition of terms
  - clock: periodic event, causes state of memory element to change; can be rising or falling edge, or high or low level
  - setup time: minimum time before the clocking event by which the input must be stable (Tsu)
  - hold time: minimum time after the clocking event until which the input must remain stable (Th)



there is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized

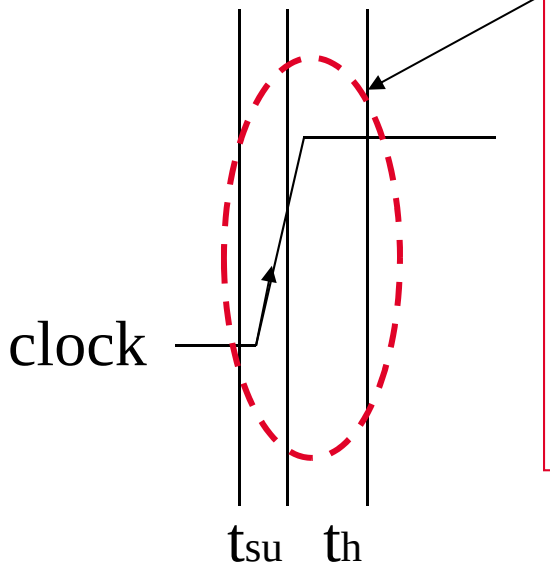




Typical Timing Specifications

review

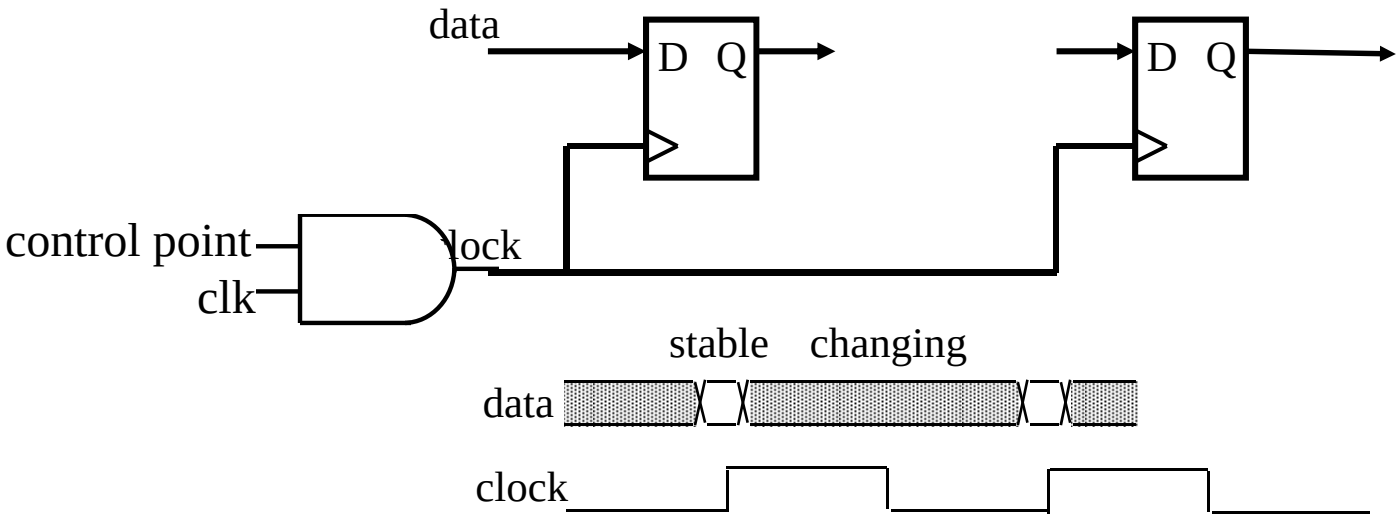
- Positive edge-triggered
  - ¿ Setup and hold times
  - ¿ Minimum clock width
  - ¿ Propagation delays:
  - ¿ output available after input sampled (low to high, high to low, max and typical)



Input sampling window:
Within this window,
signal should reach the input
and becomes stable;
If not,
the circuit become unstable, i.e. out-of-synch.
-Metastability: Neither 0 nor 1
-Practically difficult to happen but not impossible

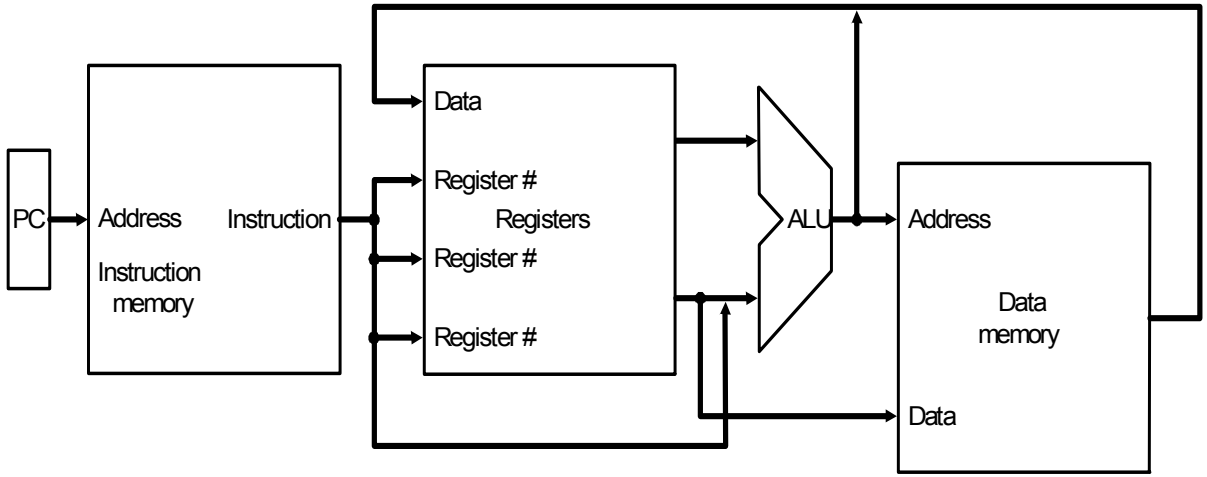
Control Point

review



Implementation to do

- Simplified View:

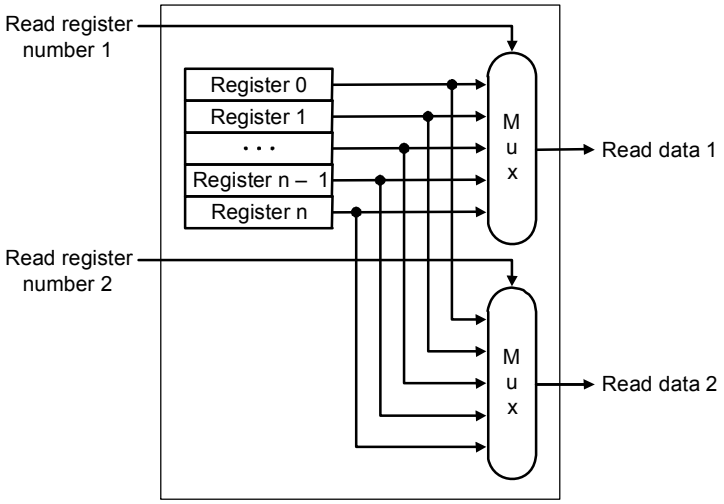


Data Path + Control on “Control Points”

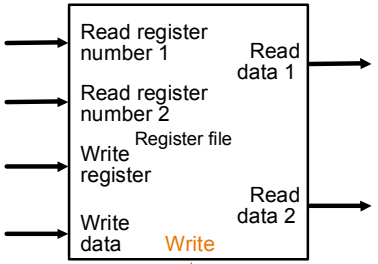
- Simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or`
  - control flow instructions: `beq, j`

Register File

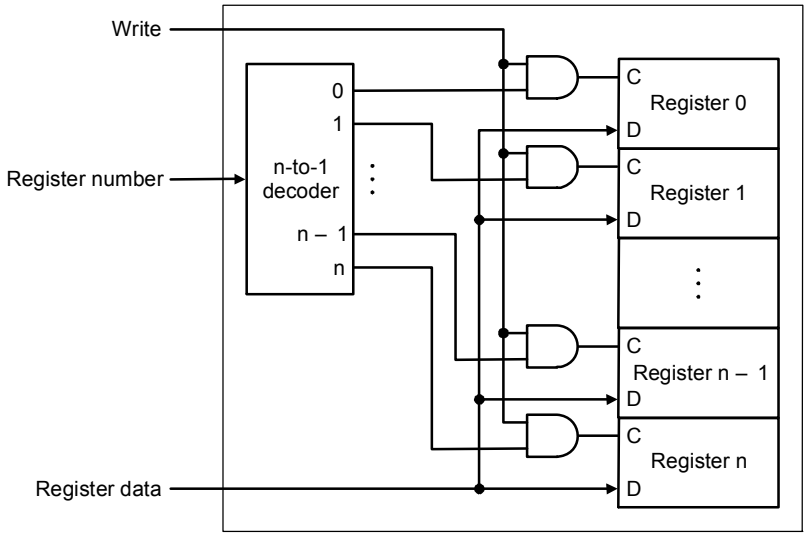
- Built using D flip-flops



2 read ports



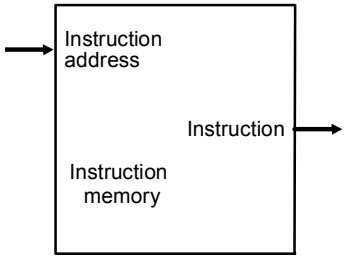
1 write port



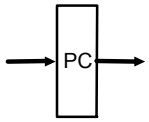
Note:
we still use the real clock
to determine when to write

Simple Implementation

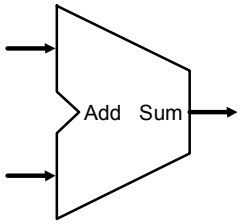
- Include the functional units we need for each instruction



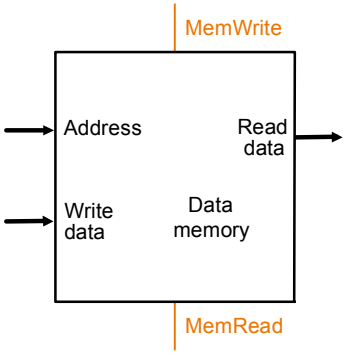
a. Instruction memory



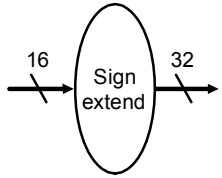
b. Program counter



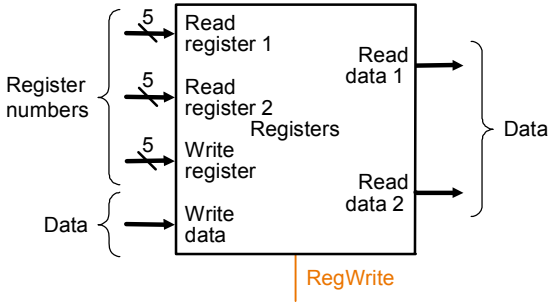
c. Adder



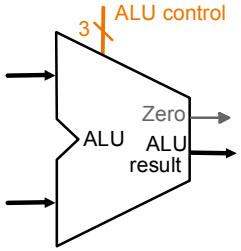
a. Data memory unit



b. Sign-extension unit



a. Registers



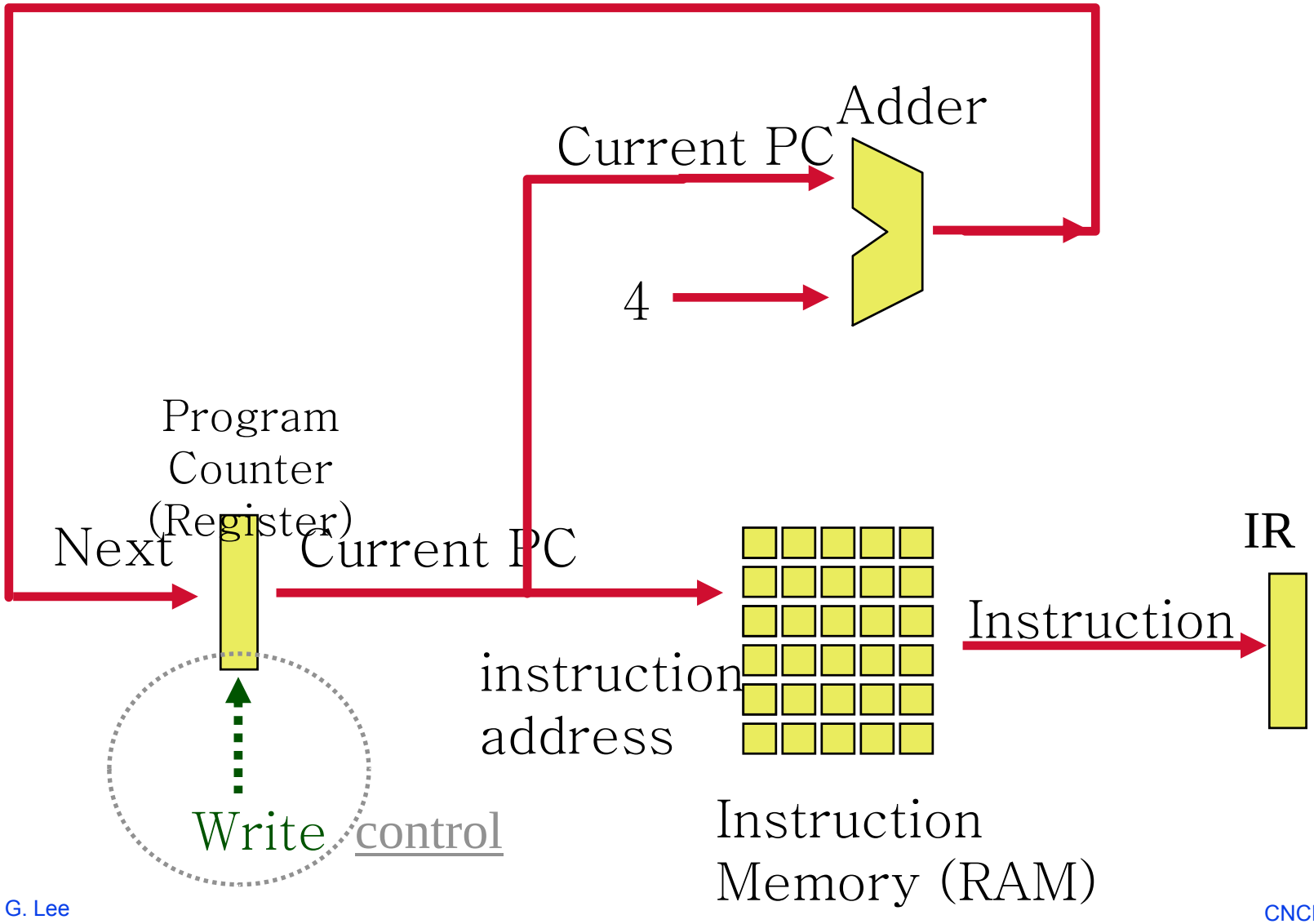
b. ALU

Why do we need this stuff?

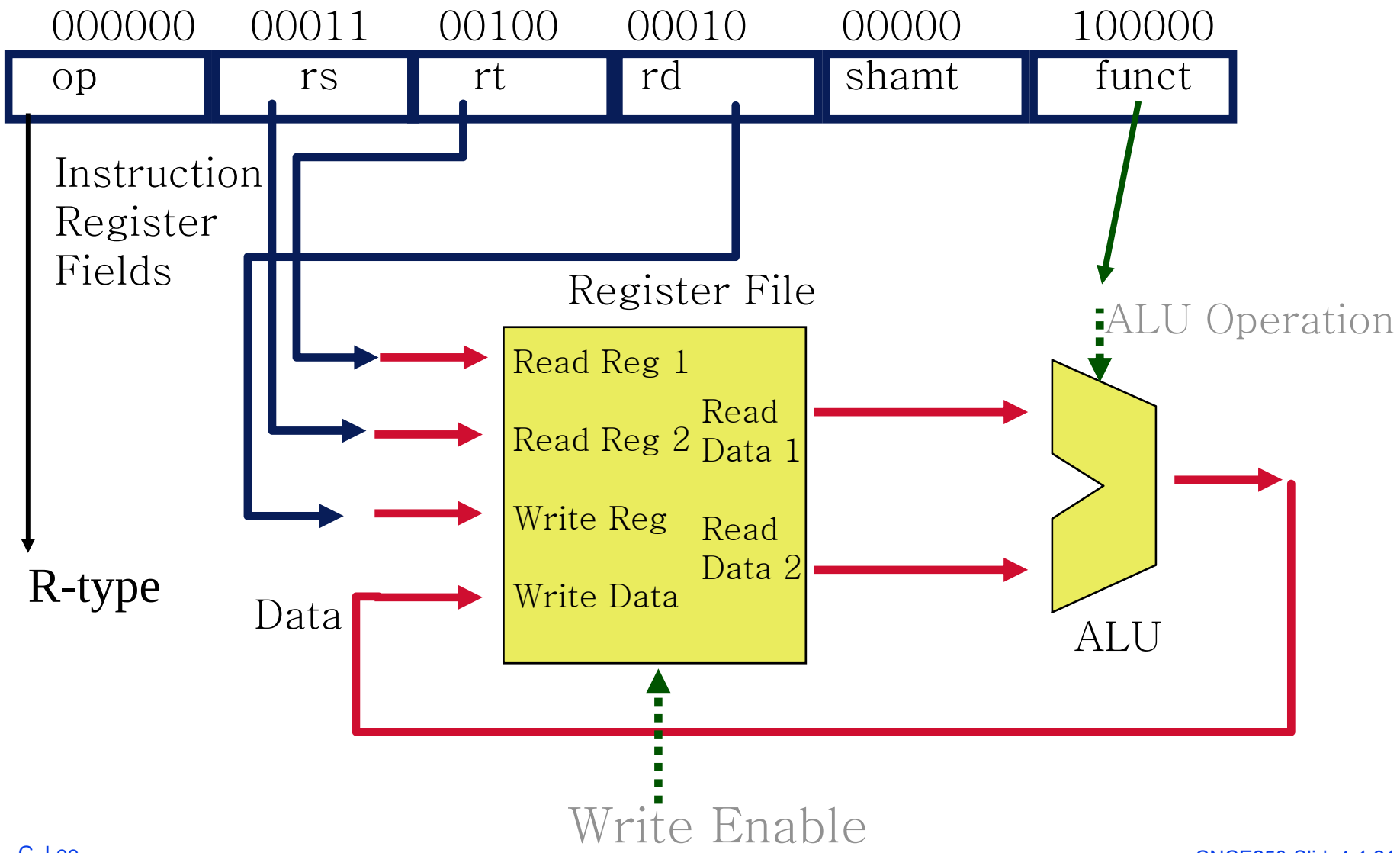
Typical Instruction Execution

1. **IF: Fetch instruction from memory and increment the PC**
2. **ID: Decode instruction ; OF: Access Register File**
3. **EX: If necessary, perform an ALU operation**
4. **WB: Write results back to register file**

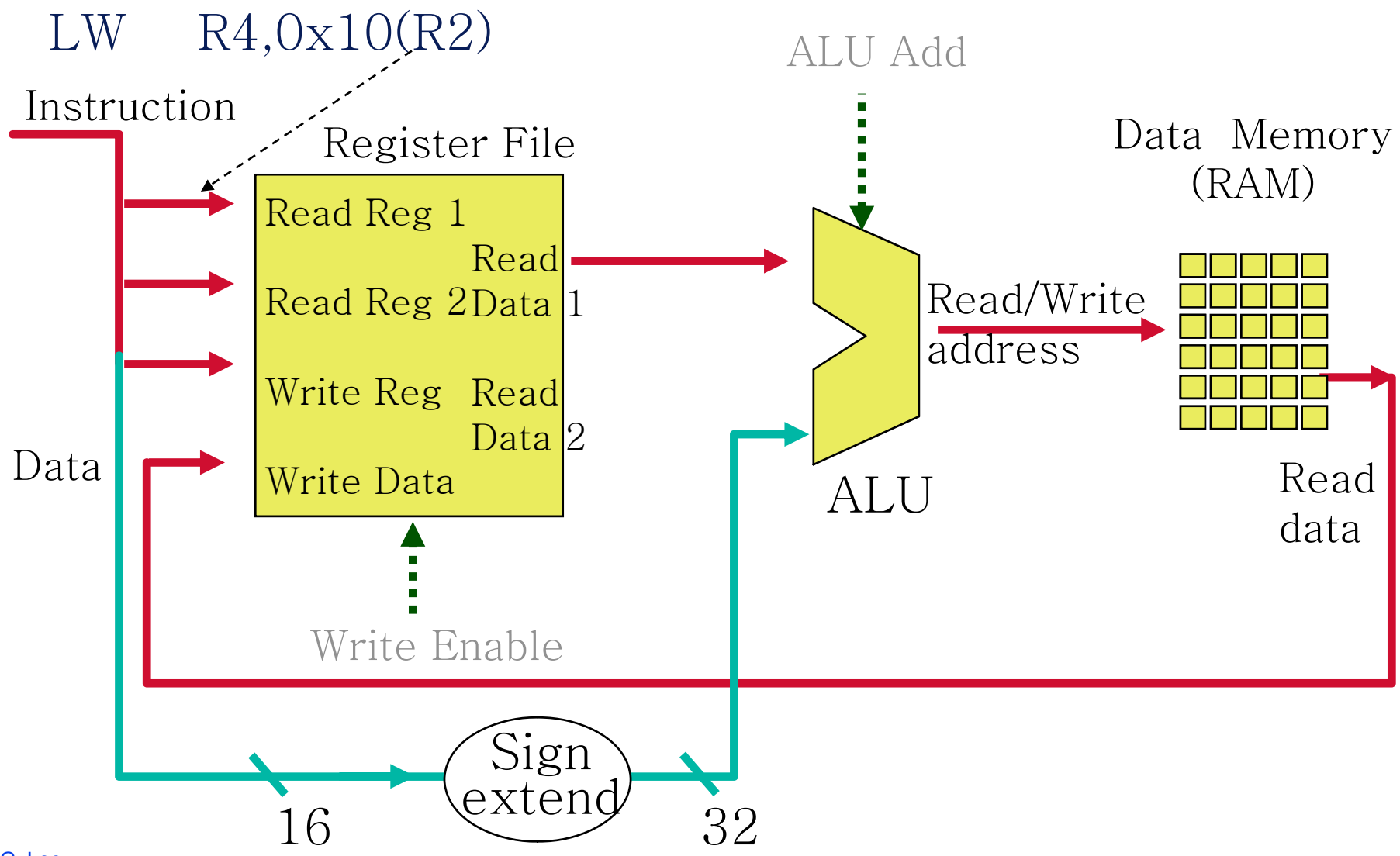
Instruction Fetch Unit



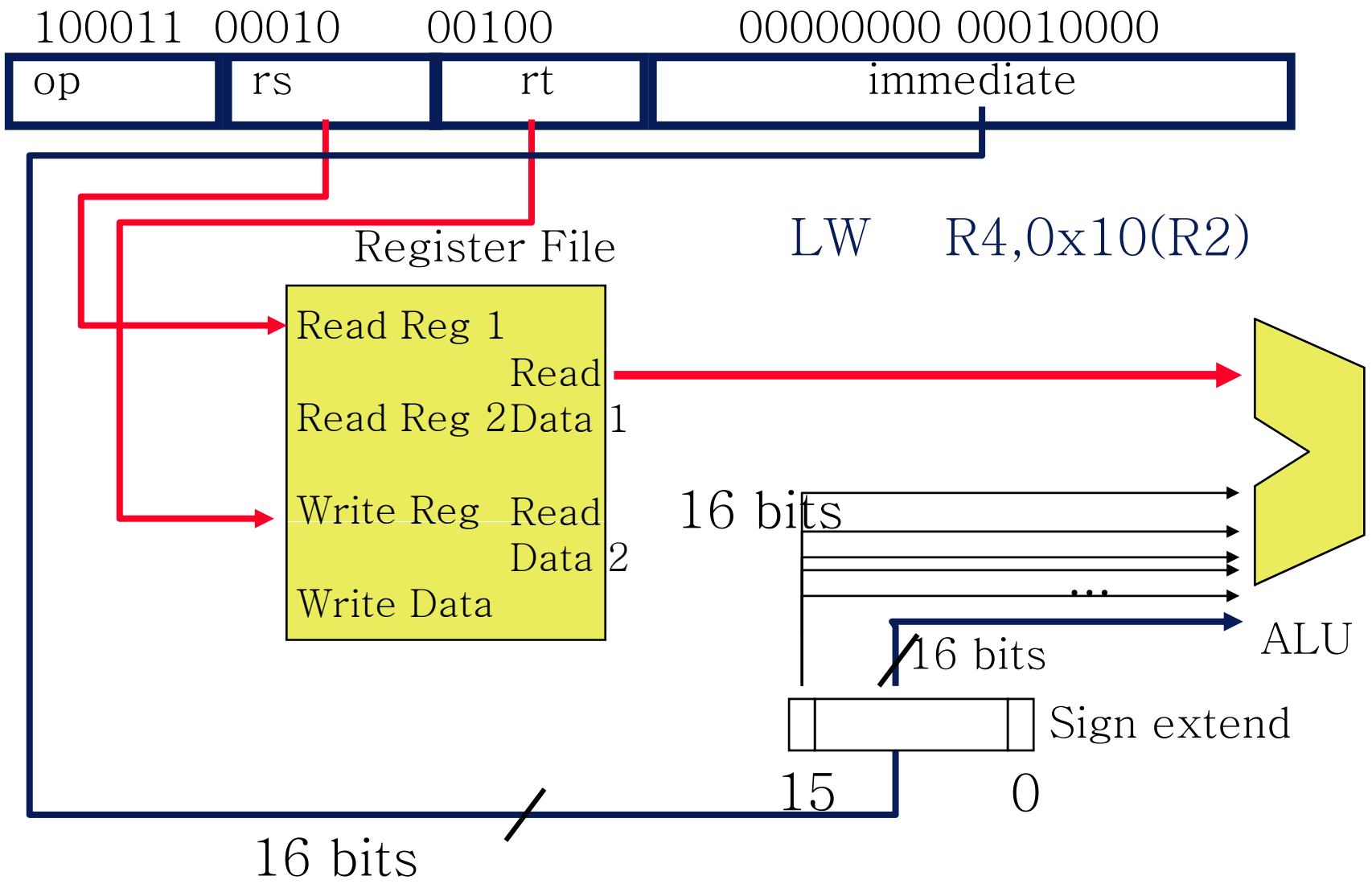
ID & EX: R-Type e.g. ADD R2, R3, R4



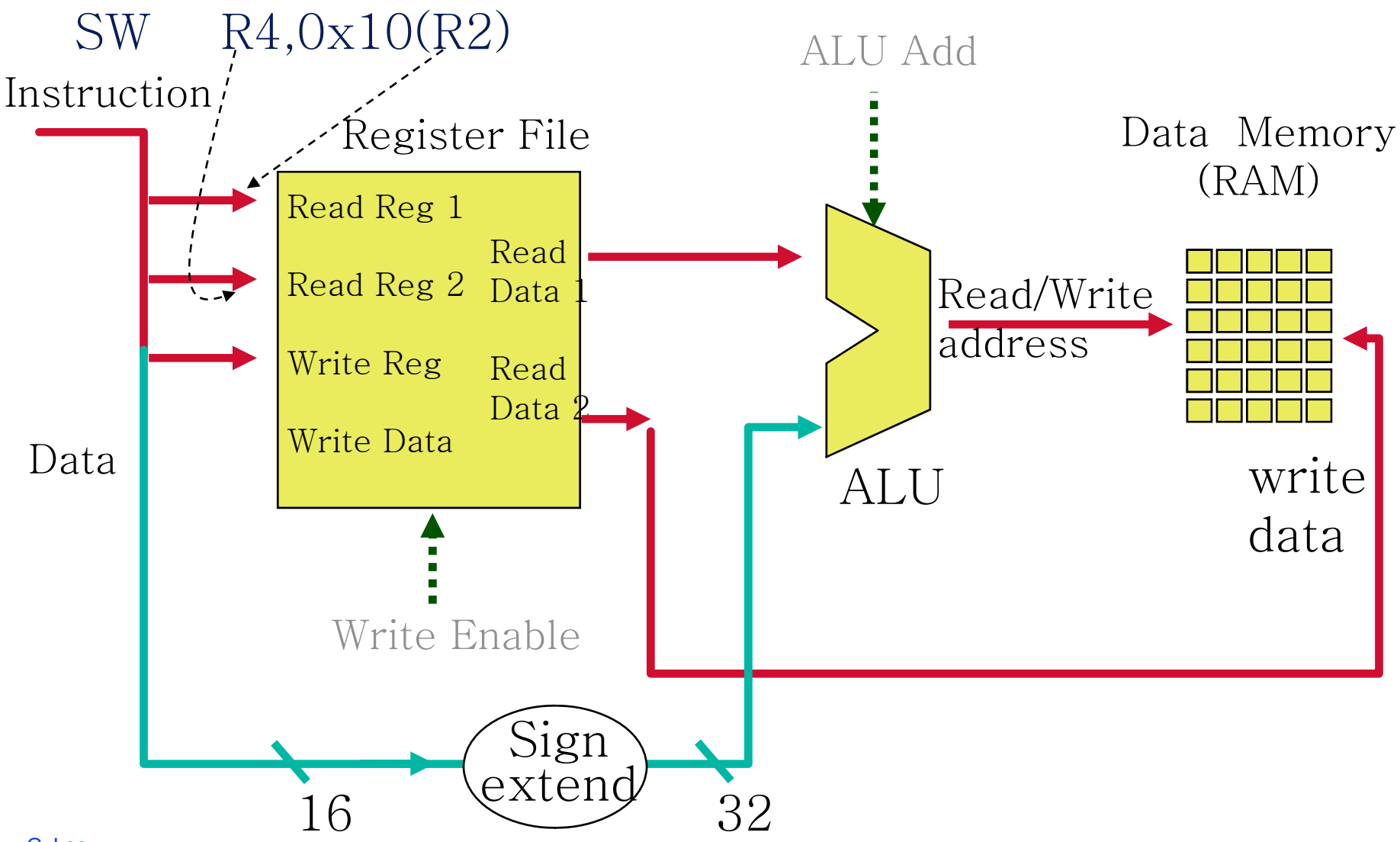
ID&EX: e.g. loads and stores



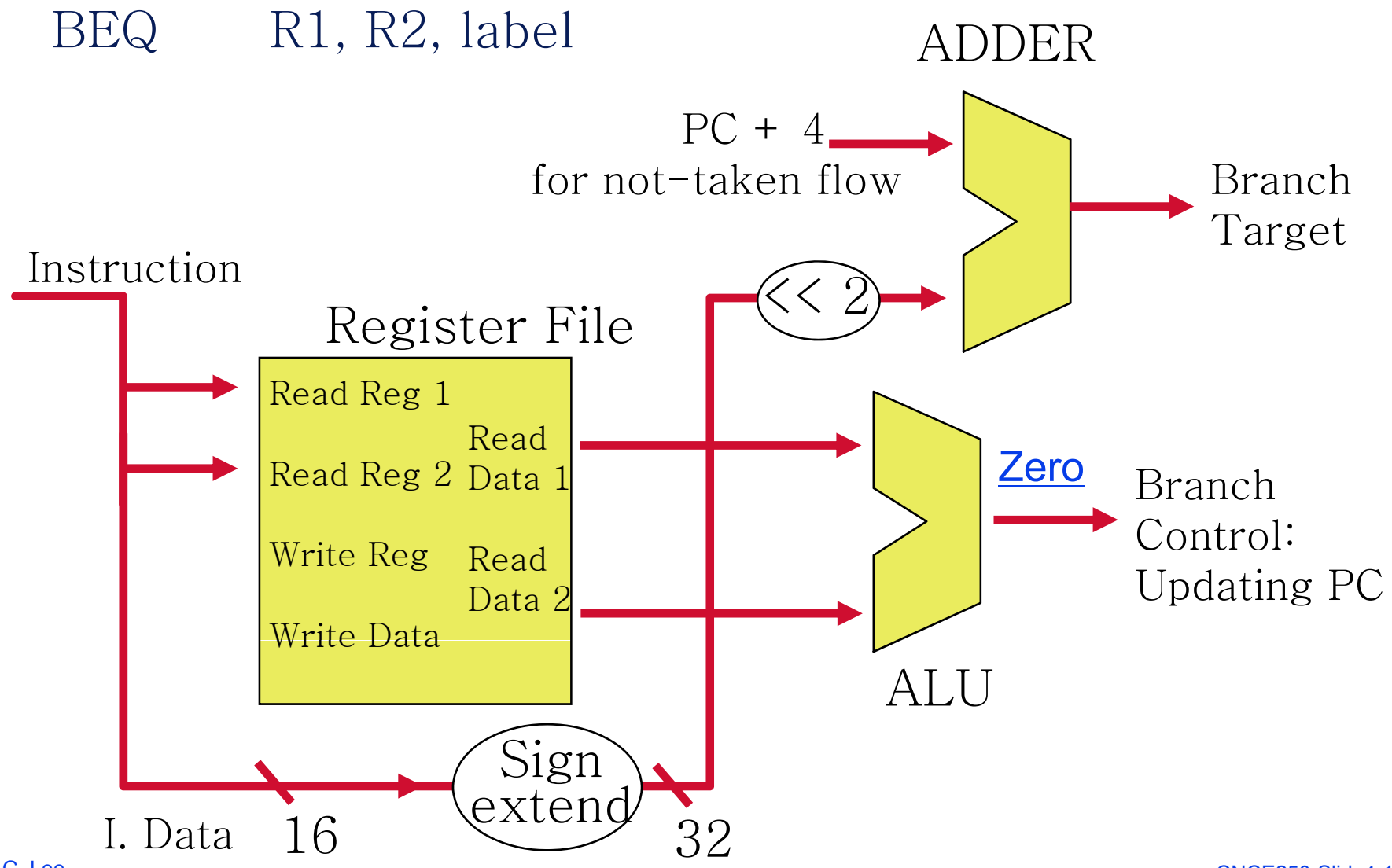
Sign Extender



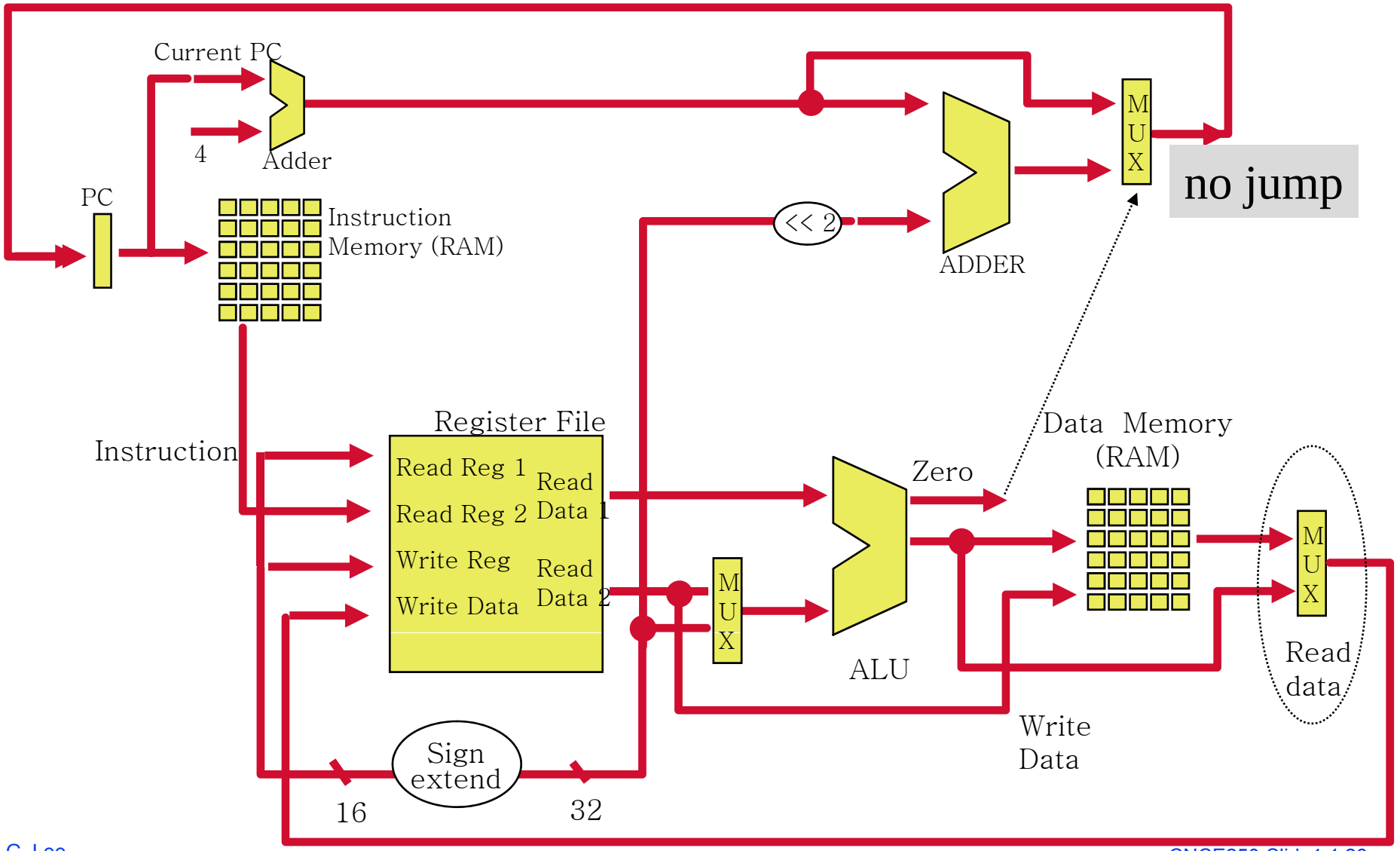
Store: reverse flow



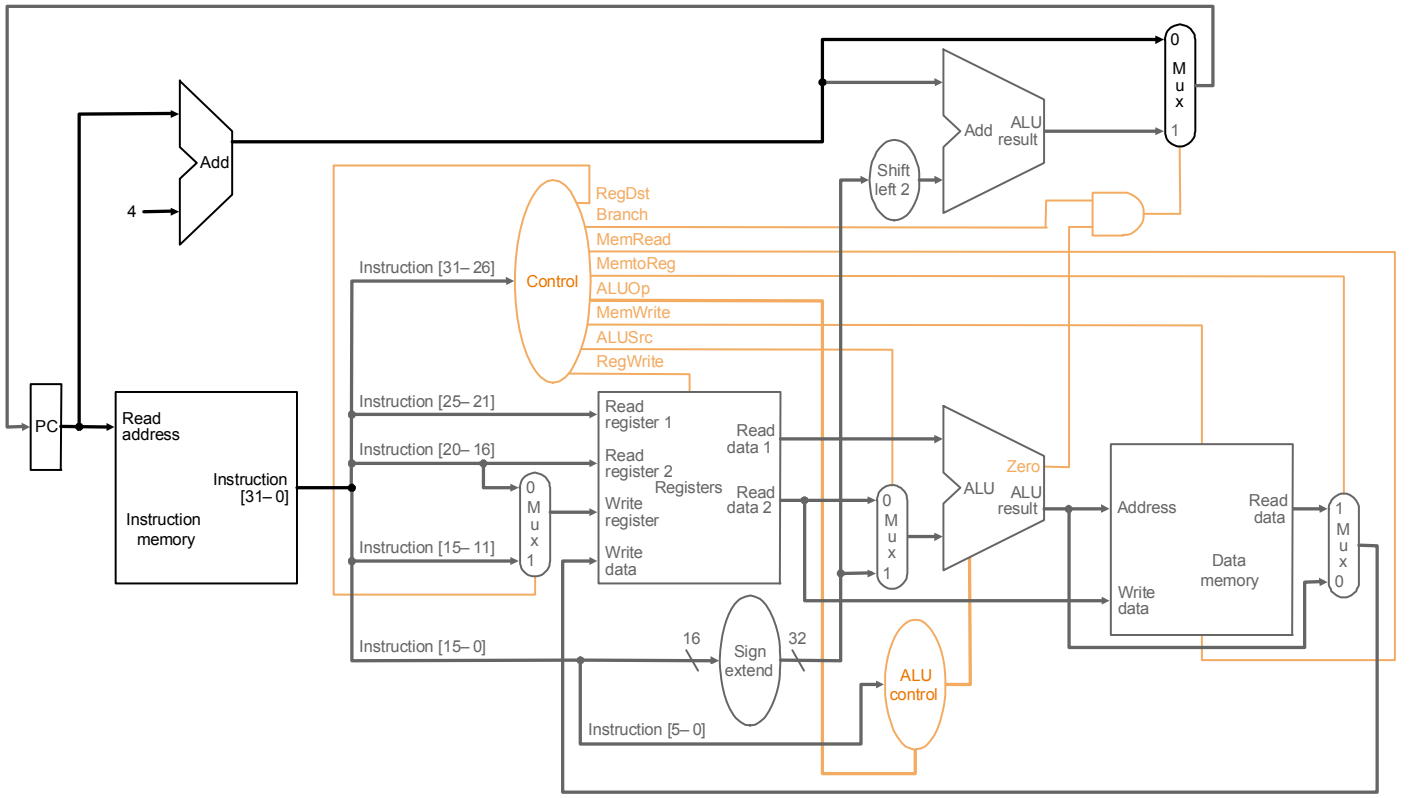
ID & EX: Branch Need Adder to Compute Branch Target



Put Together: Complete Single-cycle Datapath



Control



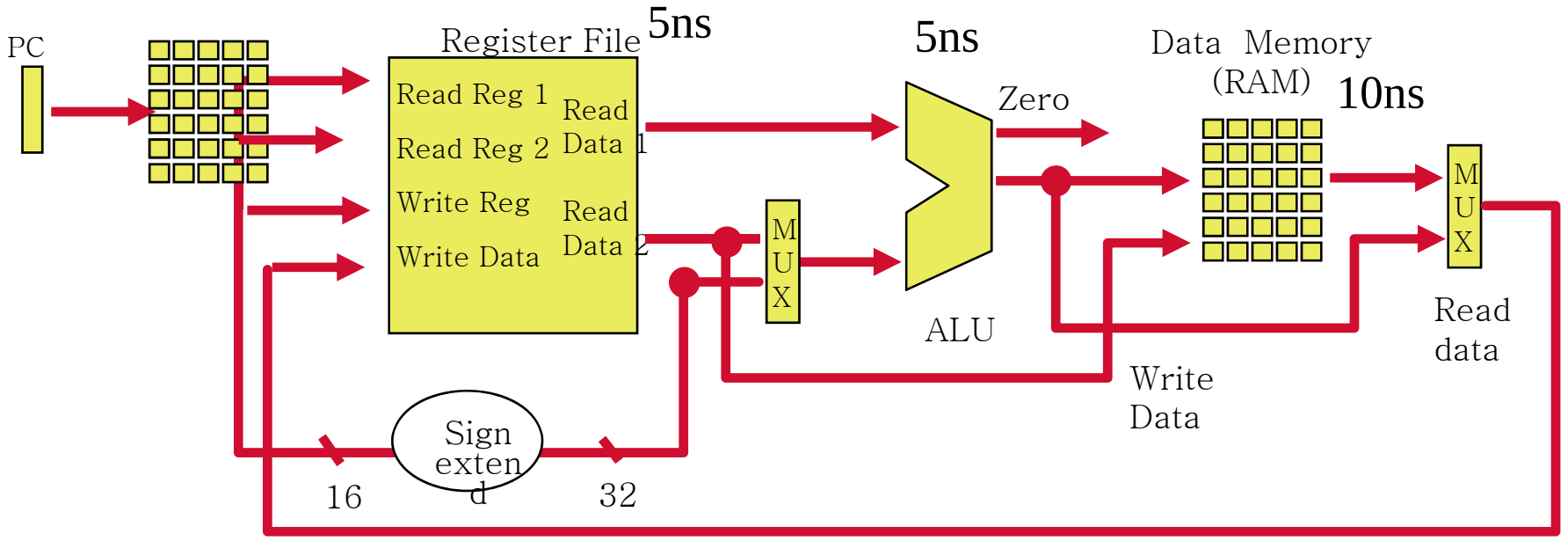
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

What's Wrong with a Single-Cycle Implementation

- Difficult to implement variable cycle clock
- Usually run the clock at the SLOWEST speed
  - ¿ This is called the critical path
  - ¿ The critical path is the path through the system which limits performance
- What if we add a floating point unit?
  - ¿ FP (floating point) math can take a very long time
- How about breaking the machine into parts
  - ¿ Per instruction type

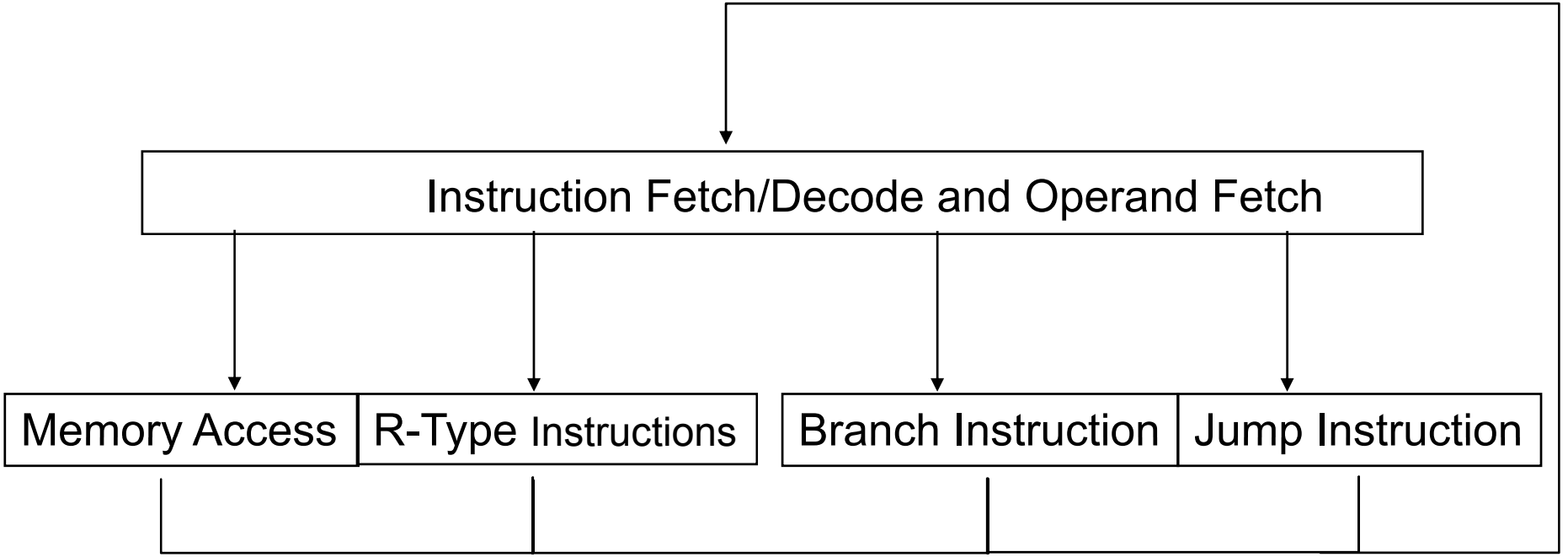
Instruction Timings

Instr Type	InstrMem	Reg Read	ALU	DataMem	Reg Write	Total
R-format	10	5	10	-	5	30 ns
Load	10	5	10	10	5	40 ns
Store	10	5	10	10	-	35 ns
Branch	10	5	10	-	-	25 ns
Jump	10	-	-	-	-	10 ns



Finite State Machine Control

Cleaner Summary



Execution Steps

- IF: Instruction Fetch

```
IR = Memory[PC];
PC = PC + 4;
```

- ID and OF: Instruction Decode and Operand Fetch

```
A = Reg[IR[25..21]];
B = Reg[IR[20..16]];
ALUOut = PC + (signExtend(IR[15..0])  << 2);
```

Note: always read registers and add PC regardless of instruction type

Execution Steps

- EX: Execution, or
memory address computation or branch completion

  - ¿ Memory Reference

    ALUOut = A + (signExtend(IR[15..0]); Effective Address (EA)

  - ¿ Arithmetic/Logical Operation

    ALUOut = A + B

  - ¿ Branch

    PC = ALUOut if zero; (A == B)

  - ¿ Jump

    PC = PC[31 ..28] || (IR[25..0) << 2);

Execution Steps

- WB: write result back for Memory access or R-type instruction completion
  - ¿ Memory Reference:

    MDR = Memory[ALUOut] or Memory[ALUOut] = B;

  - ¿ Arithmetic/Logical Instructions (R-type)

    Reg[IR[15..11]] = ALUOut;

- Load instruction completion
  - ¿ Reg[IR[20..16]] = MDR

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

More on control

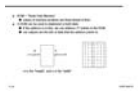
- One may add more instructions and features like
  - ¿ Jal – jump and link for subroutine call
    - ∅ Save PC (in stack and link register) and update PC with target
    - ∅ Return with saved PC
  - ¿ Exception/Interrupt handling
    - ∅ Save PC (to EPC) and update PC with predetermined location for exception
    - ∅ Interrupt vector table or cause register to specify proper handling
    - ∅ Restart with EPC

Finite State Machine for Control

- Implementation:

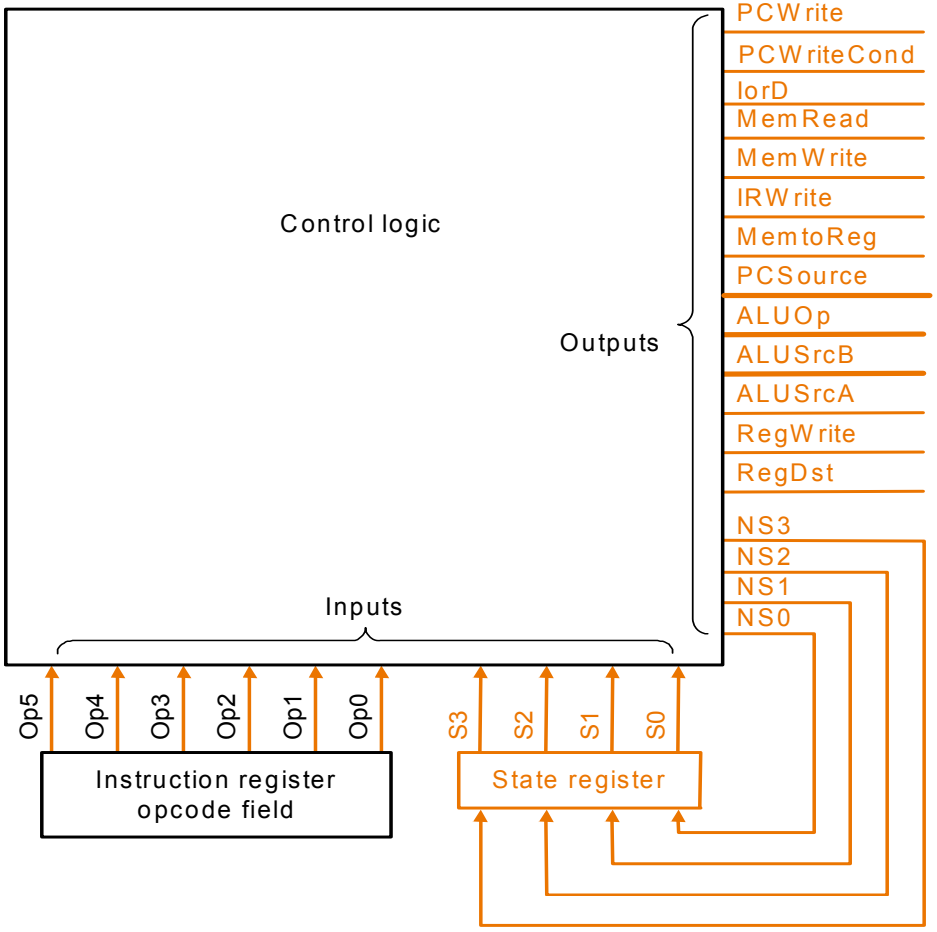
•Building logic circuit

  •Hardwired control

•Storing Table 

•Writing Program

  •microprogramming



Summary

- Per opcode, select “source” and “destination” in sequence of IF; ID/OF; EX; WB for proper signal flows
- Selection in sequence, i.e. control logic, can be implemented in various forms as a FSM in hardwired one or micro-program or mix of those.