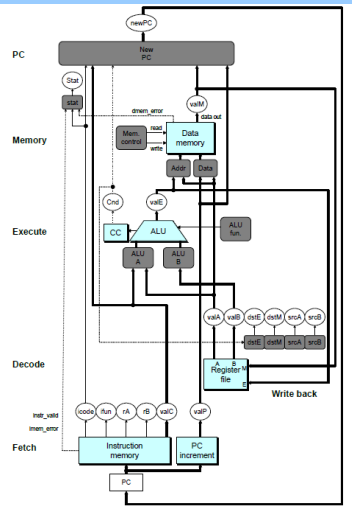
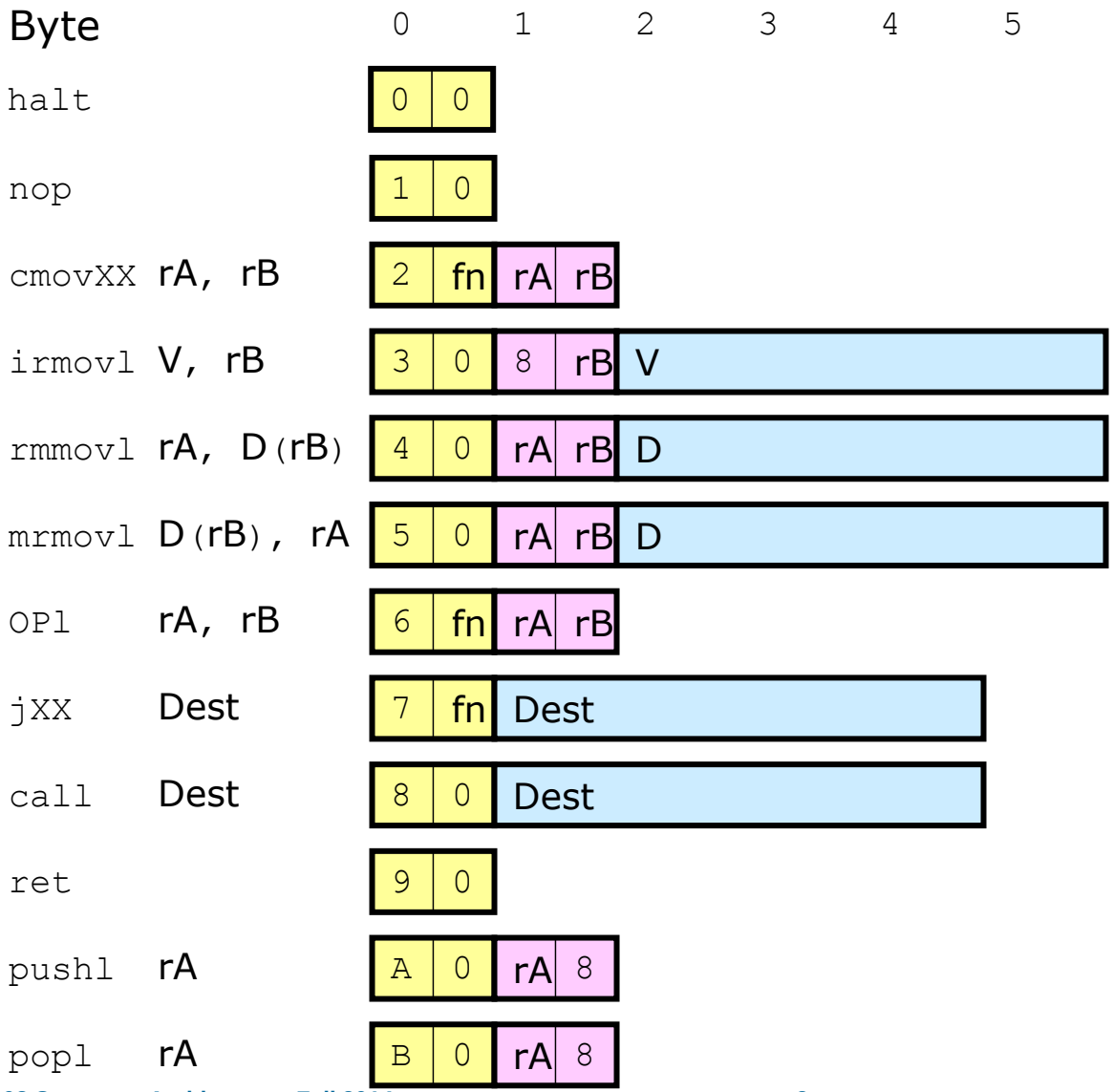
Processor Architecture

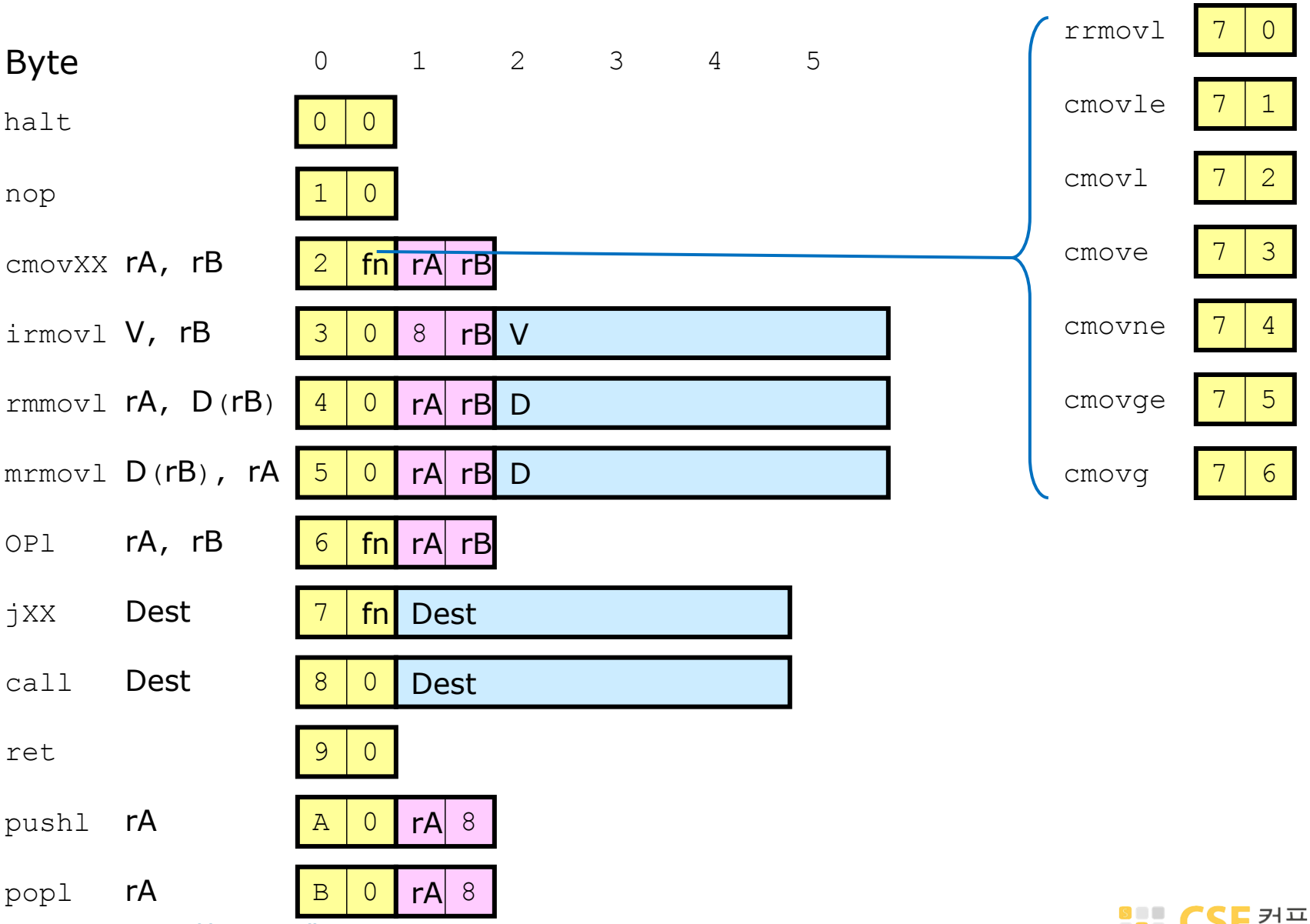
Y86 Sequential Implementation



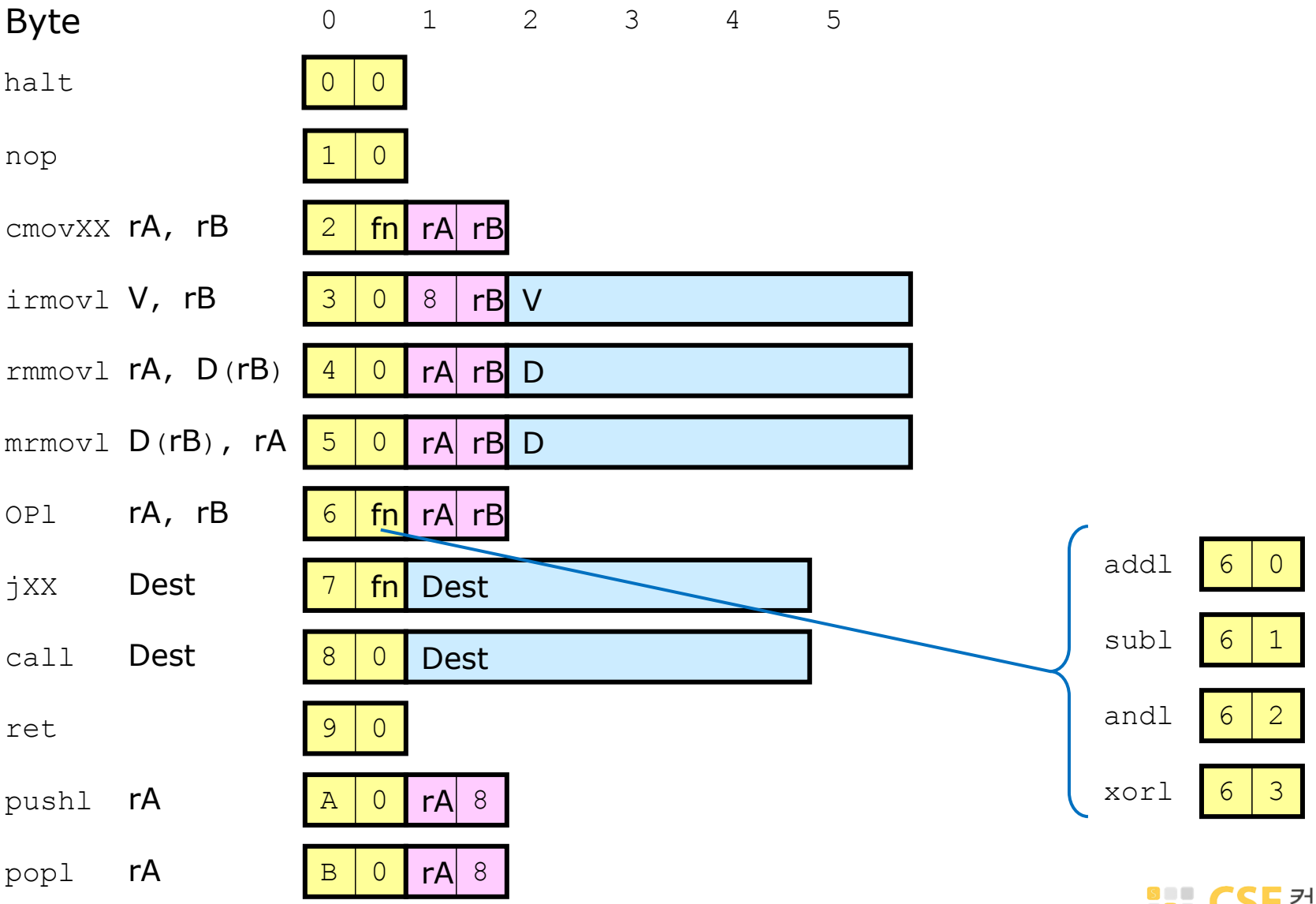
Recap: Y86 Instruction Set Overview



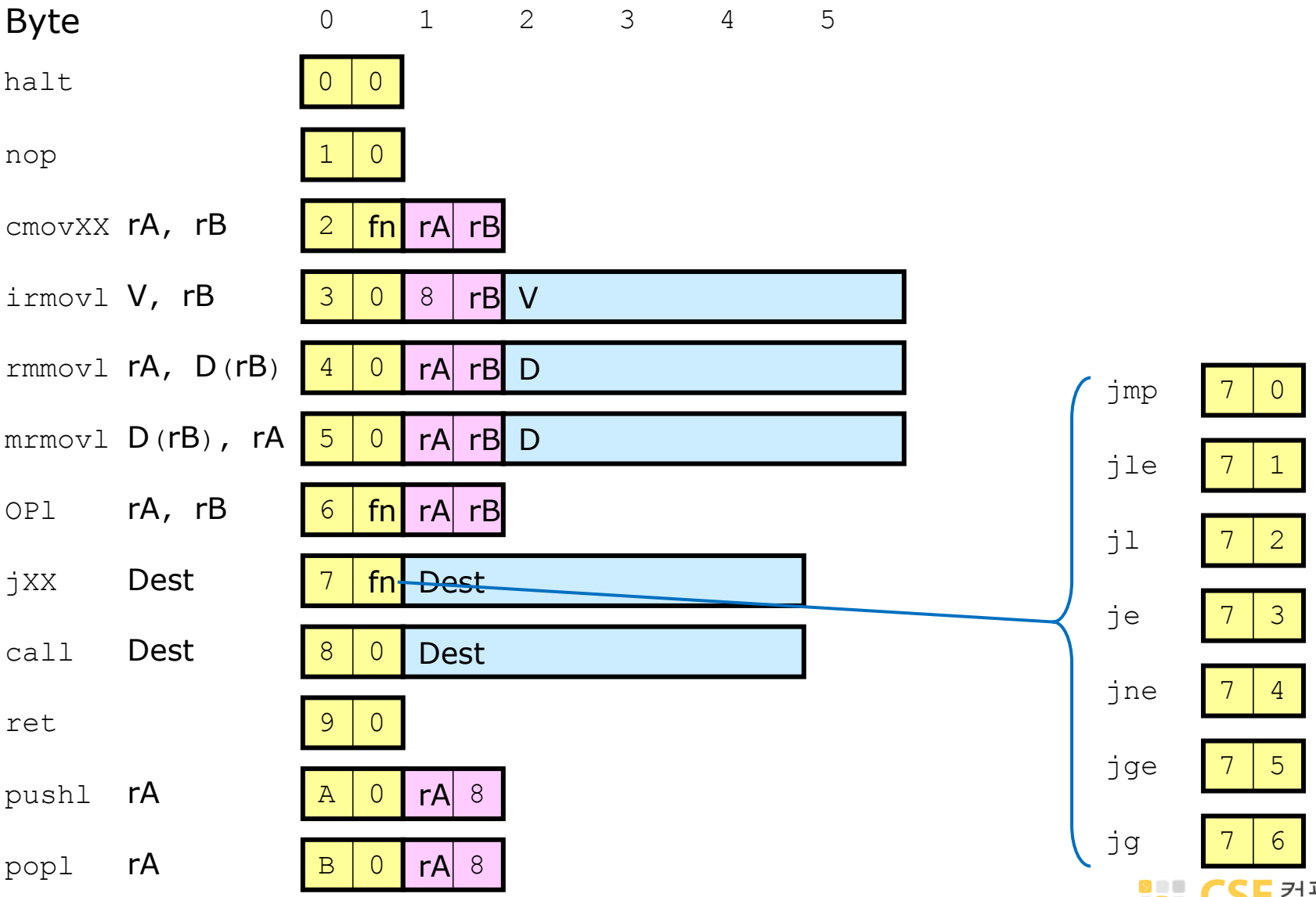
Recap: Y86 Instruction Set – cmovXX



Recap: Y86 Instruction Set – OPl

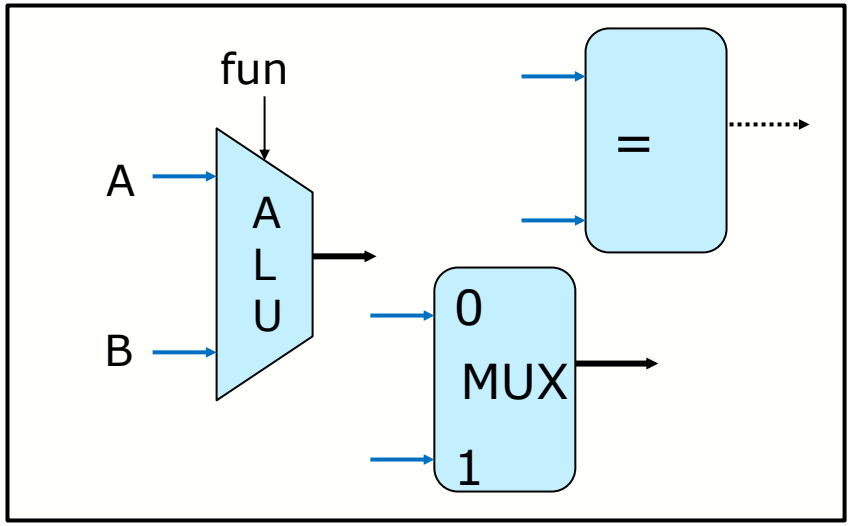


Recap: Y86 Instruction Set – jXX

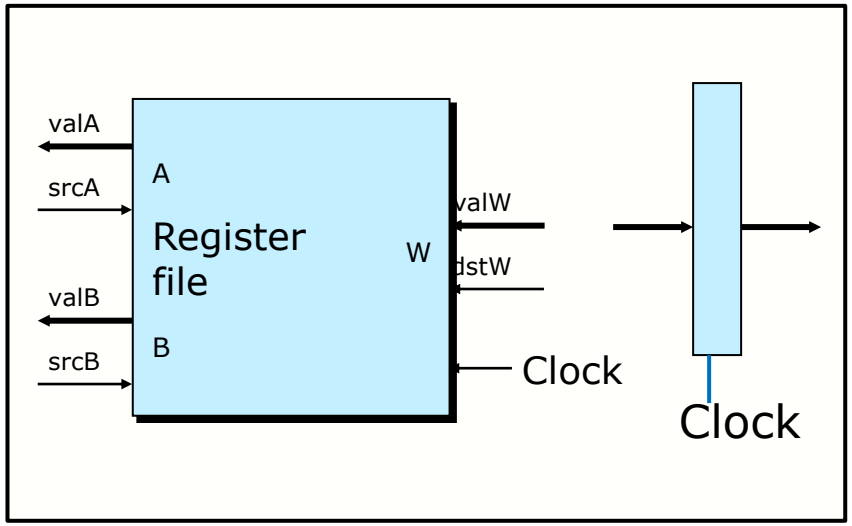


Recap: Building Blocks

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



- Storage Elements
  - Store bits
  - Addressable memories
  - Non-addressable registers
  - Loaded only as clock rises



Recap: Hardware Control Language

- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify

- Data Types
  - bool: Boolean
    - a, b, c, …
  - int: words
    - A, B, C, …
    - Does not specify word size---bytes, 32-bit words, …

- Statements
  - bool a = bool-expr ;
  - int A = int-expr ;

Recap: HCL Operations

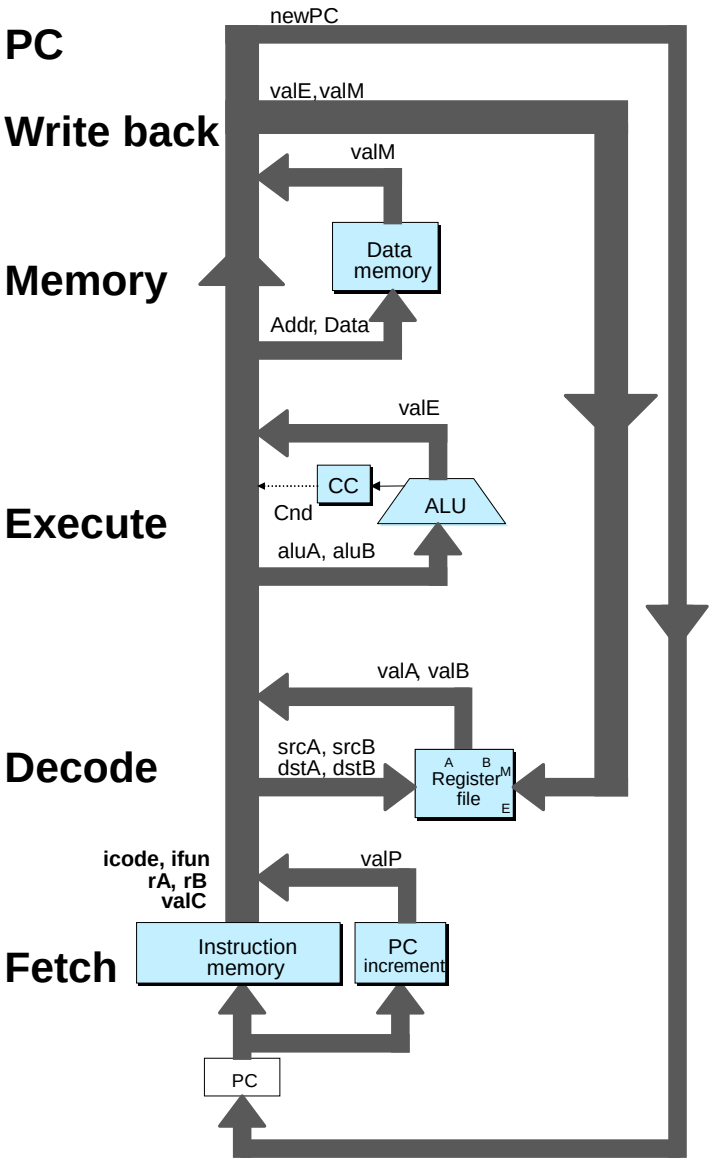
- Classify by type of value returned

- Boolean Expressions
  - 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

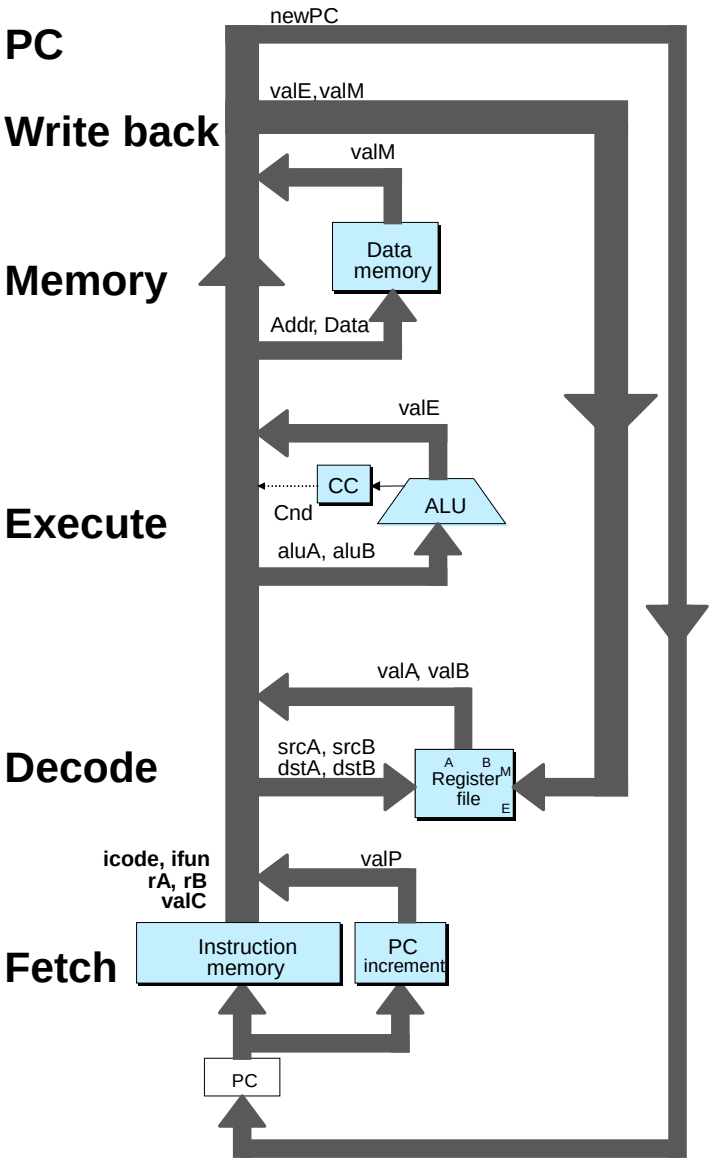
SEQ Hardware Structure

- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

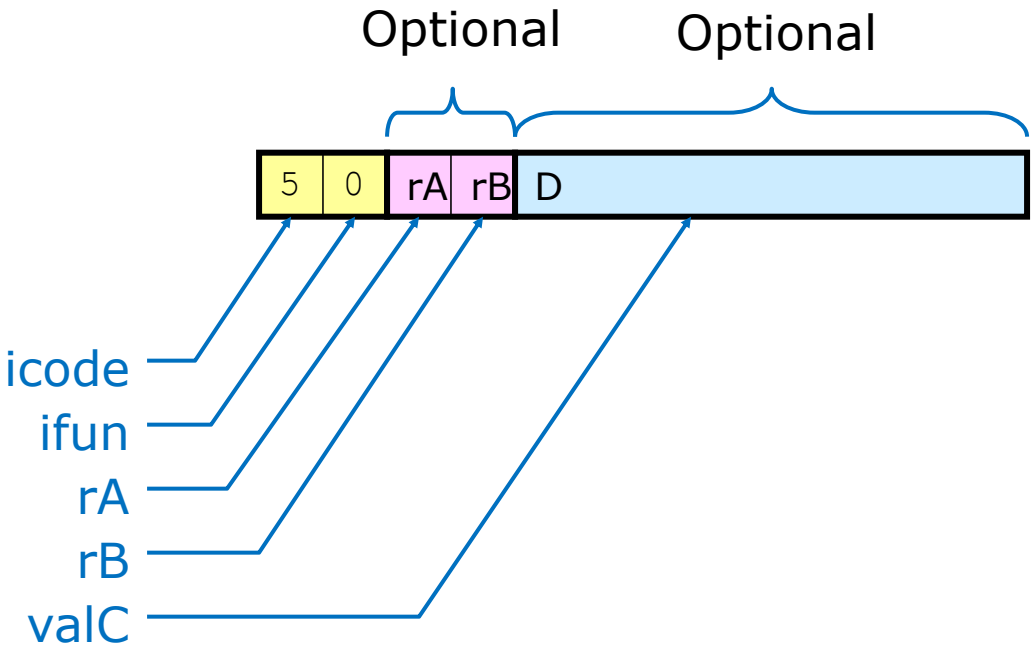


SEQ Stages

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



Instruction Decoding



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

Executing Arithmetic/Logical Operations

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

Stage Computation: Arithmetic/Logical Ops

	OPl 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 `rmmovl`

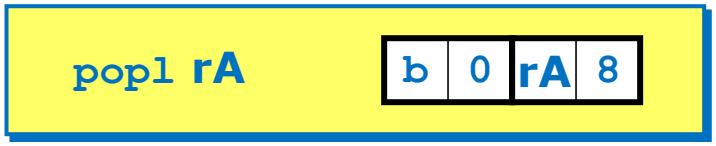
- Fetch
  - Read 6 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
- Write back
  - Do nothing
- PC Update
  - Increment PC by 6

Stage Computation: `rmmovl`

	`rmmovl` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_4[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- Use ALU for address computation

Executing `popl`



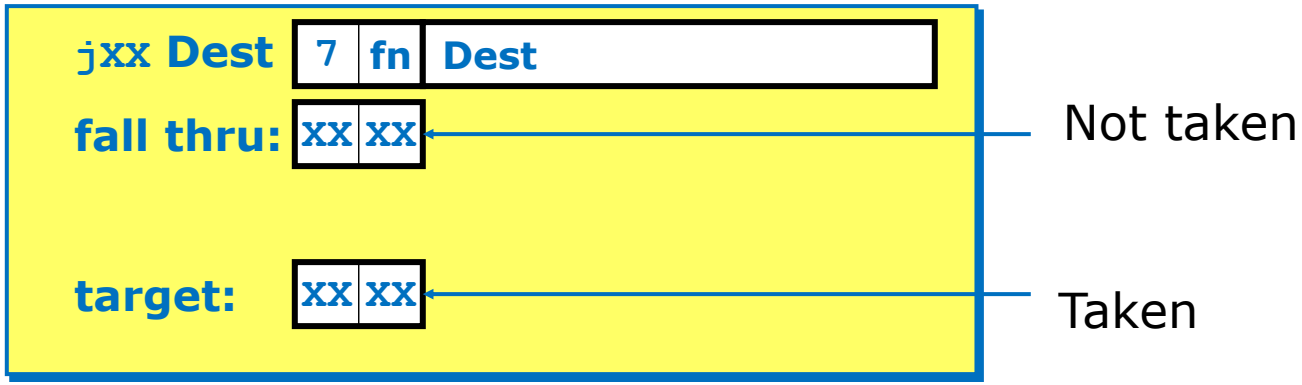
- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read from old stack pointer
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2

Stage Computation: `popl`

	`popl` rA	
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[`%esp`] valB ← R[`%esp`]	Read stack pointer Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read from stack
Write back	R[`%esp`] ← valE R[rA] ← valM	Update stack pointer Write back result
PC update	PC ← valP	Update PC

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

Executing Jumps



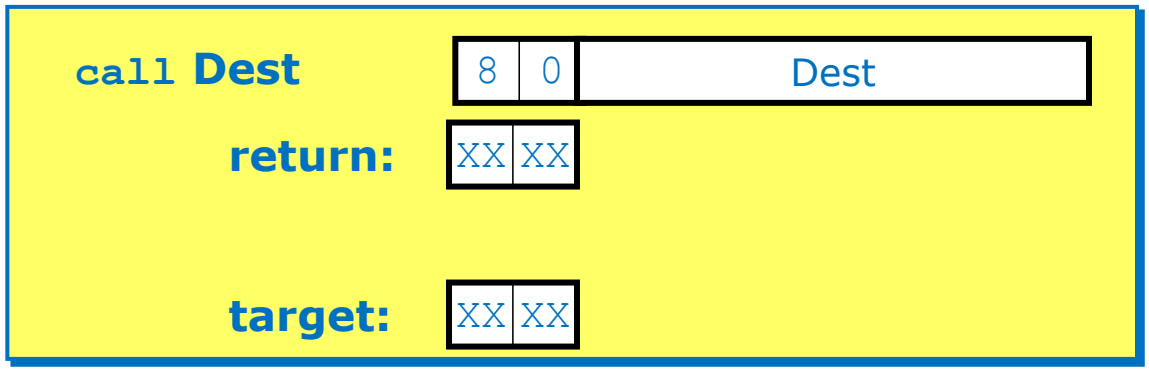
- Fetch
  - Read 5 bytes
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to PC + 5 if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	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`



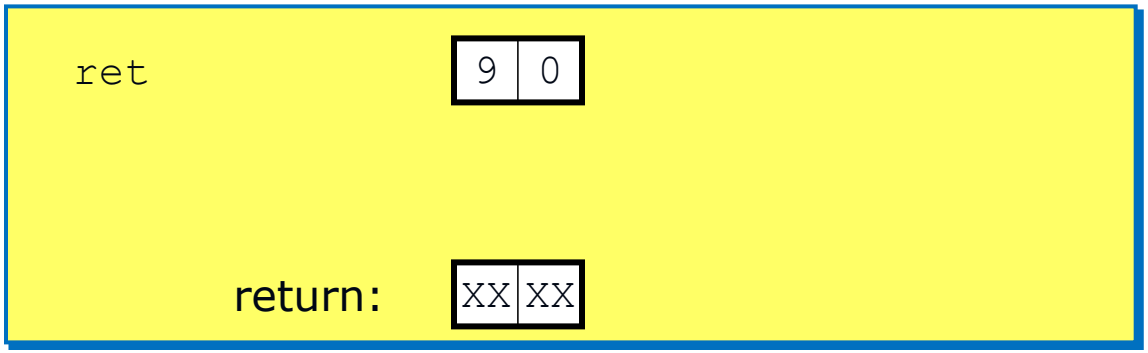
- Fetch
  - Read 5 bytes
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 4
- Memory
  - Write PC + 5 to the memory location pointed to by the decremented stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

Stage Computation: `call`

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_4[PC+1]	Read destination address
	valP ← PC+5	Compute return point
Decode	valB ← R[`%esp`]	Read stack pointer
Execute	valE ← valB + −4	Decrement stack pointer
Memory	M_4[valE] ← valP	Write return value on stack
Write back	R[`%esp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- Use ALU to decrement stack pointer

Executing `ret`



- Fetch
  - Read 1 byte
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read return address from the memory location pointed to by the old stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to return address

Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[`%esp`] valB ← R[`%esp`]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read return address
Write back	R[`%esp`] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

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

Computation Steps

		OPl 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 condition code register
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 the same general pattern
- Differ in what gets computed during 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_4[PC+1]	Read constant word
	valP	valP ← PC+5	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[`%esp`]	Read operand B
Execute	valE	valE ← valB + –4	Perform ALU operation
	Cond code		[Set condition code reg.]
Memory	valM	M_4[valE] ← valP	Memory read/write
Write back	dstE	R[`%esp`] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valC	Update PC

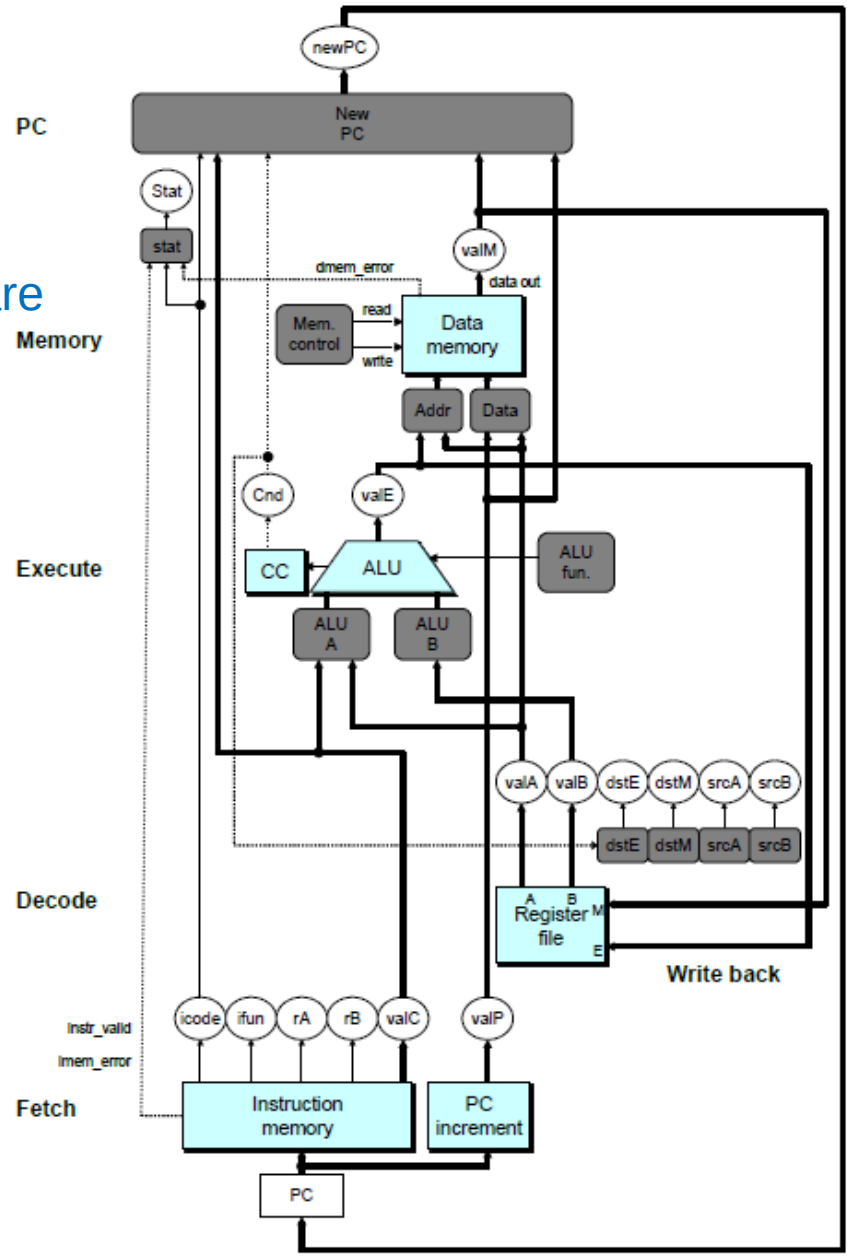
- All instructions follow the same general pattern
- Differ in what gets computed during each step

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

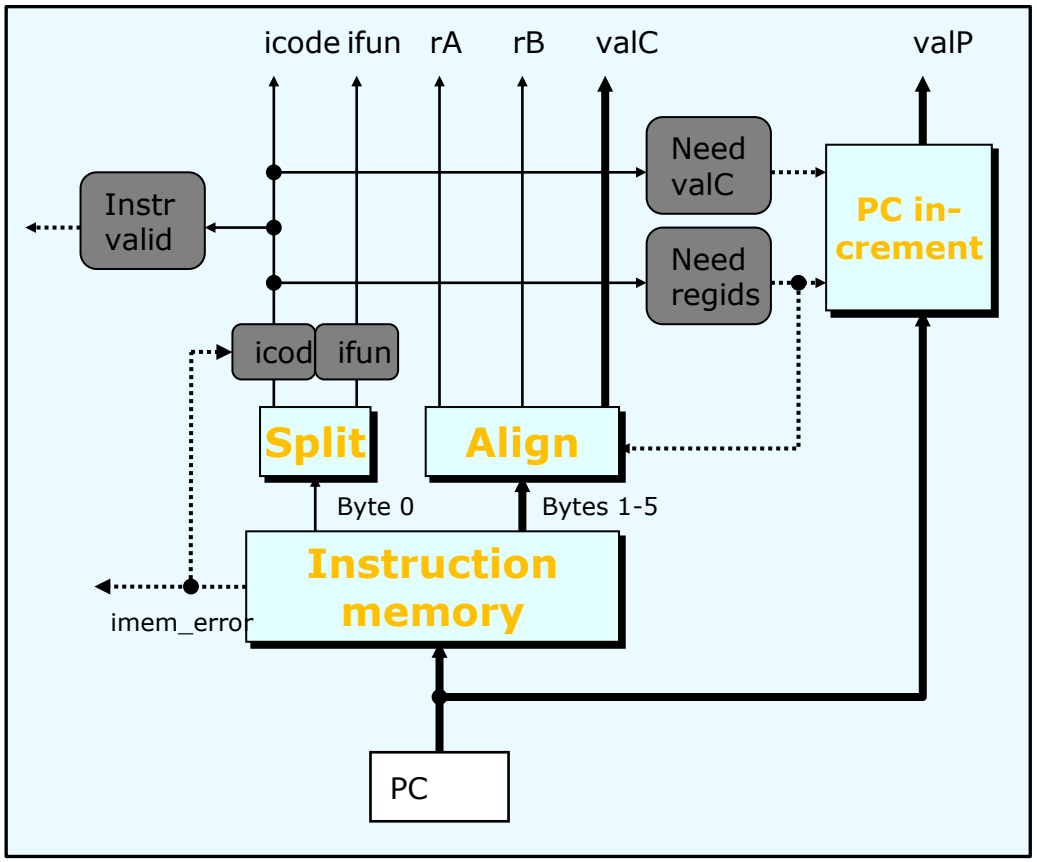
SEQ Hardware

- Key
  - Blue boxes: predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes : control logic
    - Described in HCL
  - White ovals : labels for signals
  - Thick lines : 32-bit word values
  - Thin lines : 4-8 bit values
  - Dotted lines : 1-bit values



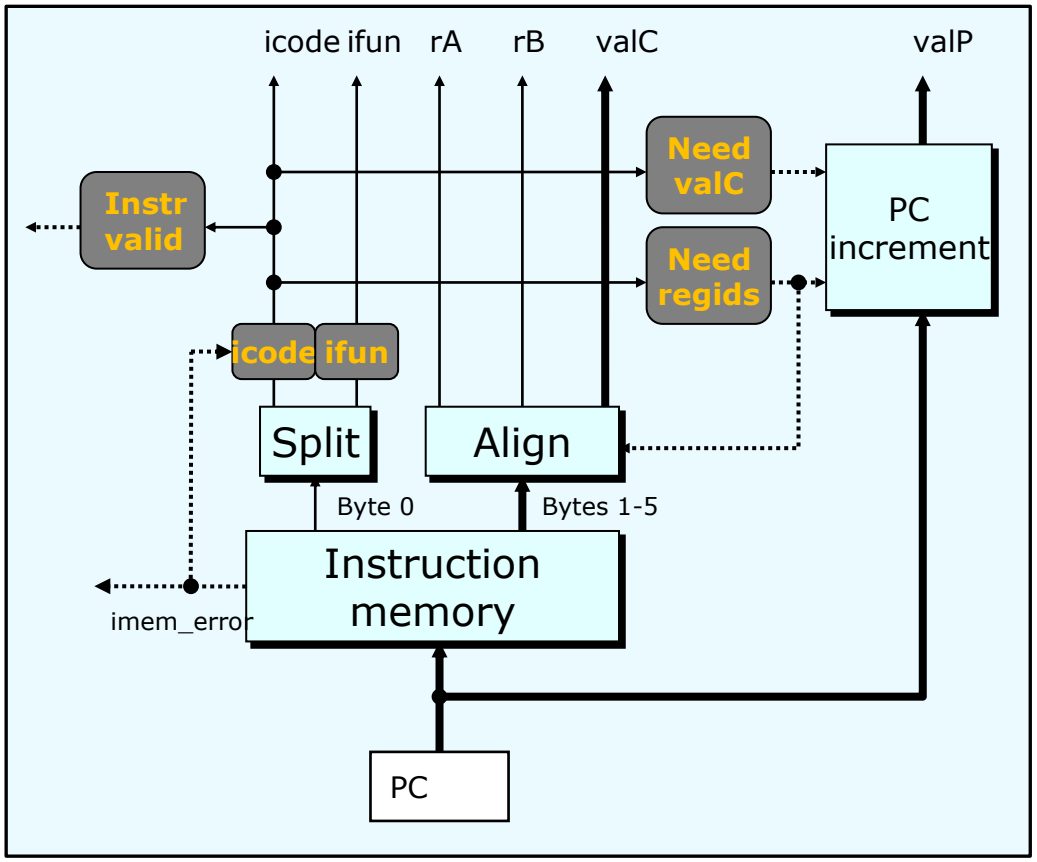
Fetch Logic #1

- Predefined Blocks
  - PC : Register containing PC
  - Instruction memory : Read 6 bytes (PC to PC+5)
    - Signal invalid address
  - Split : Divide instruction byte into icode and ifun
  - Align : Get fields for rA, rB, and valC



Fetch Logic #2

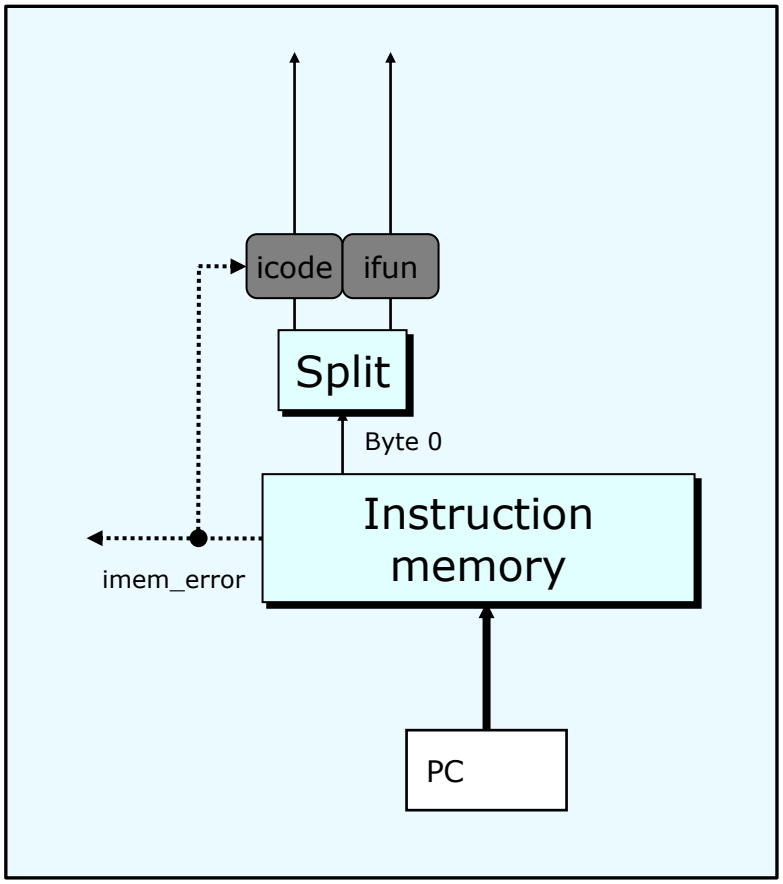
- Control Logic
  - Instr. Valid
    - Is this instruction valid?
  - icode, ifun
    - Generate no-op if invalid address
  - Need regids
    - Does this instruction have a register byte?
  - Need valC
    - Does this instruction have a constant word?



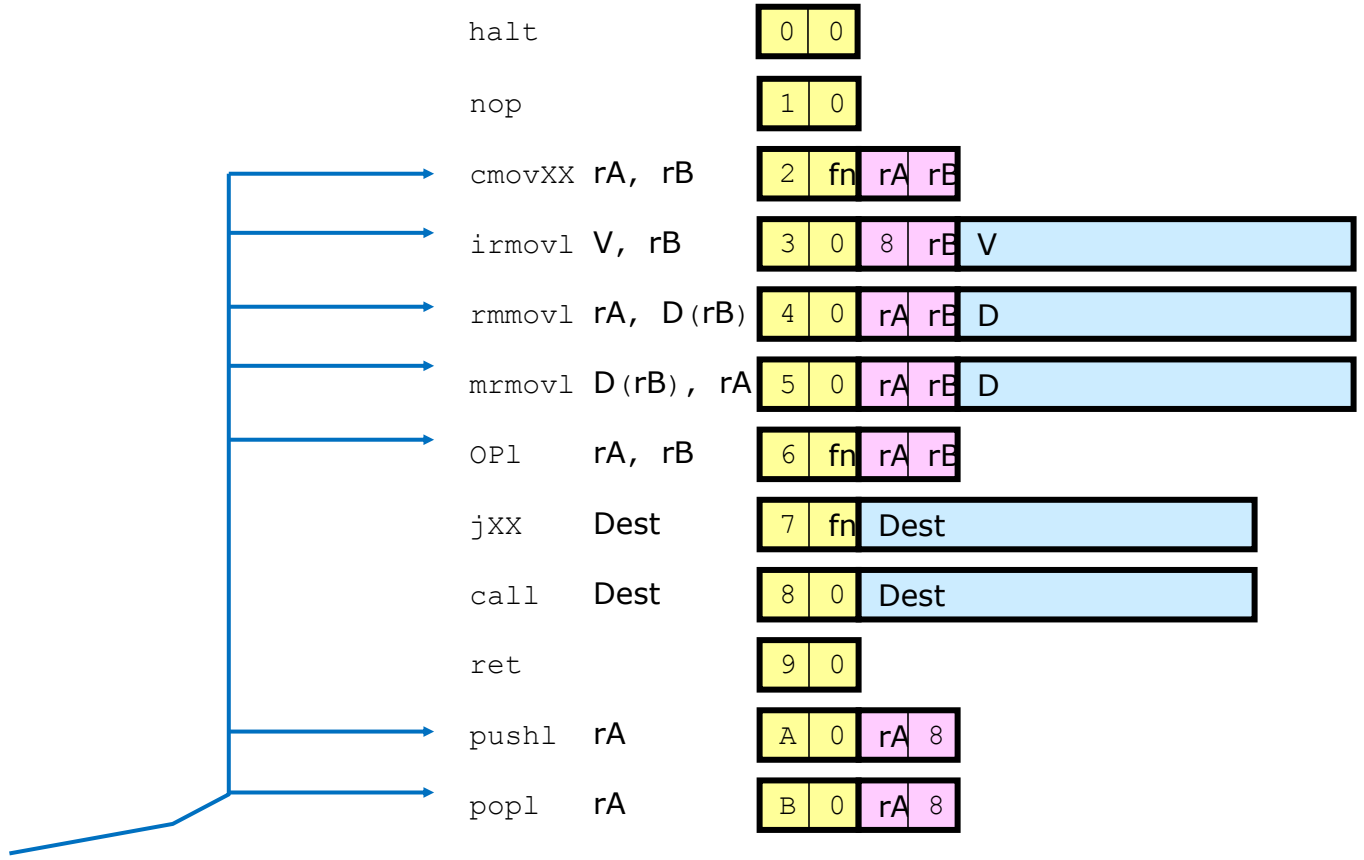
Fetch Control Logic in HCL

```
# Determine instruction code
int icode = [
        imem_error: INOP;
        1: imem_icode;
];

# Determine instruction function
int ifun = [
        imem_error: FNONE;
        1: imem_ifun;
];
```



Fetch Control Logic in HCL



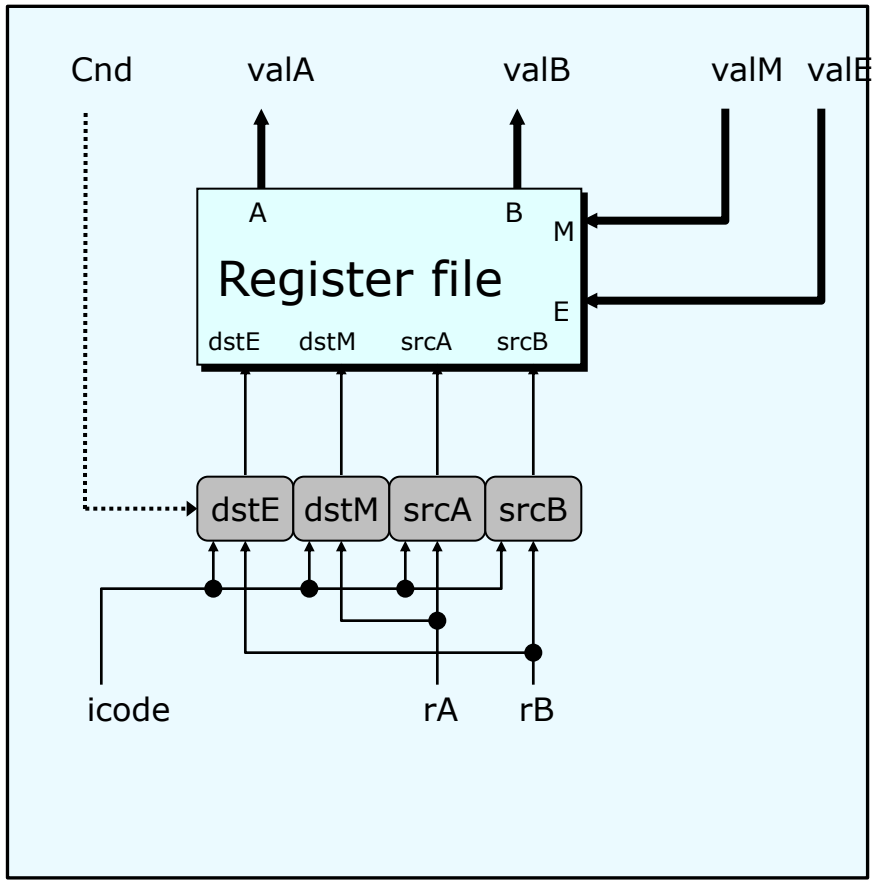

```
bool need_regids =
        icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                   IIRMOVL, IRMMOVL, IMRMOVL };

bool instr_valid = icode in
        {IHALT, INOP, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
         IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
```

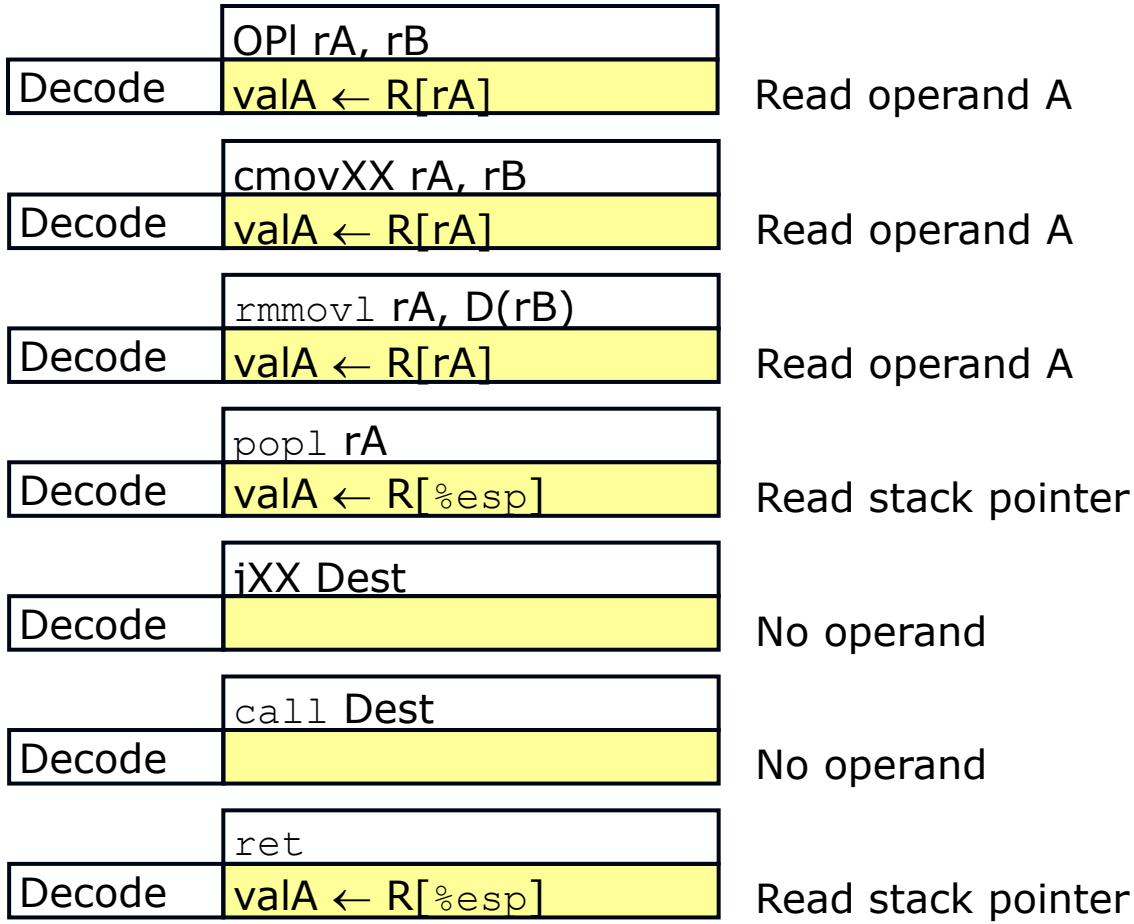
Decode Logic

- Register File
  - Read ports A, B
  - Write ports E, M
  - Addresses are register IDs or 15 (0xF) (no access)
- Control Logic
  - srcA, srcB: read port addresses
  - dstE, dstM: write port addresses
- Signals
  - Cnd: Indicate whether or not to perform conditional move

    -> Computed in Execute stage

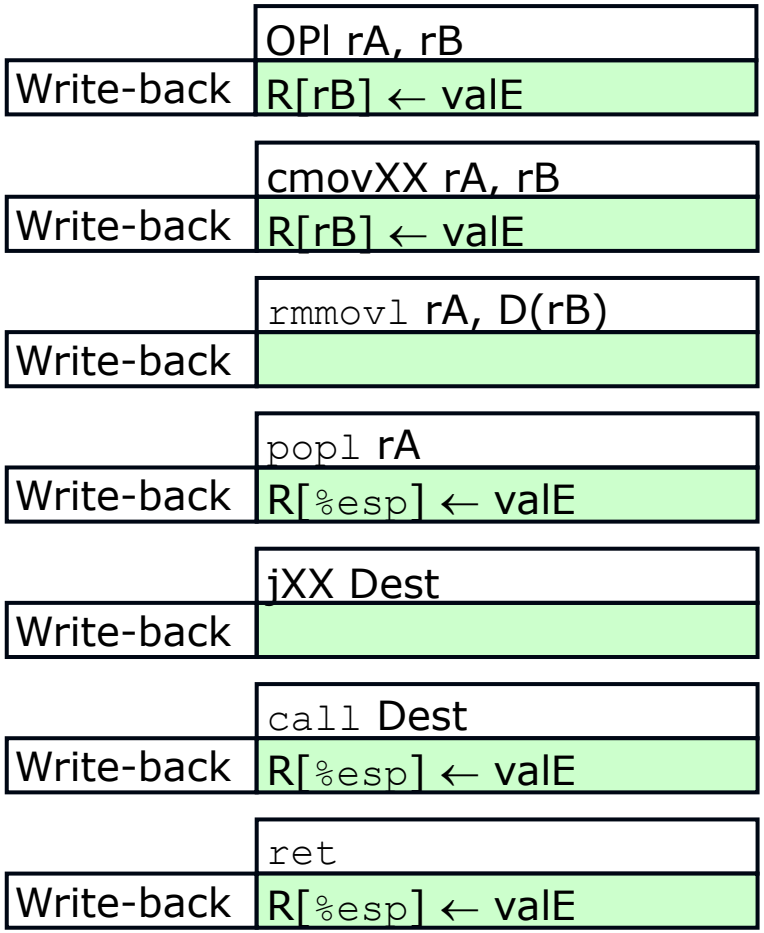


A Source




```
int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

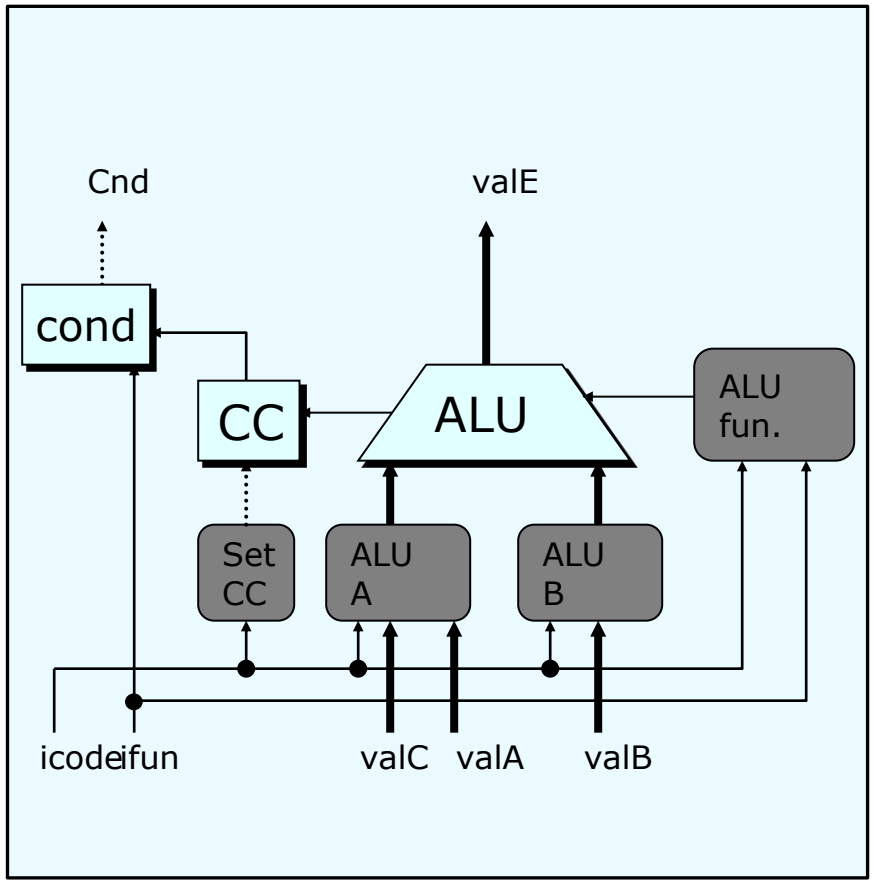
E Destination




```
int dstE = [
        icode in { IRRMOVL } && Cnd : rB;
        icode in { IIRMOVL, IOPL} : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't write any register
];
```

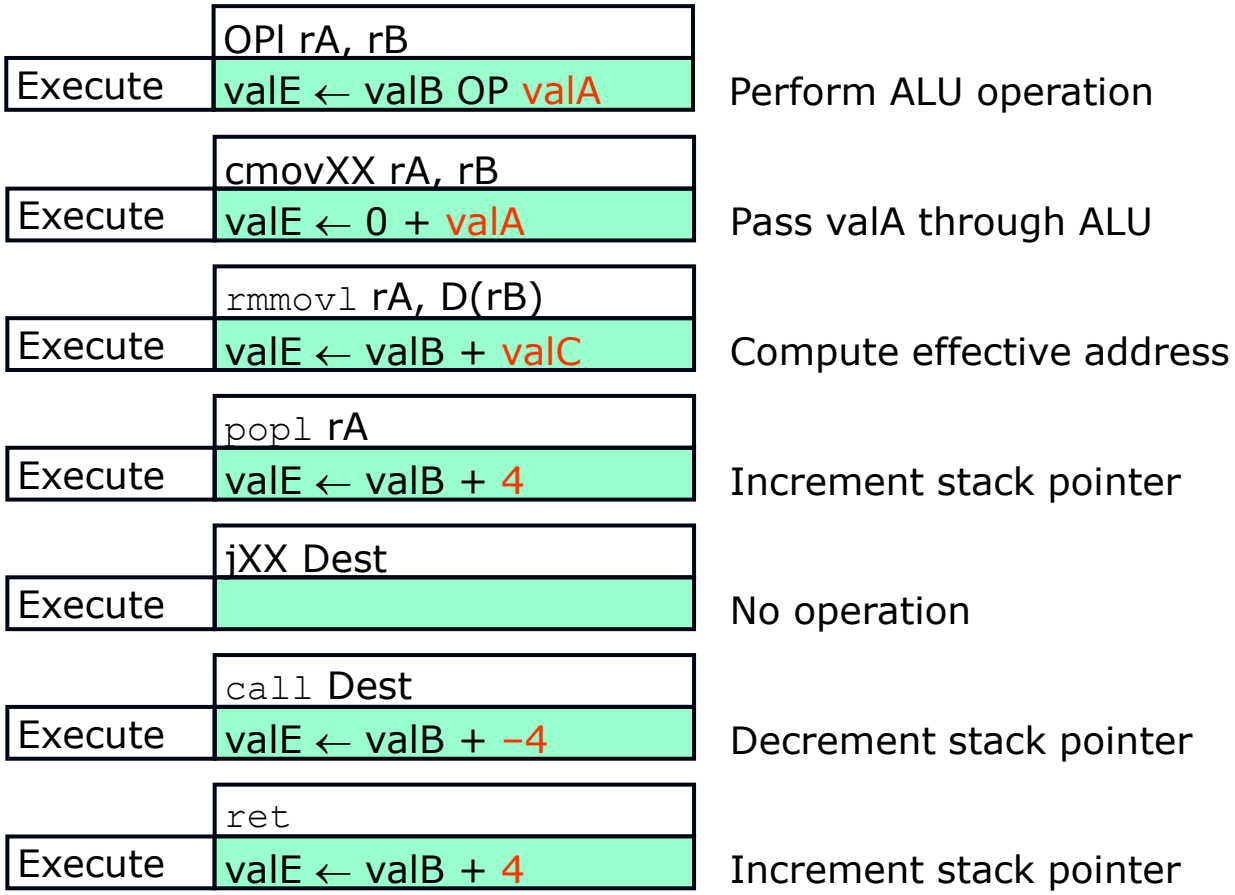
Execute Logic

- Units
  - ALU
    - Implements 4 required functions
    - Generates condition code values
  - CC
    - Register with 3 condition code bits
  - cond
    - Computes conditional jump/move flag



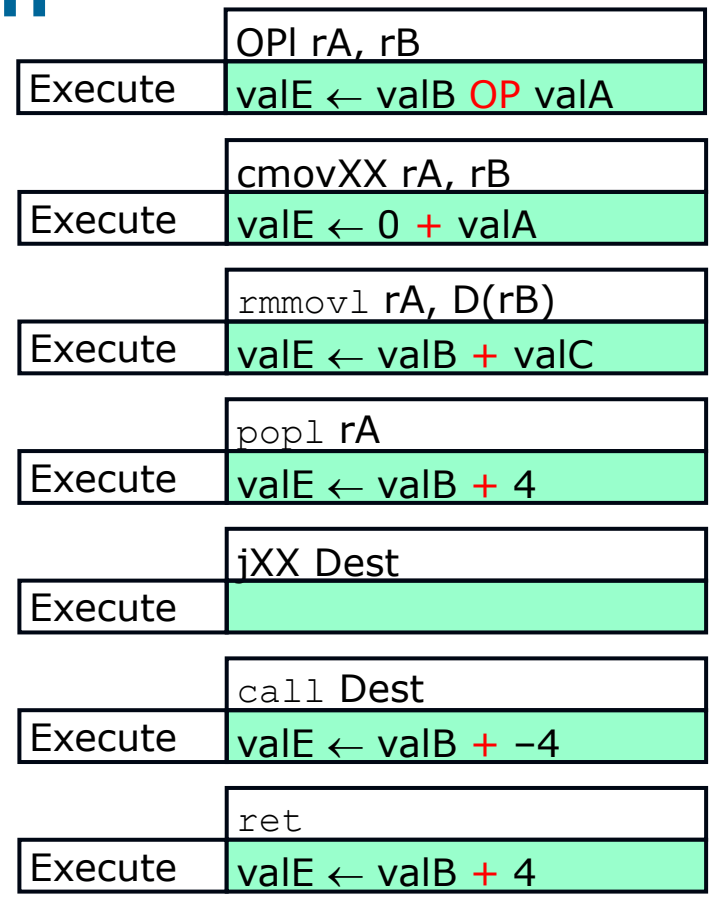
- Control Logic
  - Set CC: Should condition code register be loaded?
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?

ALU A Input




```
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];
```

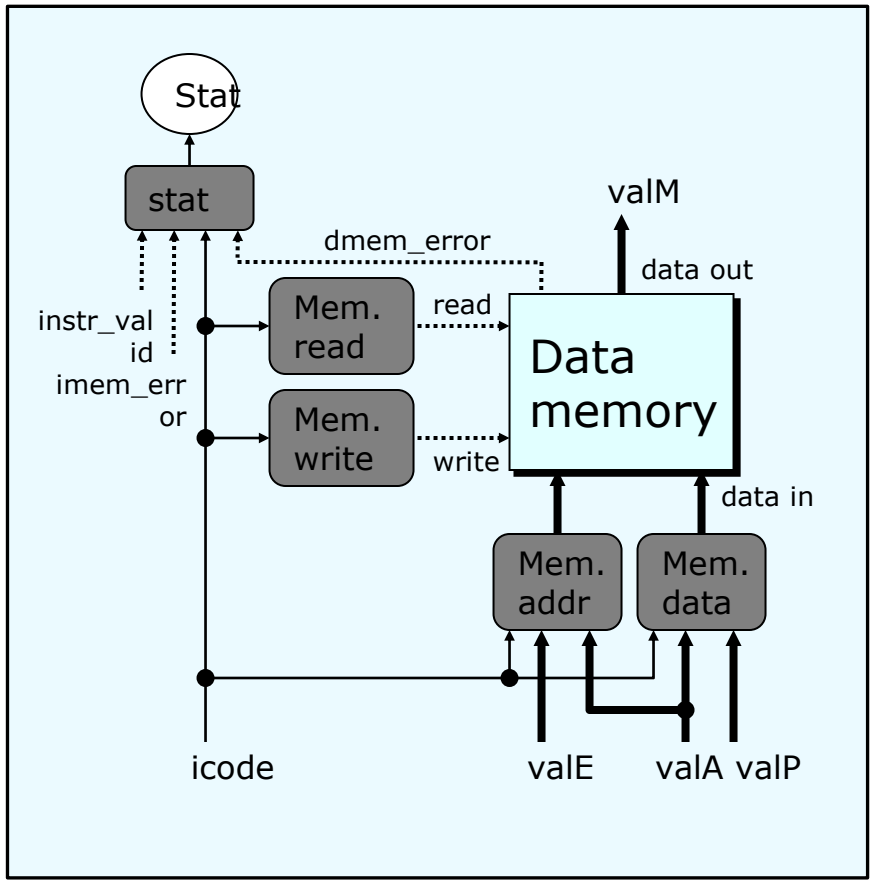
ALU Operation




```
int alufun = [
        icode == IOPL : ifun;
        1 : ALUADD;
];
```

Memory Logic

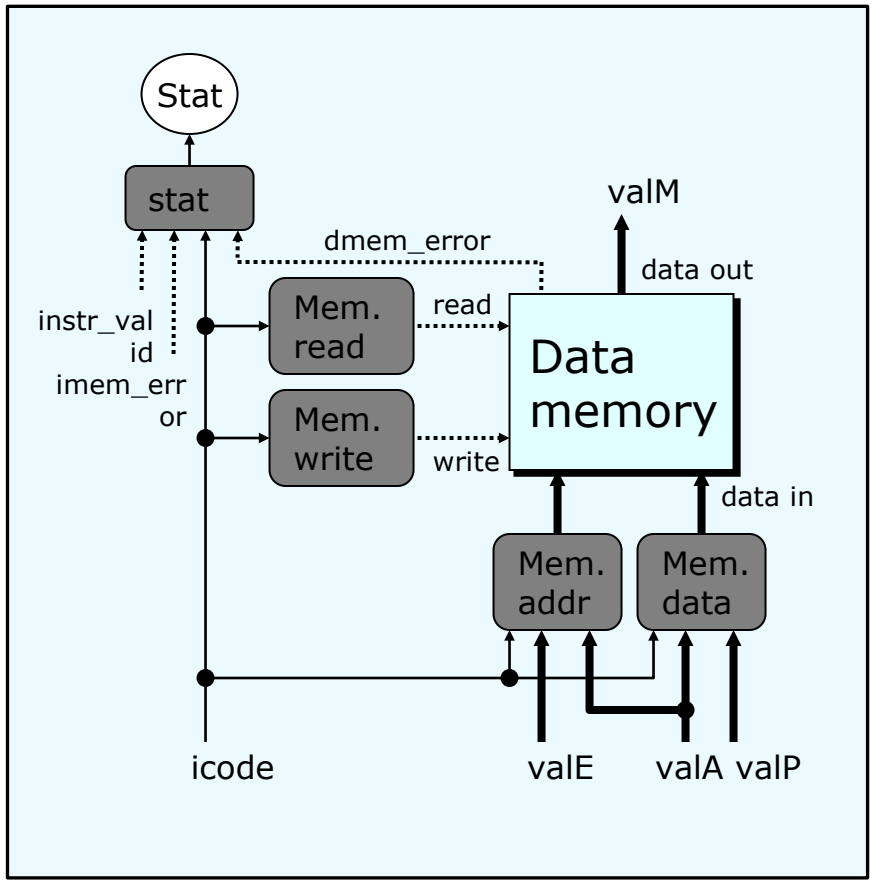
- Memory
  - Reads or writes memory word
- Control Logic
  - stat: What is instruction status?
  - Mem. read: should word be read?
  - Mem. write: should word be written?
  - Mem. addr.: Select address
  - Mem. data.: Select data



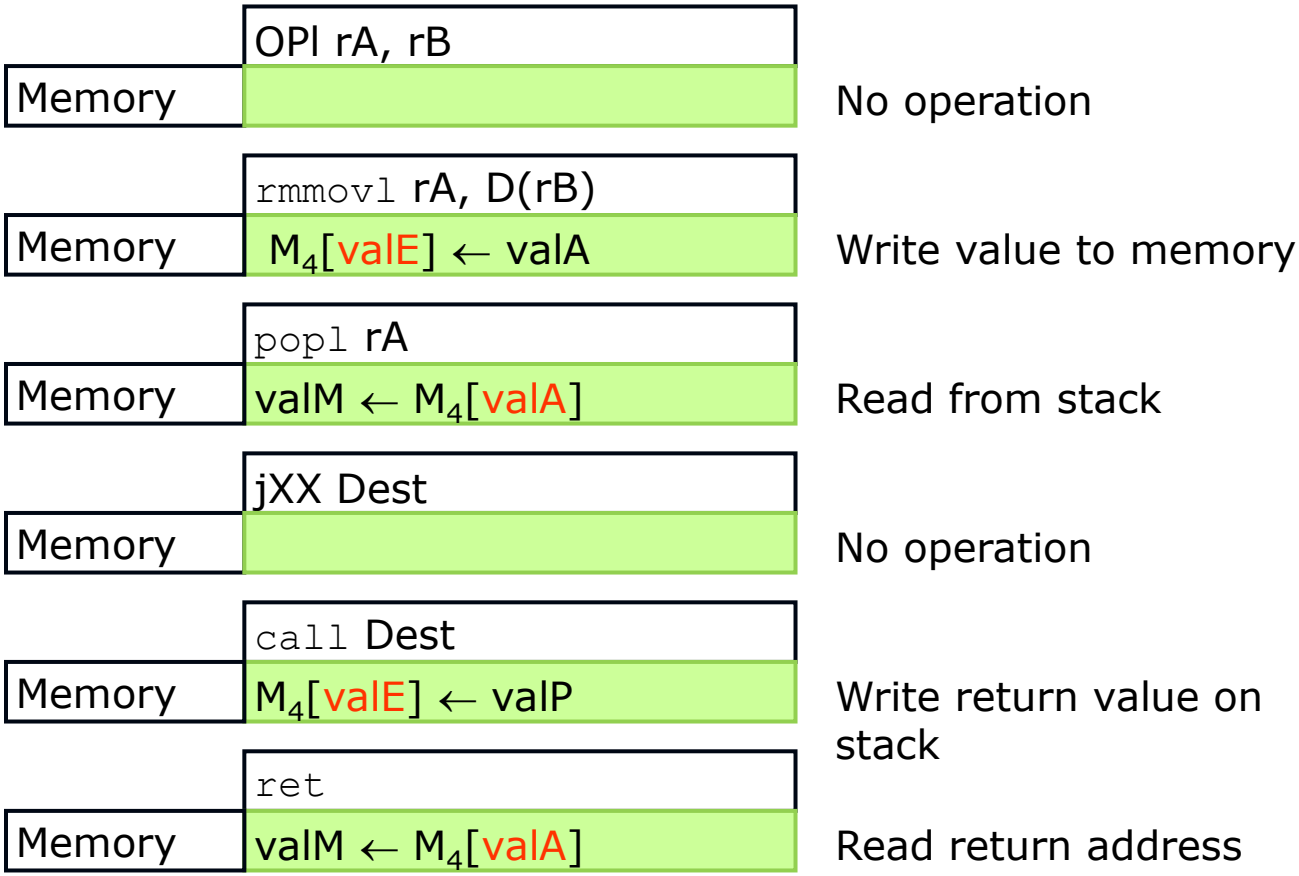
Instruction Status

- Control Logic
  - stat: What is instruction status?

```
## Determine instruction
status
int Stat = [
        imem_error ||
         dmem_error : SADR;
        !instr_valid: SINS;
        icode == IHALT : SHLT;
        1 : SAOK;
];
```

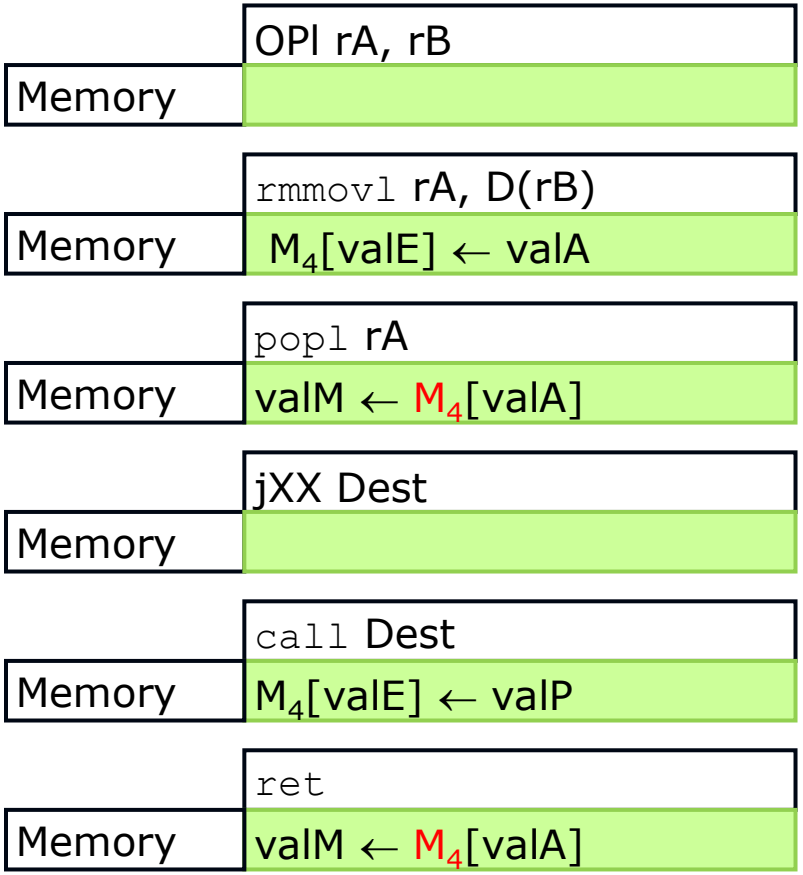


Memory Address




```
int mem_addr = [
        icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
        icode in { IPOPL, IRET } : valA;
        # Other instructions don't need address
];
```

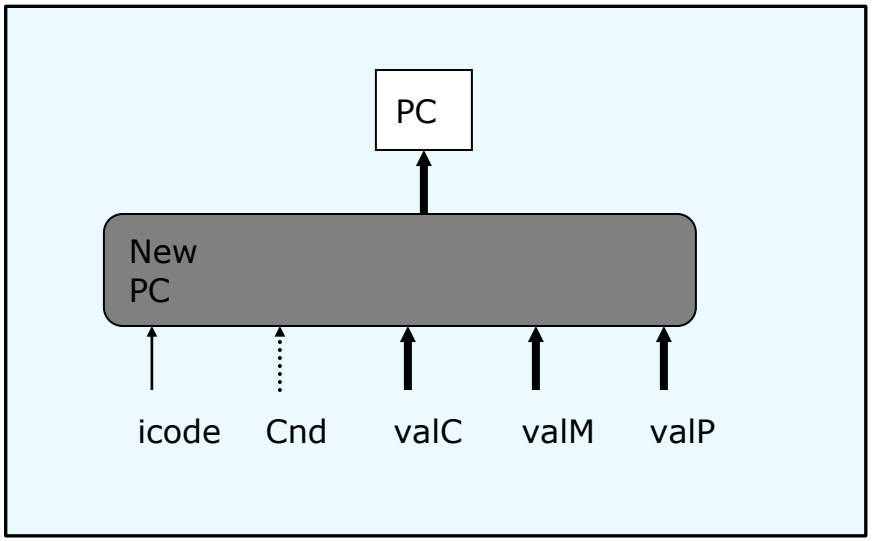
Memory Read



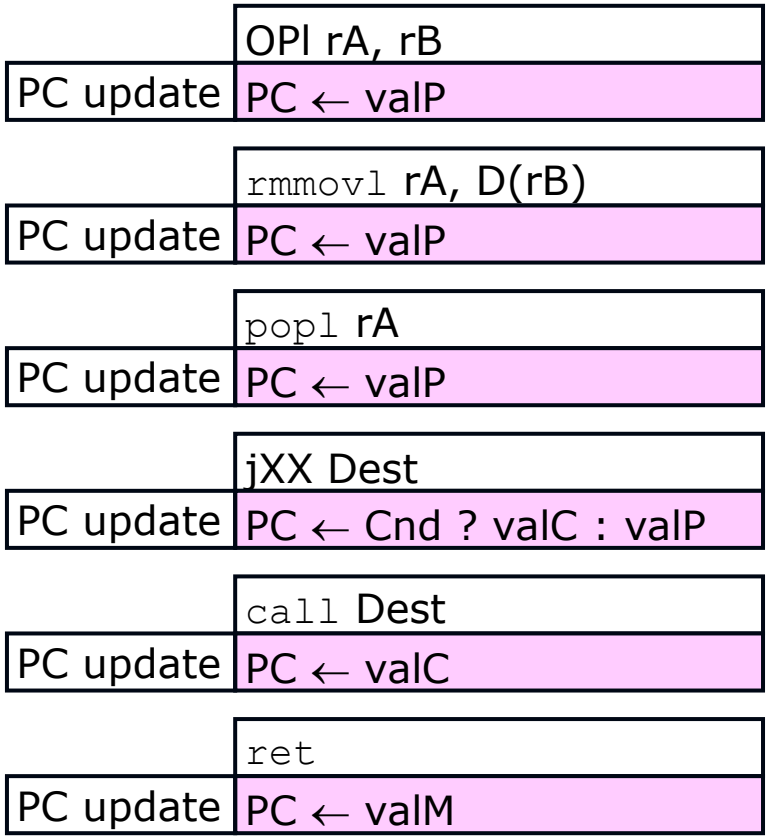

```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

PC Update Logic

- New PC
  - Select next value of PC

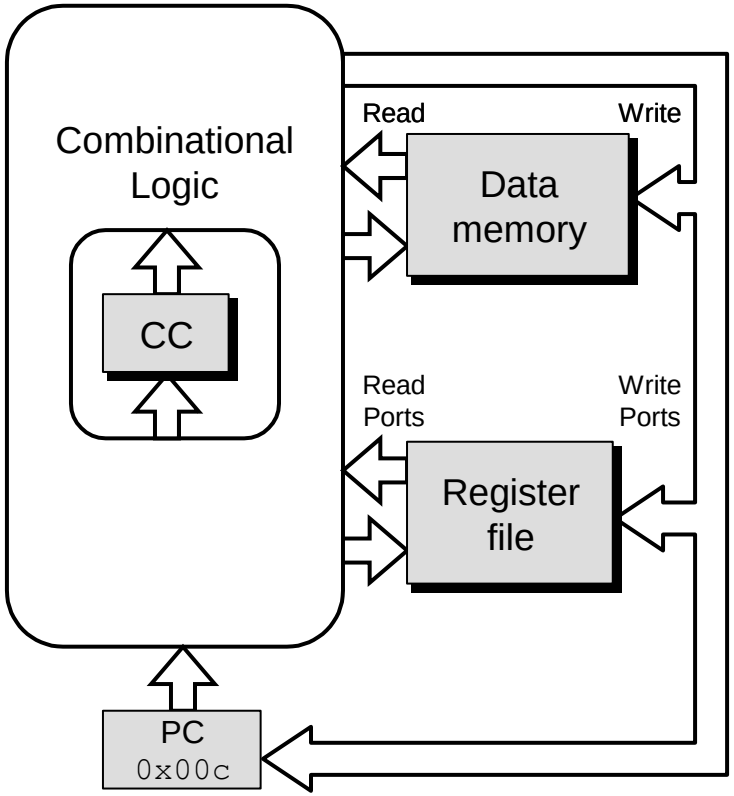


PC Update




```
int new_pc = [
        icode == ICALL : valC;
        icode == IJXX && Cnd : valC;
        icode == IRET : valM;
        1 : valP;
];
```

SEQ Operation

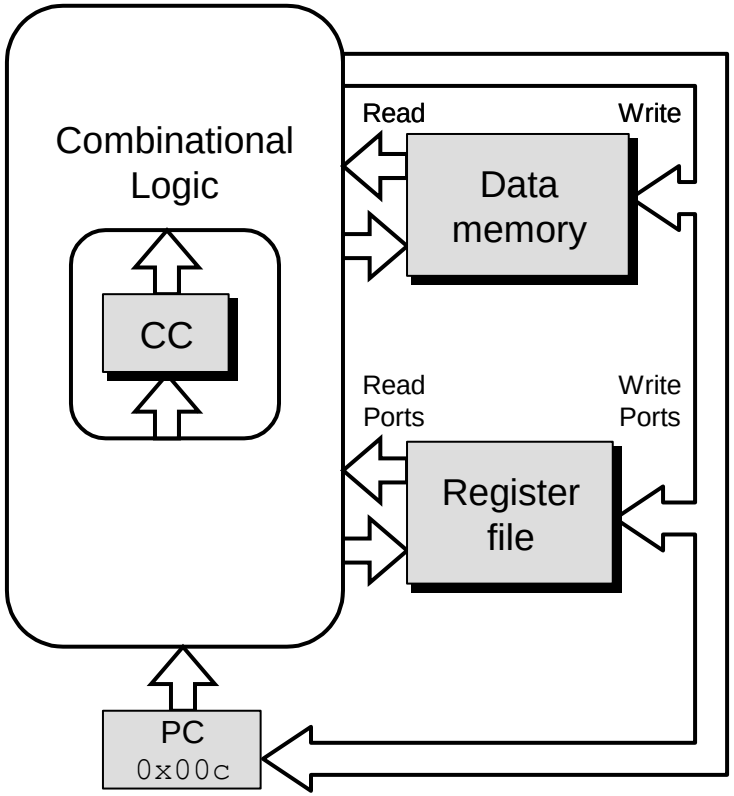


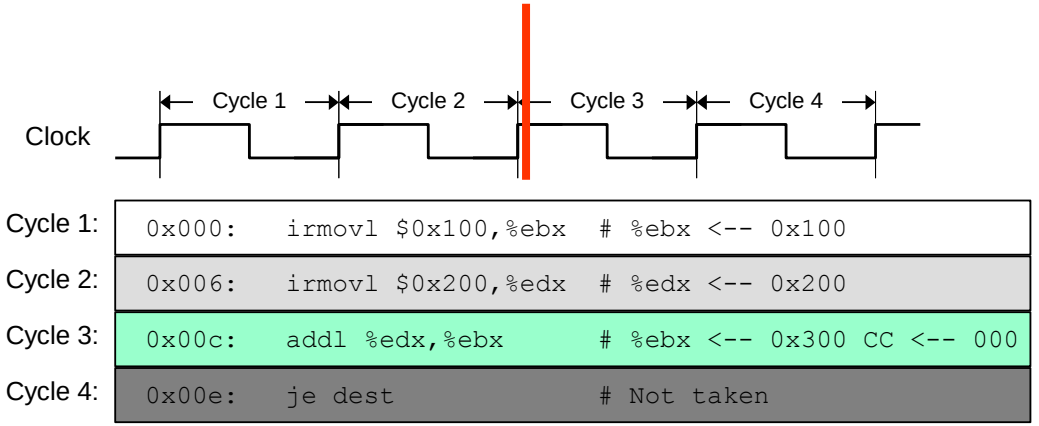
- State
  - PC register
  - Cond. Code register
  - Data memory
  - Register file

  All updated as clock rises

- Combinational Logic
  - ALU
  - Control logic
  - Memory reads
    - Instruction memory
    - Register file
    - Data memory

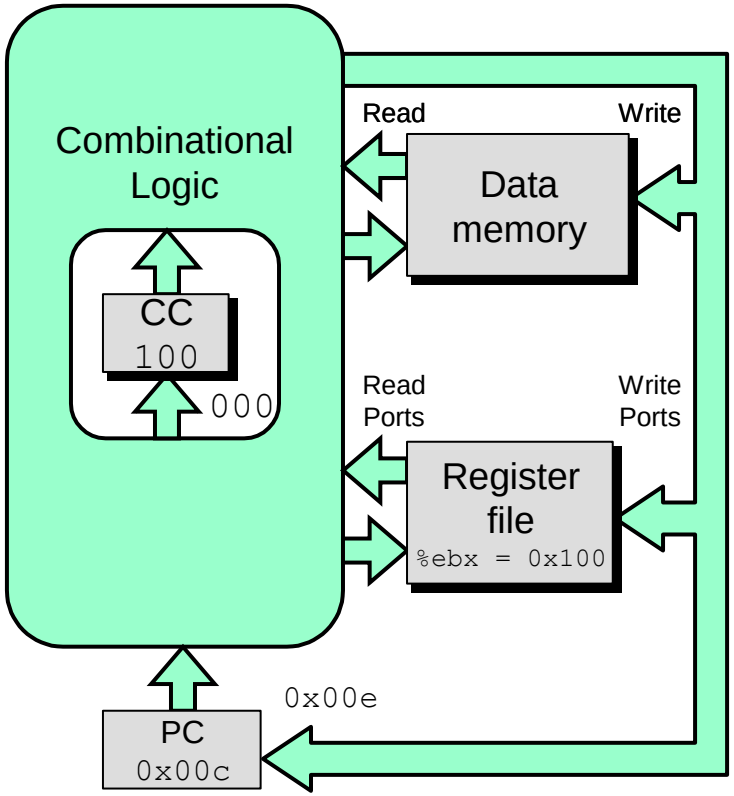
SEQ Operation #2

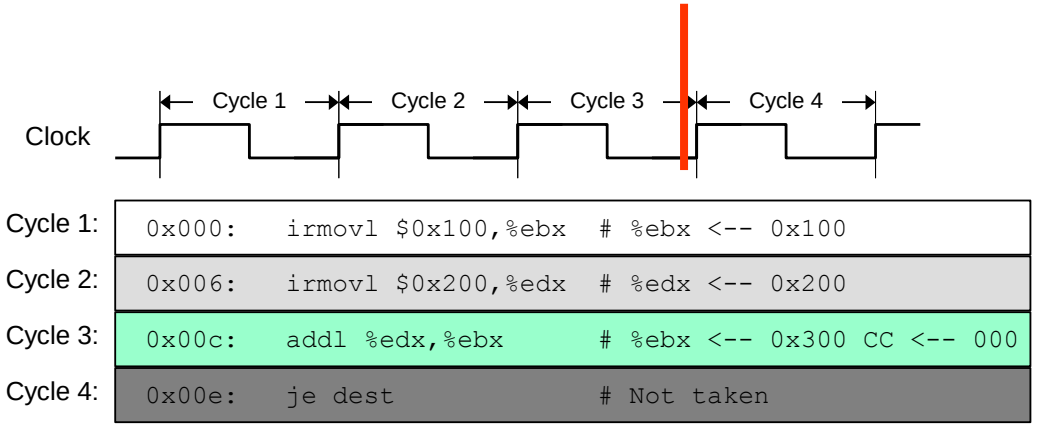




- combinational logic starting to react to state changes

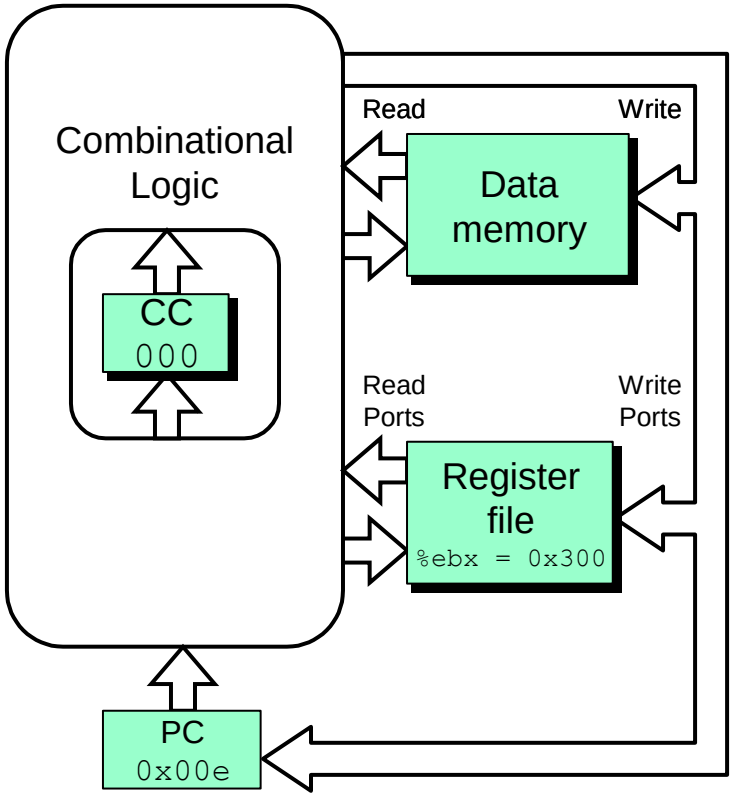
SEQ Operation #3

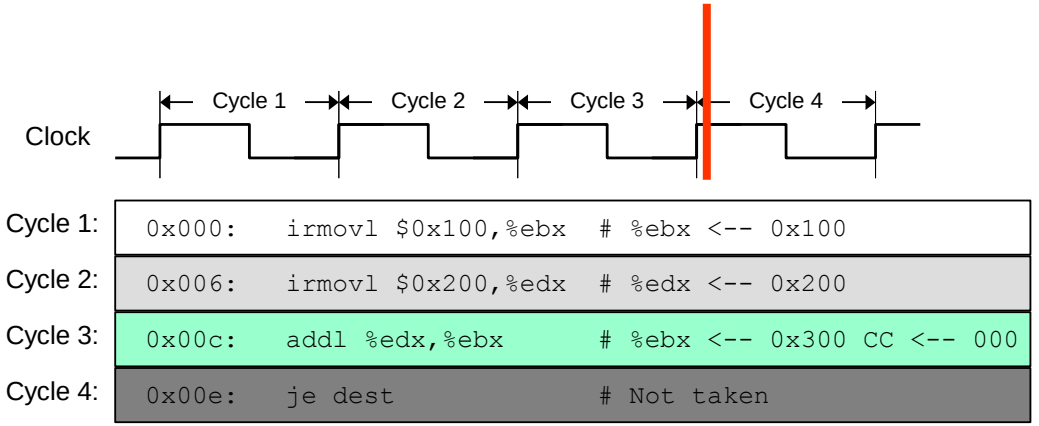




- combinational logic generates results for `addl` instruction

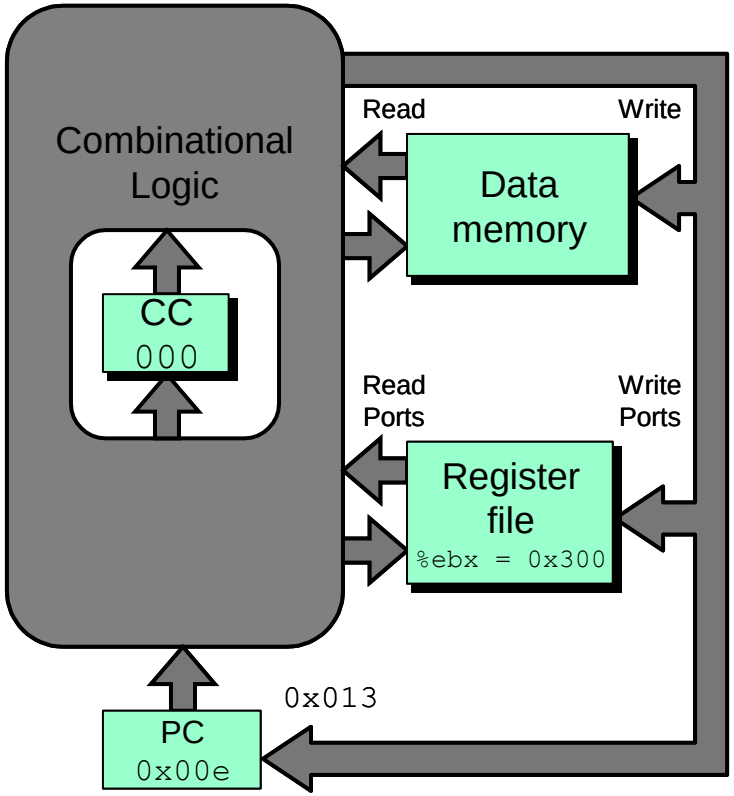
SEQ Operation #4

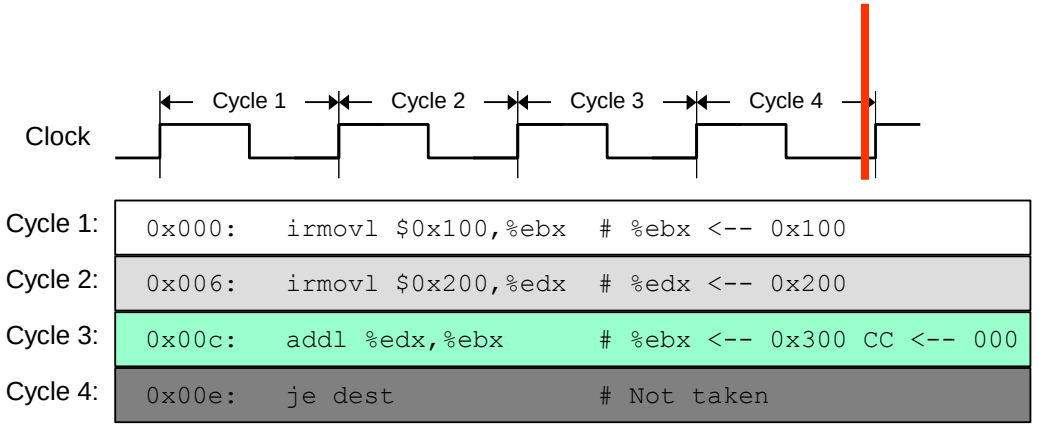




- state set according to `addl` instruction
- combinational logic starting to react to state changes

SEQ Operation #5





- combinational logic generates results for `je` instruction

SEQ Summary

- Implementation
  - Express every instruction as a series of simple steps
  - Follow the same general flow for each instruction type
  - Assemble registers, memories, predesigned combinational blocks
  - Connect with control logic

- Limitations
  - Too slow to be practical
  - In one cycle, must propagate through instruction memory, register file, ALU, and data memory
  - Would need to run clock very slowly
  - Hardware units only active for fraction of clock cycle