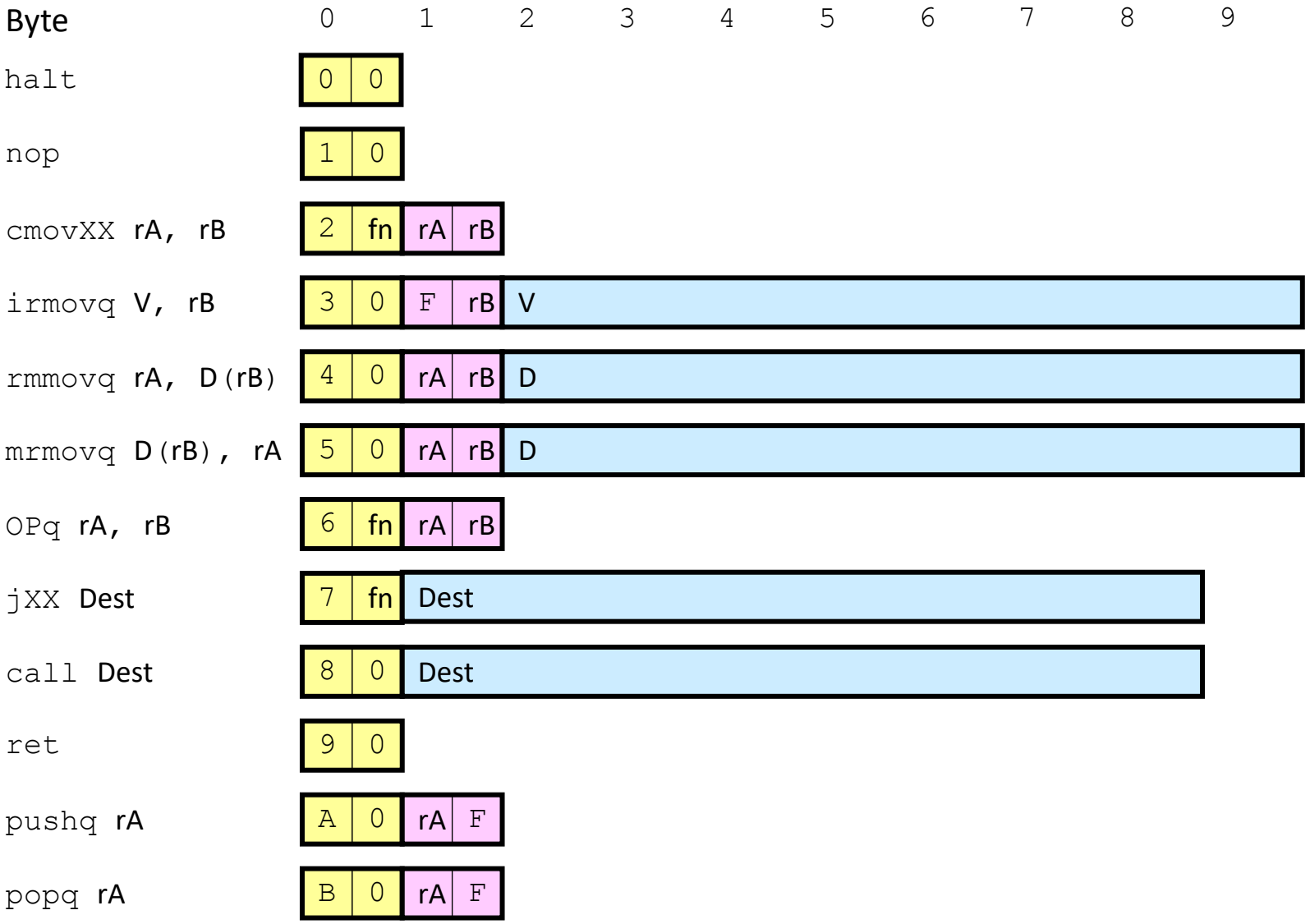
Sequential Implementation

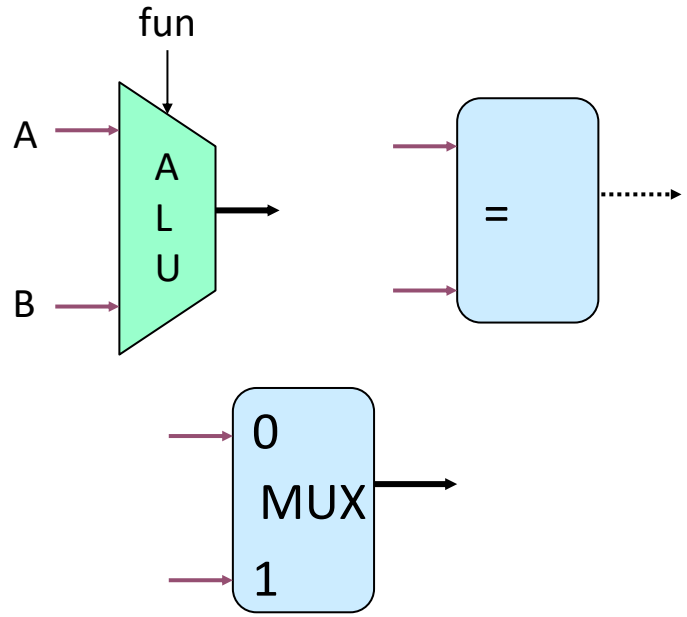
CMPU 224 – Computer Organization

Jason Waterman

Y86-64 Instruction Set

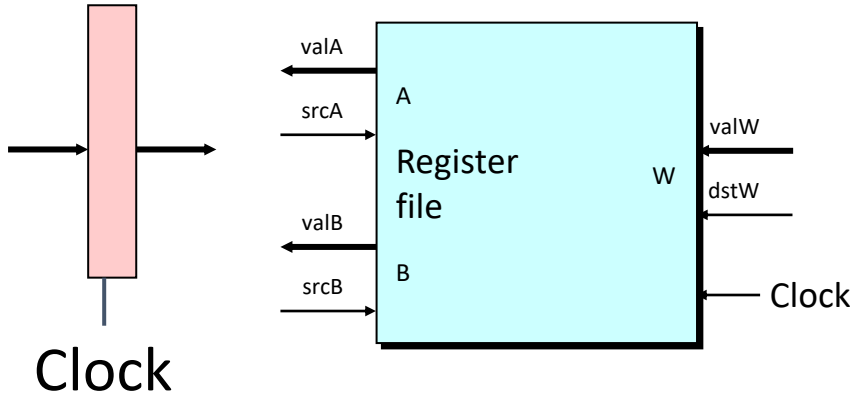


Building Blocks



- Combinational Logic
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control

- Storage Elements
  - Store bits
  - Registers
  - Addressable memories
  - Loaded only as clock rises



Hardware Control Language

- Very simple hardware description language
  - Can only express limited aspects of hardware operation
    - Parts we want to explore and modify
  - Boolean operations have syntax similar to C logical operations
  - We'll use it to describe control logic for processors
- Data Types
  - `bool`: Boolean
    - `a`, `b`, `c`, …
  - `int`: words
    - `A`, `B`, `C`, …
    - Does not specify word size---bytes, 64-bit words, …
- Statements
  - `bool a = bool-expr ;`
  - `int A = int-expr ;`

HCL Operations

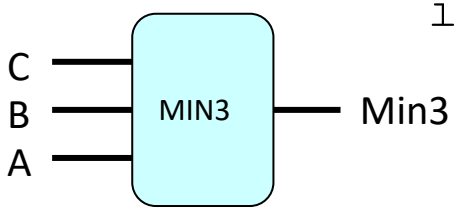
- Classify by type of value returned

- Boolean Expressions – evaluate to a Boolean
  - Logic Operations
    - `a && b, a || b, !a`
  - Word Comparisons
    - `A == B, A != B, A < B, A <= B, A >= B, A > B`
  - Set Membership
    - `A in { B, C, D }`
      - Same as `A == B || A == C || A == D`
- Word Expressions
  - Case expressions
    - `[ a : A; b : B; c : C ]`
    - Evaluate test expressions `a`, `b`, `c`, … in sequence
    - Return word expression `A`, `B`, `C`, … for first successful test

HCL Word-Level Examples

- **Find minimum of three input words**
- **HCL case expression**
- **Final case guarantees match**

Minimum of 3 Words

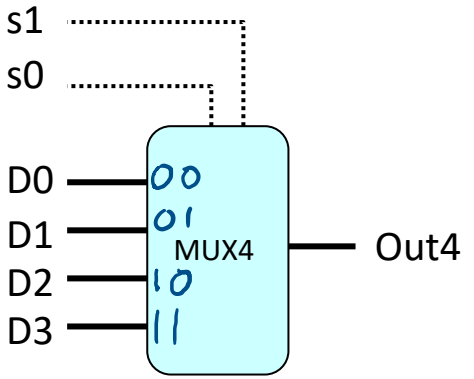



```
int Min3 = [
  A < B && A < C : A;
  B < C          : B;
  1              : C;
];
```

- **Select one of 4 inputs based on two control bits**
- **HCL case expression**
- **Simplify tests by assuming sequential matching**

S_1 S_0	D
00	D_0
01	D_1
10	D_2
11	D_3

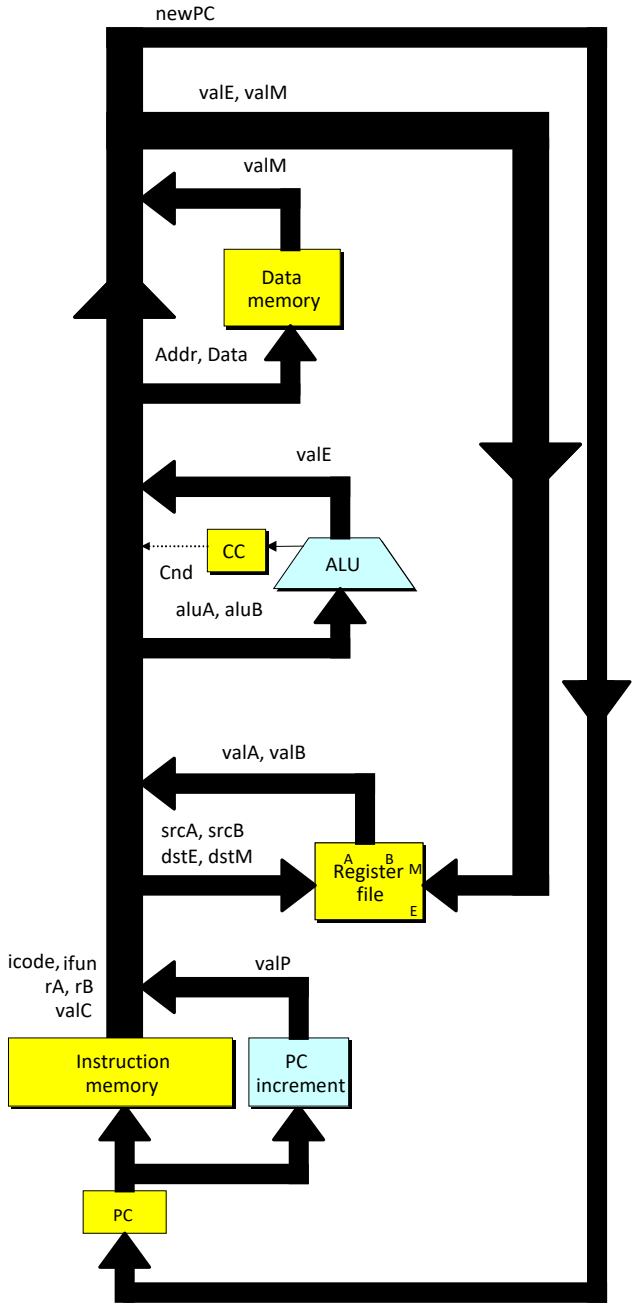
4-Way Multiplexor




```
int Out4 = [
  !s1&&!s0: D0;
  !s1     : D1;
  !s0     : D2;
  1       : D3;
];
```

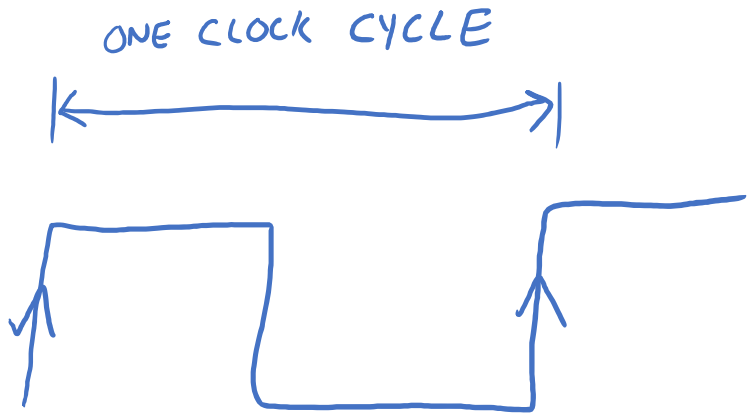
SEQ Hardware Structure

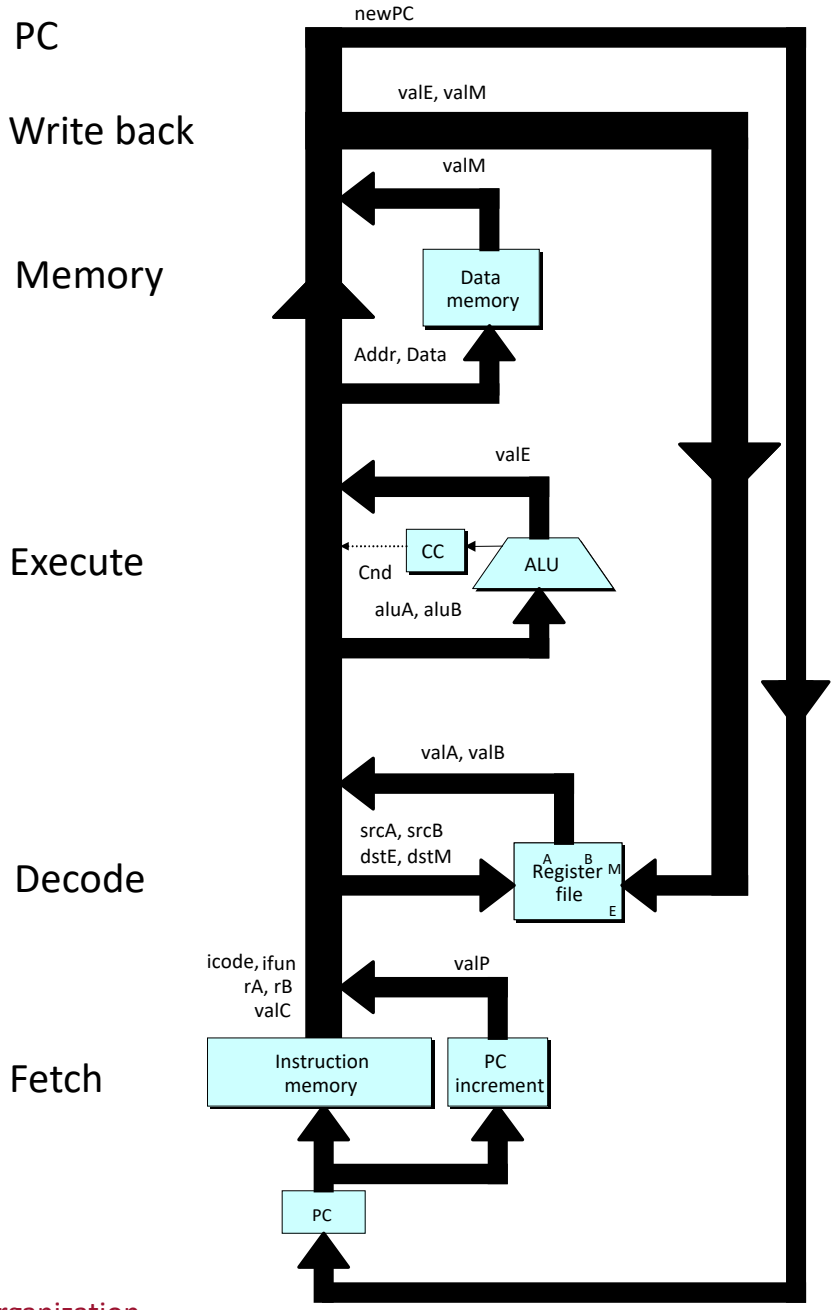
- State
  - Program counter register (PC)
  - Condition code register (CC)
    - ZF: Zero
    - SF:Negative
    - OF: Overflow
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions



SEQ Hardware Structure

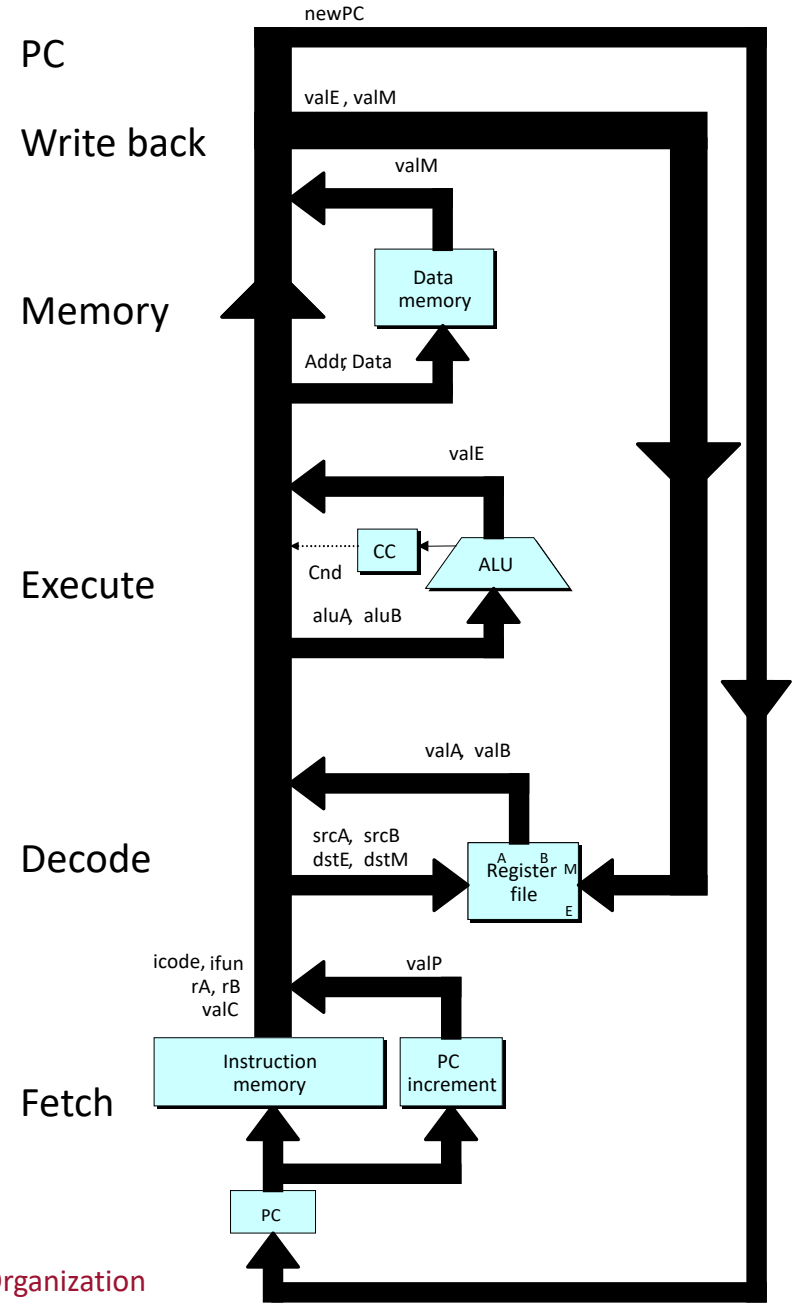
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter



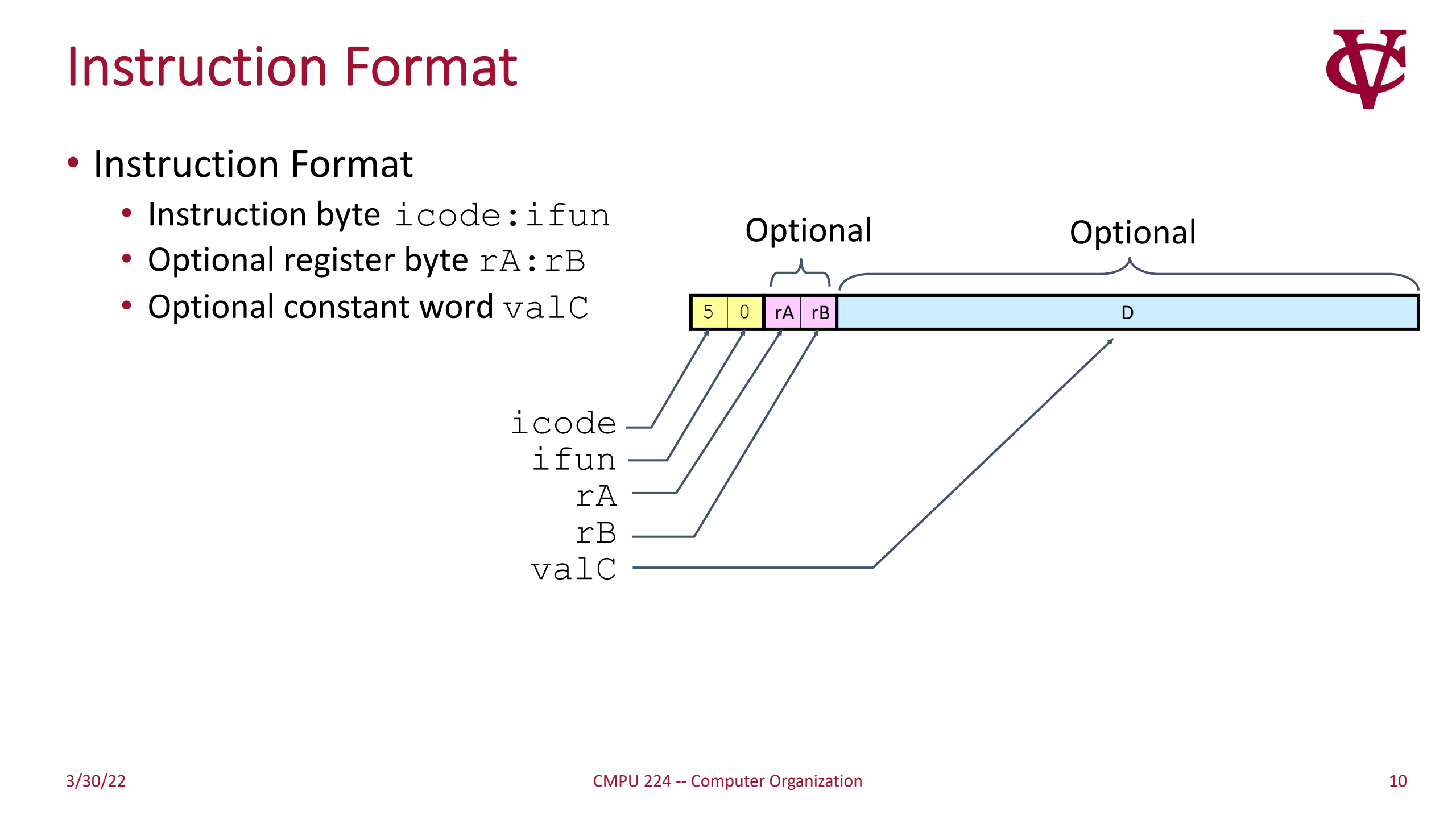


SEQ Stages

- Fetch
  - Read an instruction from Instruction Memory
- Decode
  - Gets values for the operands rA and rB
- Execute
  - Operation or address calculation
  - Sets Condition Codes
- Memory
  - Read or write memory
- Write Back
  - Update registers
- PC
  - Update program counter with next instruction address



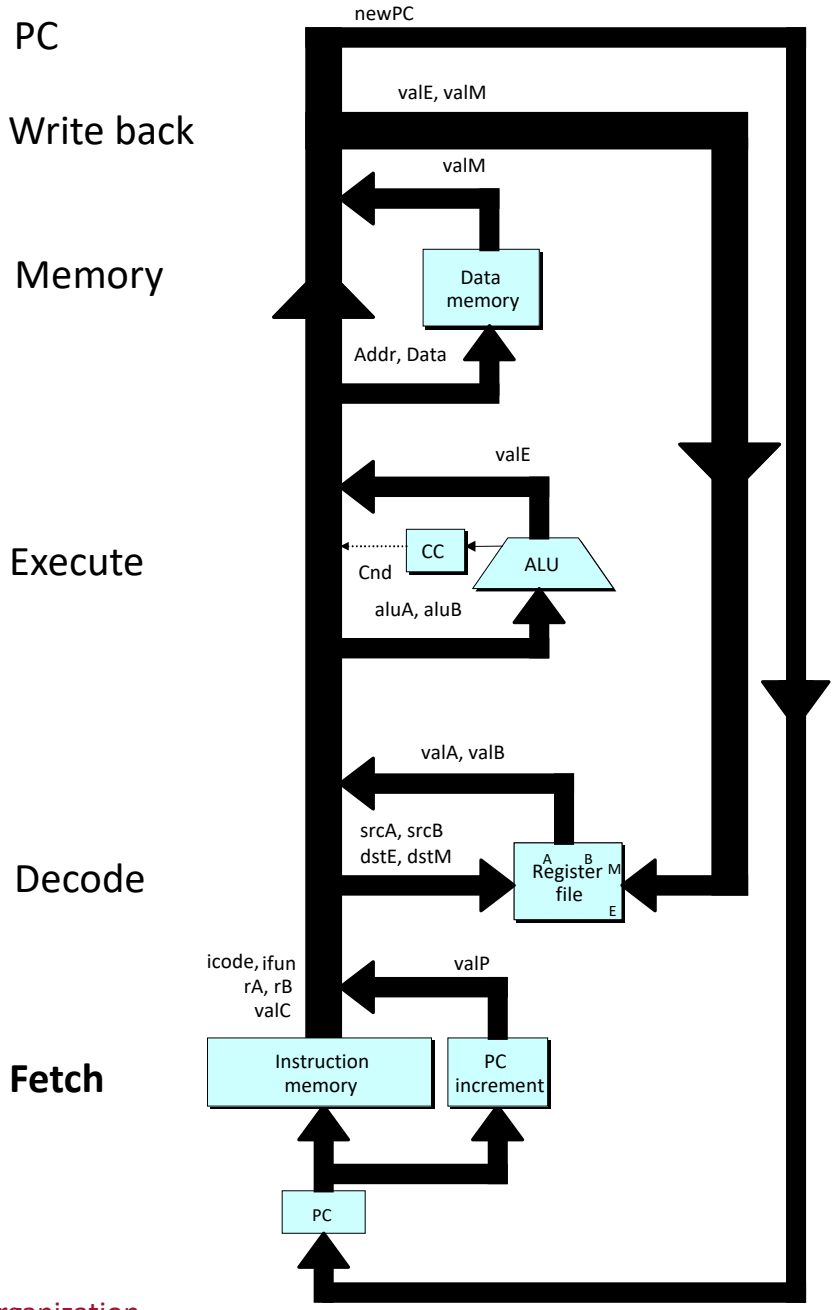
Instruction Format



- Instruction Format
  - Instruction byte `icode:ifun`
  - Optional register byte `rA:rB`
  - Optional constant word `valC`

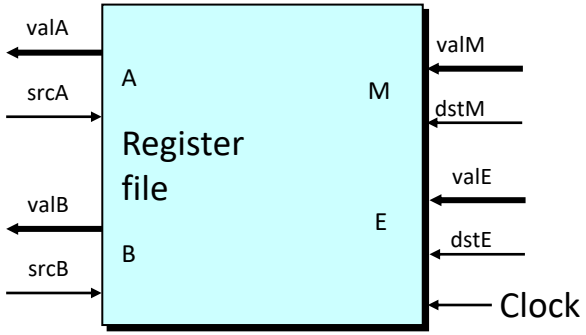
SEQ Stages -- Fetch

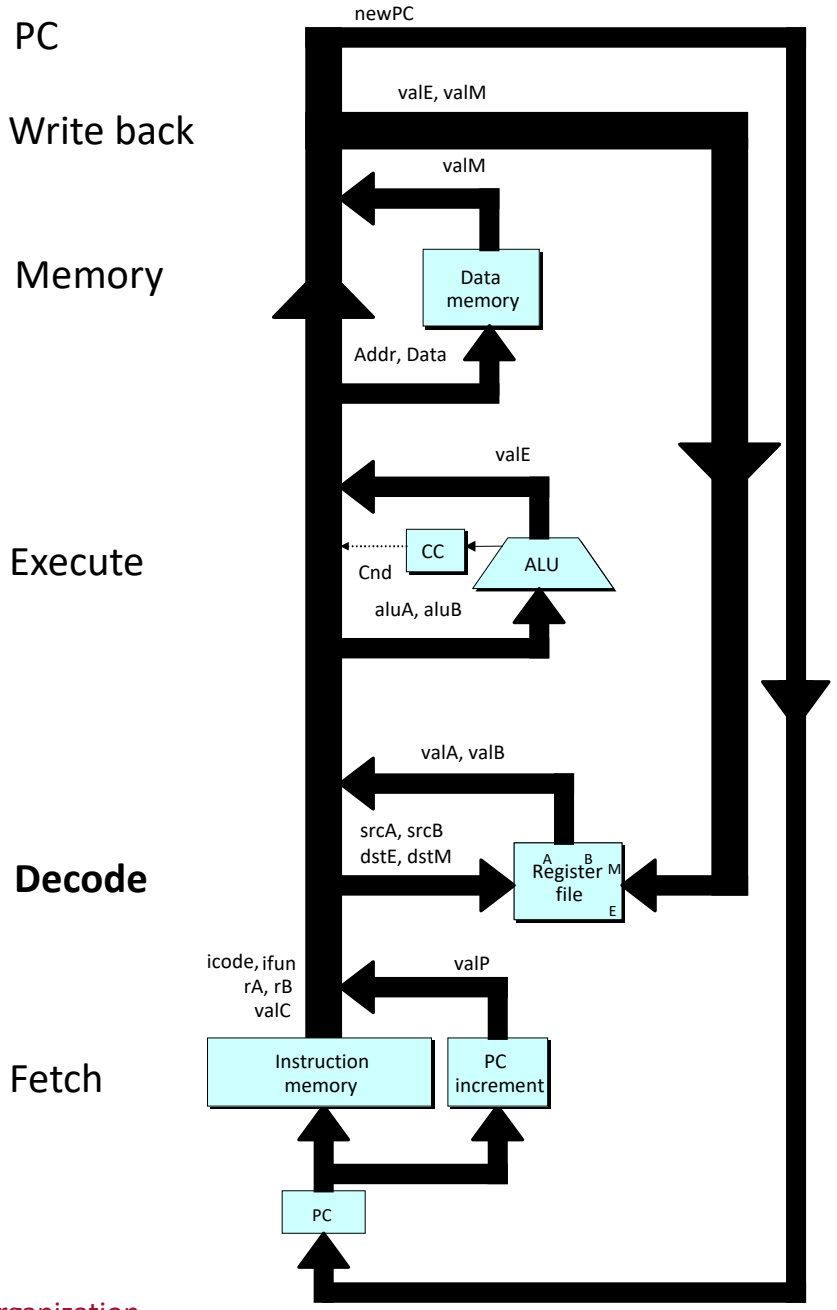
- Fetch
  - Read an instruction from Instruction Memory
  - Reads the bytes of an instruction from memory, using the Program Counter (PC) as the memory address
  - Extracts the **`icode`** and **`ifun`** values from the instruction
  - Optionally extracts register operand specifiers **`rA`** and **`rB`**
  - Optionally extracts 8-byte constant word **`valC`**
  - Computes the address of the instruction following the current one as **`valP`** (PC + length of the fetched instruction)



SEQ Stages -- Decode

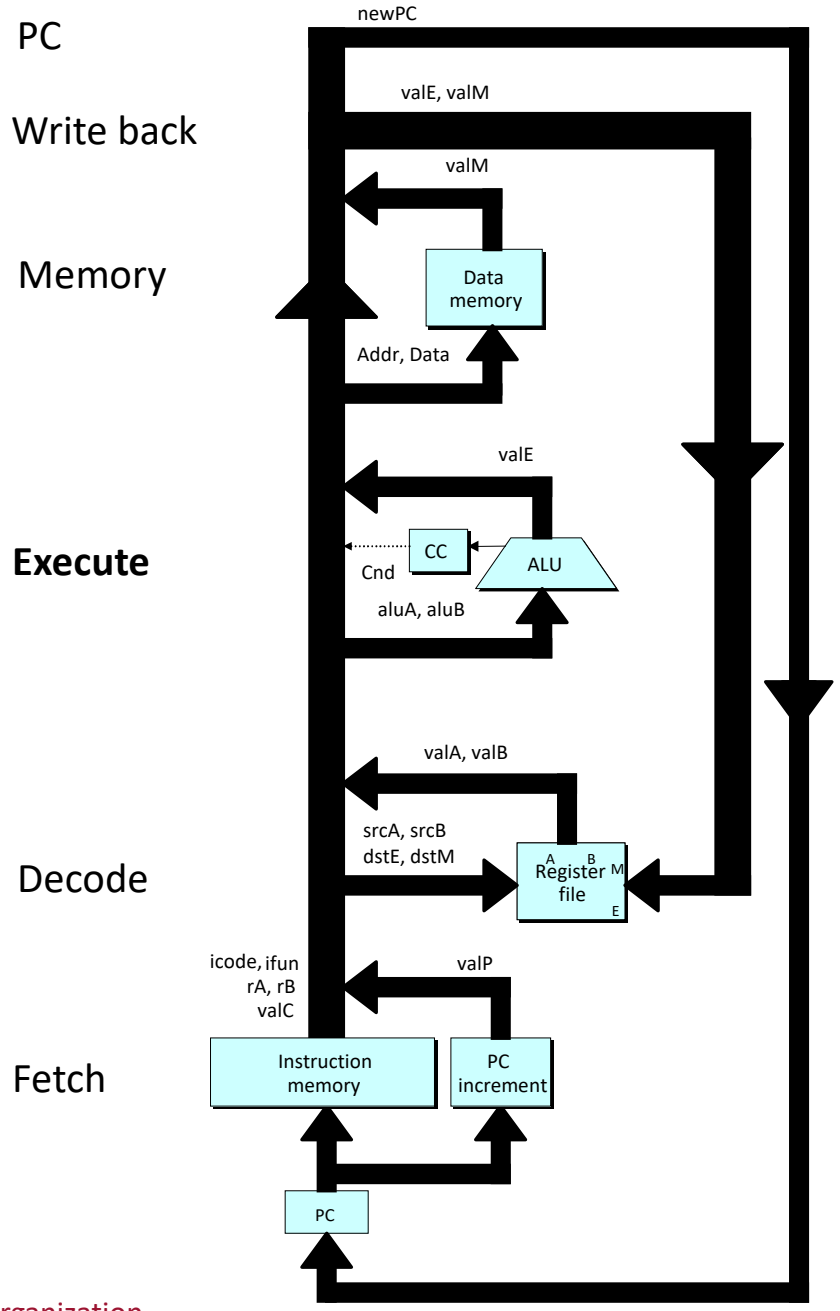
- Decode
  - Reads up to two operands from the register file, giving values **`valA`** and/or **`valB`**
  - Typically reads registers designated by **`rA`** and **`rB`**
  - For some instructions it reads register `%rsp`
    - Which ones?
    - `push`, `pop`, `call`, `ret`





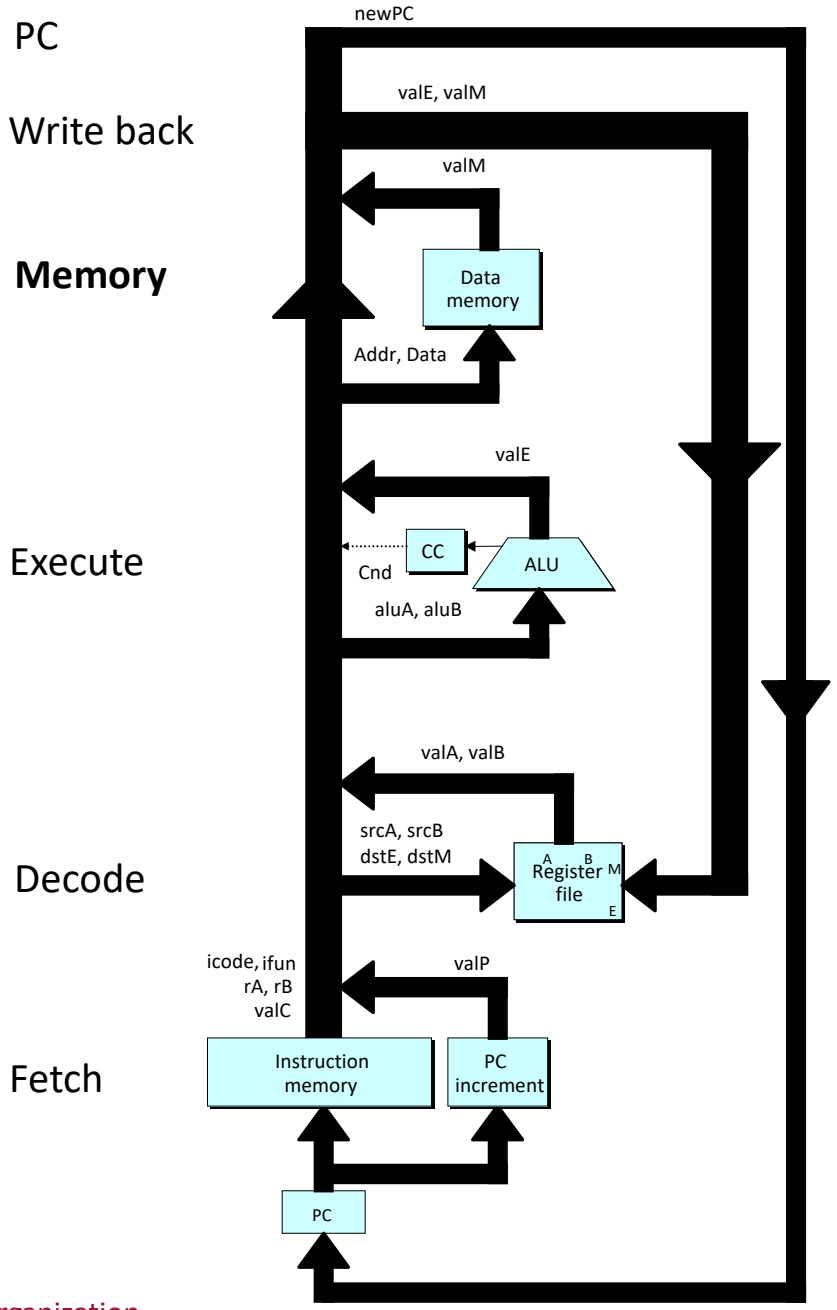
SEQ Stages -- Execute

- Execute
  - Compute value (math or logic)
    - Condition codes are possibly set
  - Compute memory address
    - `rmmovq` rA, D(rB)
  - Also handles jumps and conditional moves
  - **`valE`**, the value or address computed



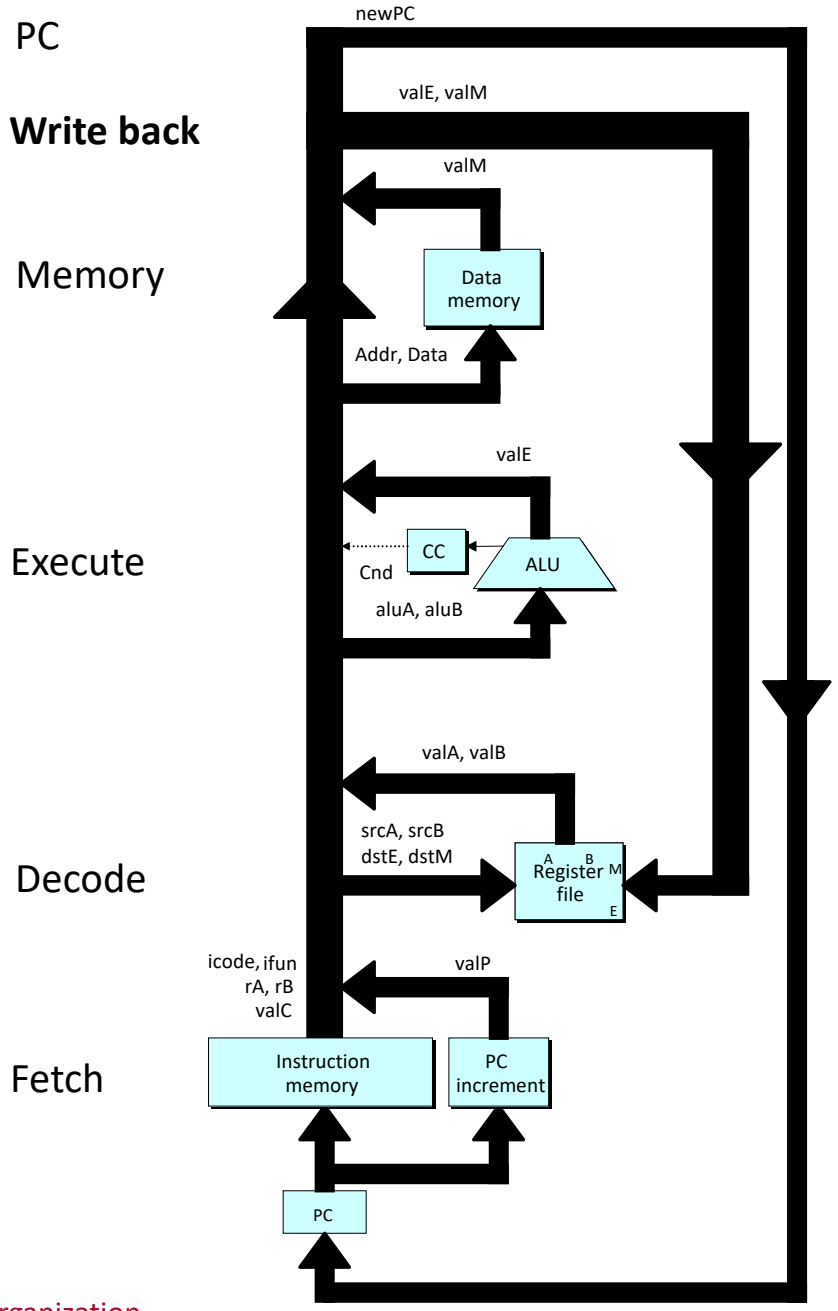
SEQ Stages -- Memory

- Memory
  - Either reads or writes data to memory
  - **`valM`**, the data read from memory



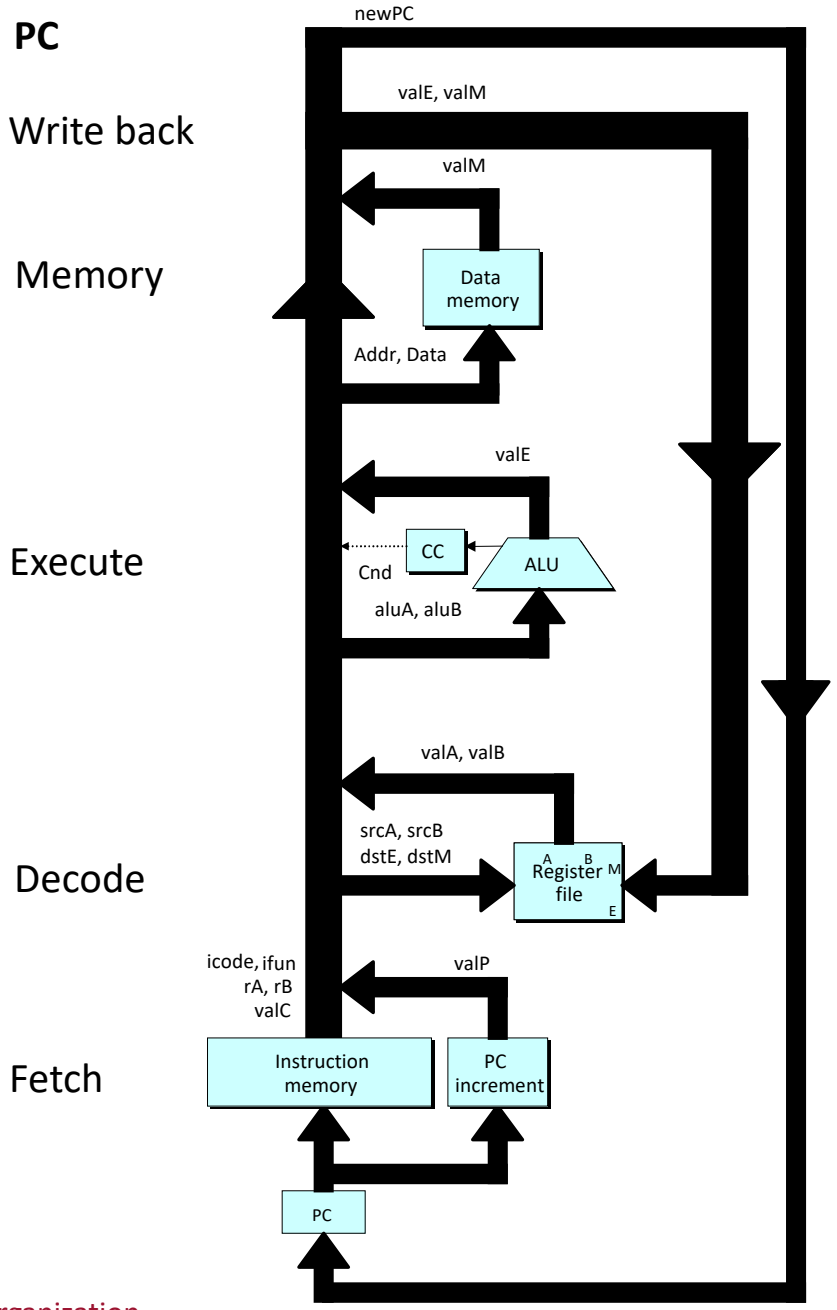
SEQ Stages -- Write Back

- Write Back
  - Writes up to two results to the register file: valE, valM
  - Why would you need to update two registers?
    - popq %rax
    - Need to update %rax and %rsp



SEQ Stages -- PC

- PC
  - Update program counter to the address of the next instruction



Executing Arithmetic/Logical Operation

- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers: rA, rB
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register: rB
- PC Update
  - Increment PC by 2

Stage Computation: Arith/Log Ops

	OPq rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Executing rmmovq

- Fetch
  - Read 10 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
- Write back
  - Do nothing
- PC Update
  - Increment PC by 10

Stage Computation: rmmovq

	`rmmovq` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_8[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- Use ALU for address computation

Executing popq

- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer (%rsp)
- Execute
  - Increment stack pointer by 8
- Memory
  - Read value at address from old stack pointer
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2

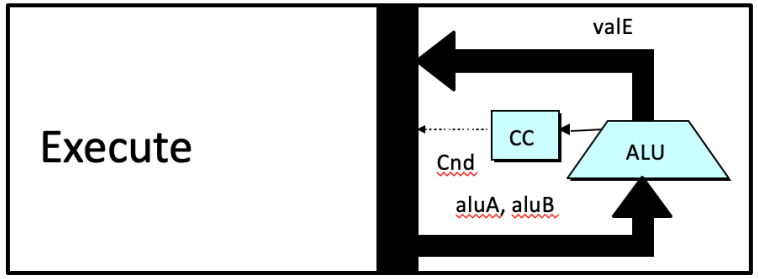
Stage Computation: popq

	`popq` rA	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valP ← PC+2	Compute next PC
Decode	valA ← R[`%rsp`]	Read stack pointer
	valB ← R[`%rsp`]	Read stack pointer
Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read from stack
Write back	R[`%rsp`] ← valE	Update stack pointer
	R[rA] ← valM	Write back result
PC update	PC ← valP	Update PC

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

Executing Conditional Moves

- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - If !cnd, then set destination register to 0xF
- Memory
  - Do nothing
- Write back
  - Update register (or not)
- PC Update
  - Increment PC by 2

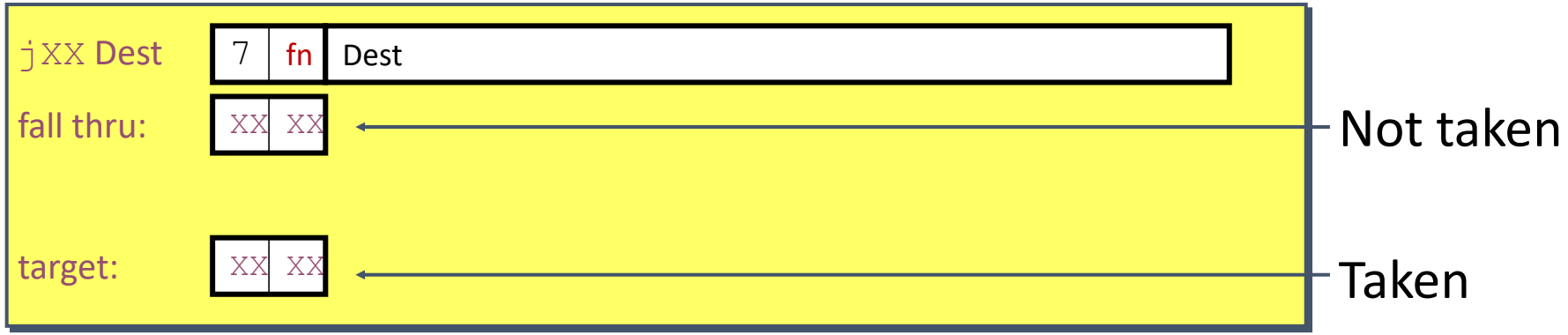


Stage Computation: Cond. Move

	cmovXX rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← 0	Read operand A
Execute	valE ← valB + valA If ! Cond(CC,ifun) rB ← 0xF	Pass valA through ALU (Disable register update)
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Read register rA and pass through ALU
- Cancel move by setting destination register to 0xF
  - If condition codes & move condition indicate no move

Executing Jumps



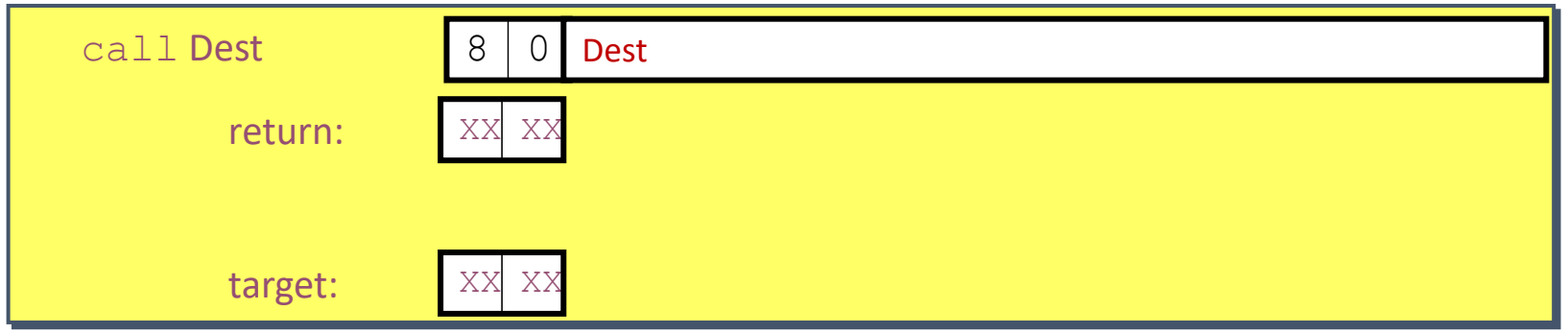
- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	Read instruction byte Read destination address Fall through address
Decode		
Execute	Cnd ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Cnd ? valC : valP	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Executing call



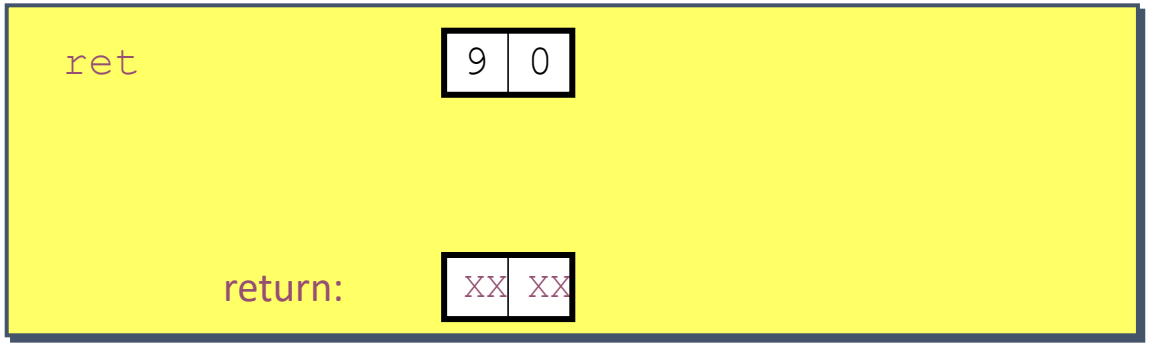
- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 8
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

Stage Computation: call

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_8[PC+1]	Read destination address
	valP ← PC+9	Compute return point
Decode	valB ← R[`%rsp`]	Read stack pointer
Execute	valE ← valB + –8	Decrement stack pointer
Memory	M_8[valE] ← valP	Write return value on stack
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

Executing ret



- Fetch
  - Read 1 byte
- Decode
  - Read `%rsp`
- Execute
  - Calculate `%rsp + 8`
- Memory
  - Read return address M[`%rsp`]
- Write back
  - Update `%rsp`
- PC Update
  - Set PC to return address

Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[`%rsp`] valB ← R[`%rsp`]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read return address
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

Computation Steps

		OPq rA, rB	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB	rA:rB ← M_1[PC+1]	Read register byte
	valC		[Read constant word]
	valP	valP ← PC+2	Compute next PC
Decode	valA, srcA	valA ← R[rA]	Read operand A
	valB, srcB	valB ← R[rB]	Read operand B
Execute	valE	valE ← valB OP valA	Perform ALU operation
	Cond code	Set CC	Set/use cond. code reg
Memory	valM		[Memory read/write]
Write back	dstE	R[rB] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valP	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computation Steps

		`call` Dest	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_8[PC+1]	Read constant word
	valP	valP ← PC+9	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[`%rsp`]	Read operand B
Execute	valE	valE ← valB + –8	Perform ALU operation
	Cond code		[Set /use cond. code reg]
Memory	valM	M_8[valE] ← valP	Memory read/write
Write back	dstE	R[`%rsp`] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valC	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computed Values

- Fetch
  - icode — Instruction code
  - ifun — Instruction function
  - rA — Instr. Register A
  - rB — Instr. Register B
  - valC — Instruction constant
  - valP — Incremented PC
- Decode
  - srcA — Register ID A
  - srcB — Register ID B
  - dstE — Destination Register E
  - dstM — Destination Register M
  - valA — Register value A
  - valB — Register value B
- Execute
  - valE — ALU result
  - Cnd — Branch/move flag
- Memory
  - valM — Value from memory