Sequential Y86-64 wrap up

CSCI 237: Computer Organization
18th Lecture, Mar 21, 2025

Jeannie Albrecht

Administrative Details

- Lab 3
  - Due today-ish
- Please don’t discuss the exam (yet)
- After break, all labs are C programs

Last time

- Sequential Y86-64 implementations (Ch 4.3)
  - Organizing Processes into Stages
  - SEQ Hardware Structure
  - SEQ Stage Implementations
  - SEQ Timing

Recap: Stage Computation: popq

(FYI this was actually correct!)

	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

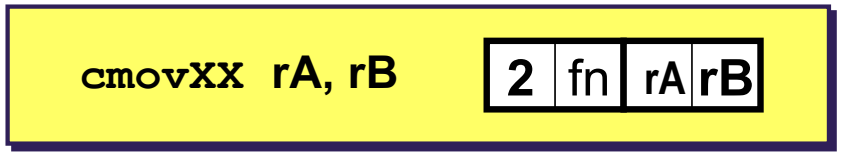
(Move M_8[R[%rsp]] to R[rA])

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

Today

- Sequential Y86-64 implementations (Ch 4.3)
  - Organizing Processes into Stages wrap up
  - SEQ Hardware Structure
  - SEQ Stage Implementations
  - SEQ Timing

Executing Conditional Moves



rrmovq	2	0
cmovle	2	1
cmovl	2	2
cmove	2	3
cmovne	2	4
cmovge	2	5
cmovg	2	6

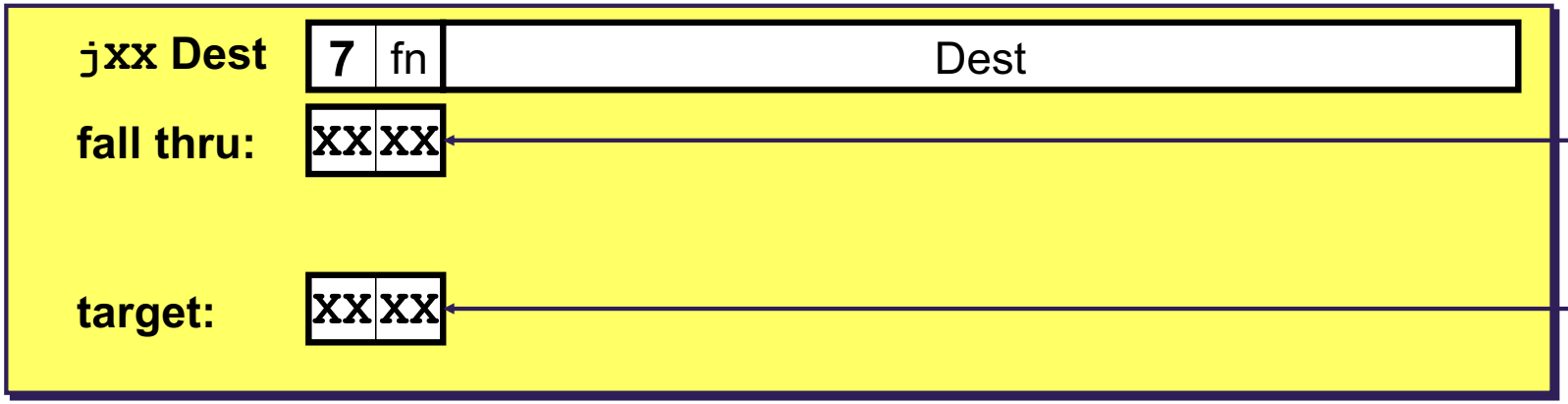
- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - If !cond, then set destination register to 0xF
- Memory
  - Do nothing
- Write back
  - Update register (or not)
- PC Update
  - Increment PC by 2

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 if !Cond)
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



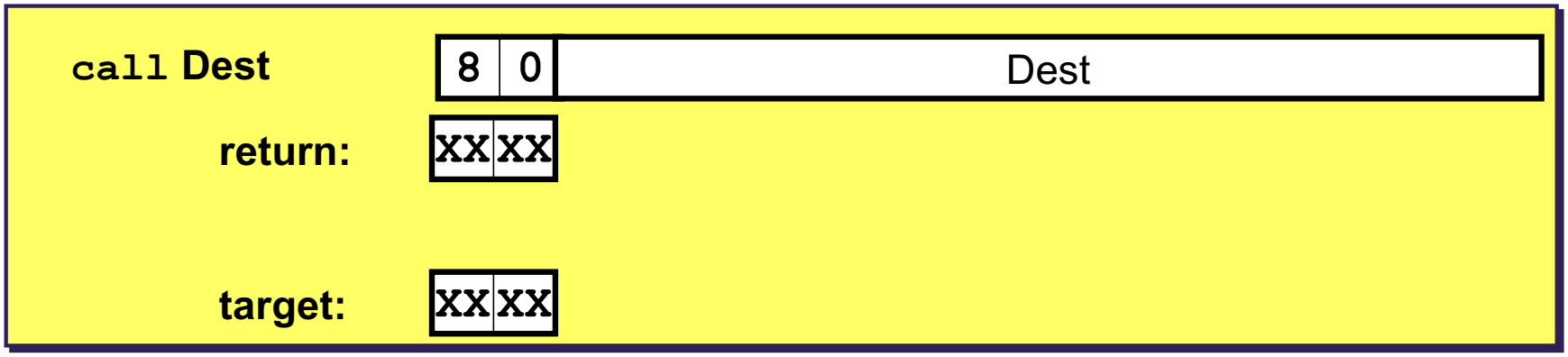
- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

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`



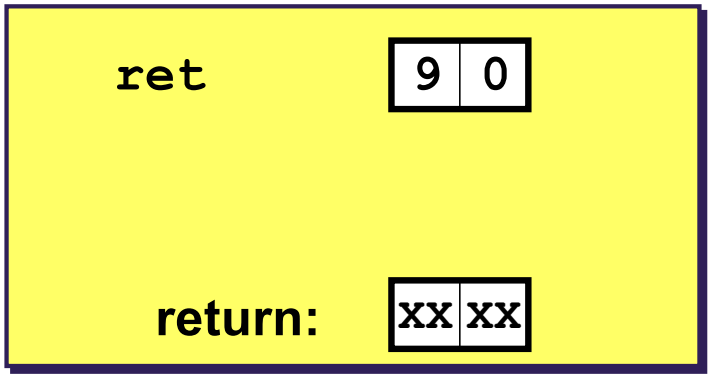
- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 8
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

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 stack pointer
- Execute
  - Increment stack pointer by 8
- Memory
  - Read return address from 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[%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

Summary: 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

Summary: 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

Summary of Computed Values

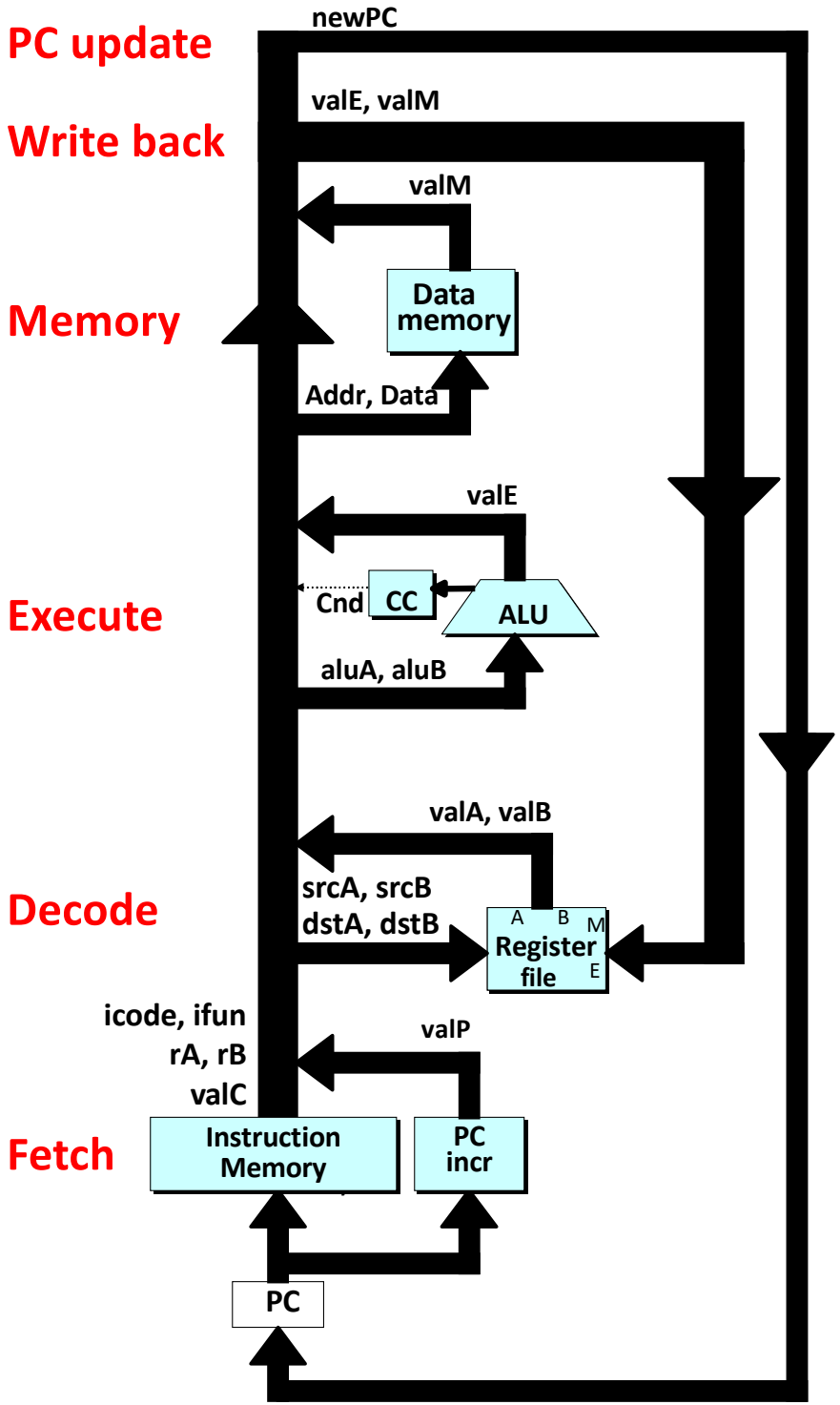
- Fetch
  - icode — Instruction code
  - ifun — Instruction function
  - rA — Instr. Register A
  - rB — Instr. Register B
  - valC — Instruction constant
  - valP — Incremented PC
- Decode
  - srcA — Register ID A
  - srcB — Register ID B
  - dstE — Destination Register E
  - dstM — Destination Register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

Today

- Sequential Y86-64 implementations (Ch 4.3)
  - Organizing Processes into Stages wrapup
  - SEQ Hardware Structure
  - SEQ Stage Implementations
  - SEQ Timing

Recap: SEQ Stages

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



SEQ Hardware

- Key
  - Blue boxes:

    predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes:

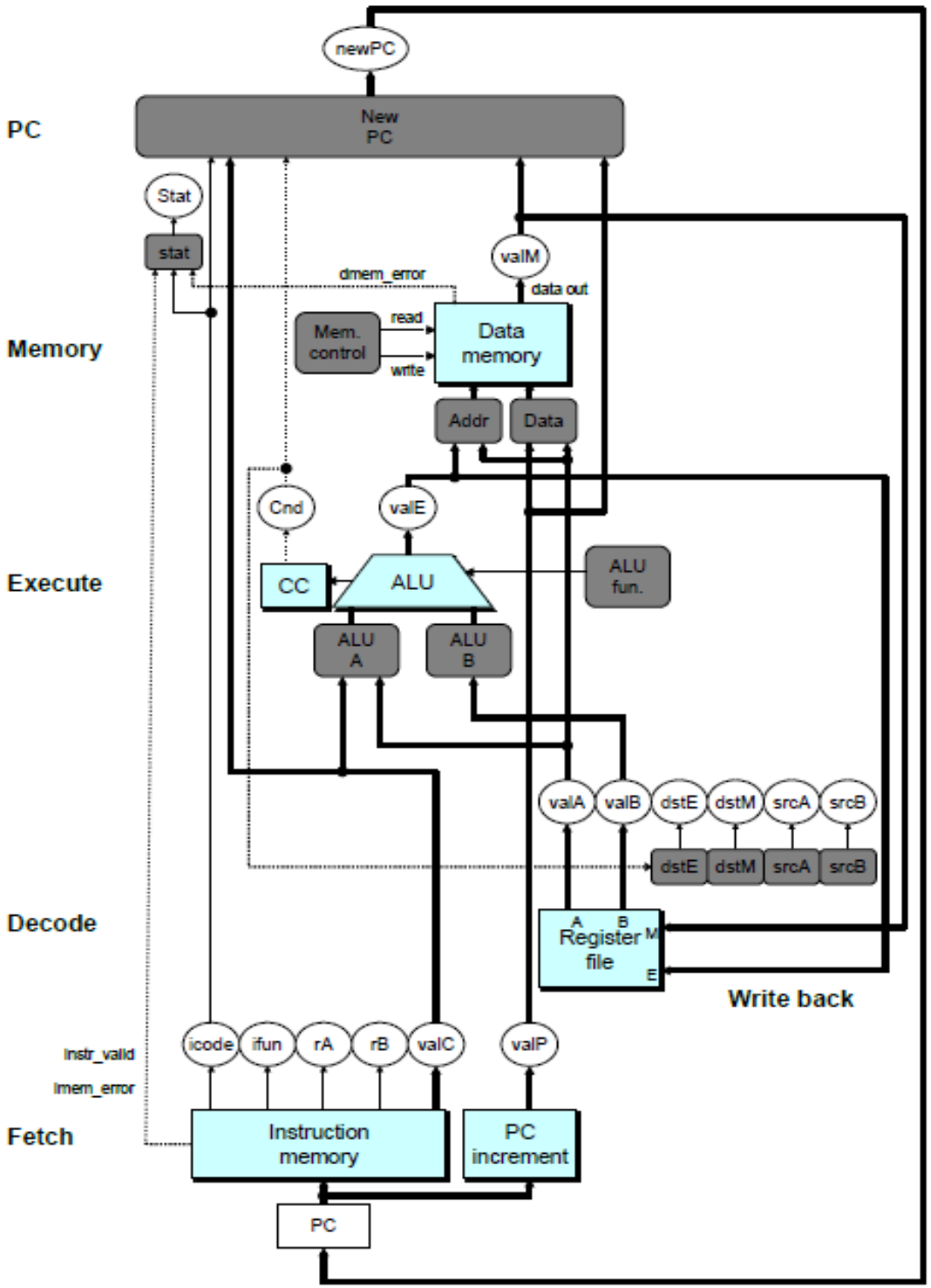
    control logic
    - Describe in HCL
  - White ovals:

    labels for signals
  - **Thick lines:**

    64-bit word values
  - Thin lines:

    4-8 bit values
  - Dotted lines:

    1-bit values



SEQ Hardware

- Key
  - Blue boxes:

    predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes:

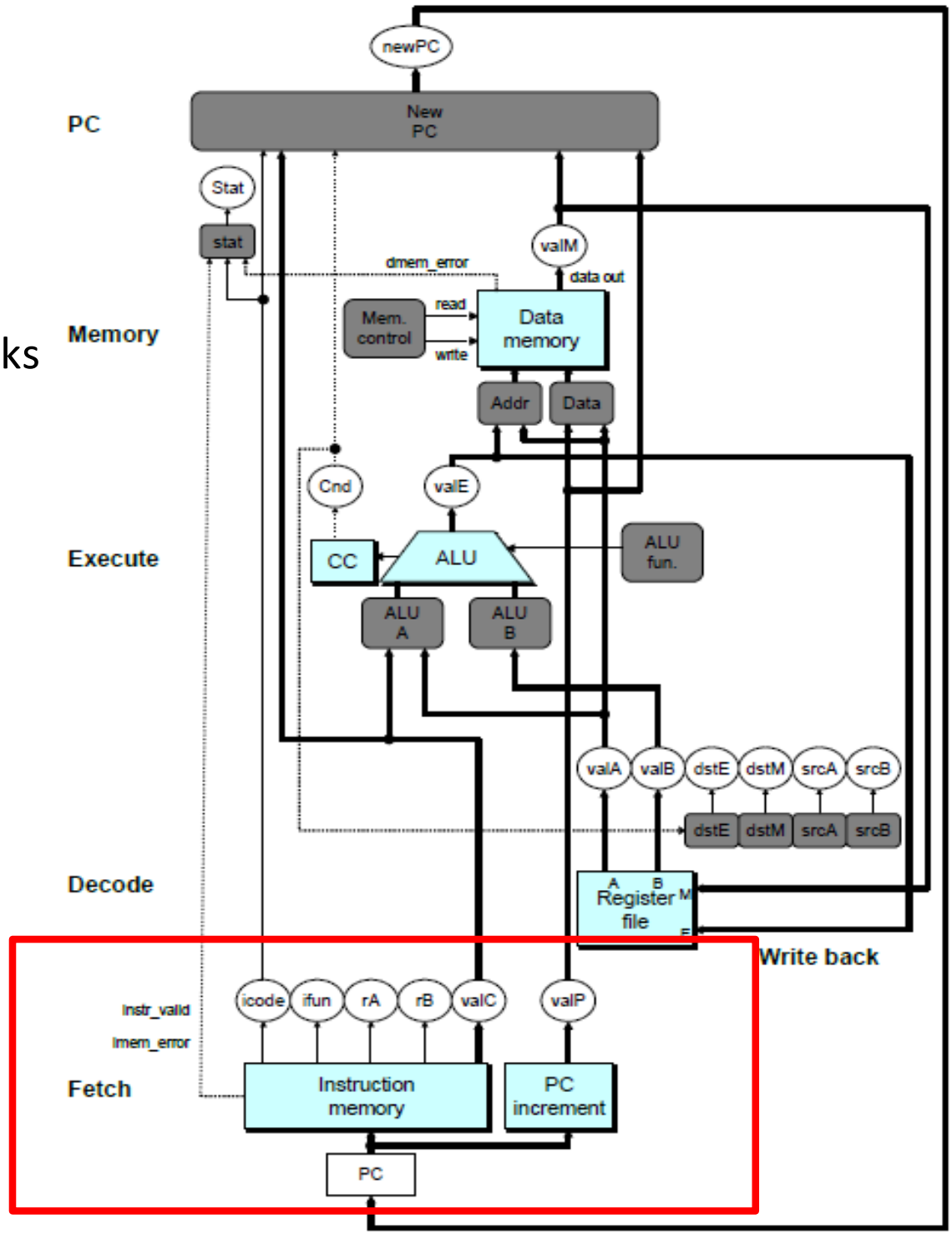
    control logic
    - Describe in HCL
  - White ovals:

    labels for signals
  - Thick lines:

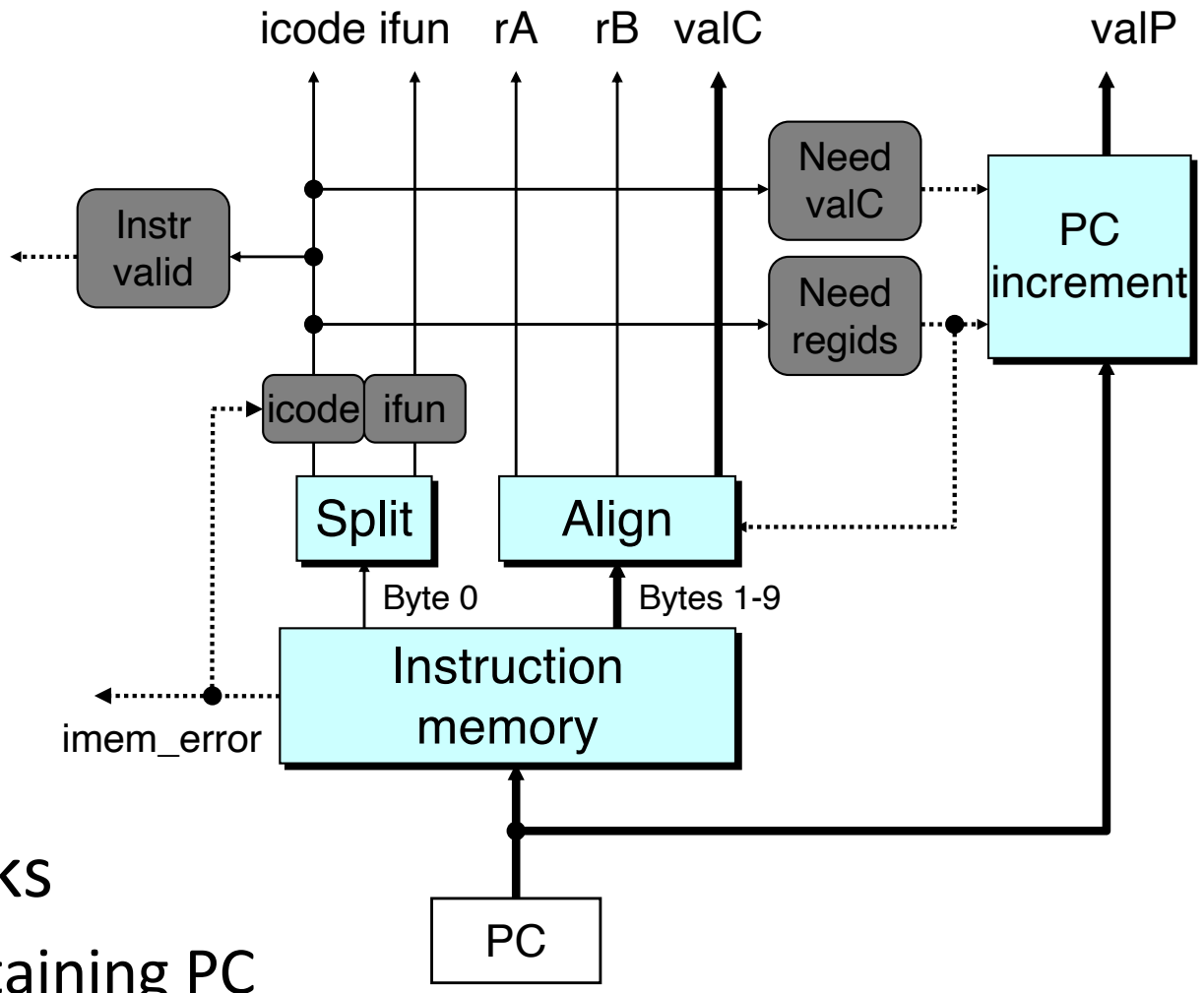
    64-bit word values
  - Thin lines:

    4-8 bit values
  - Dotted lines:

    1-bit values



Fetch Logic

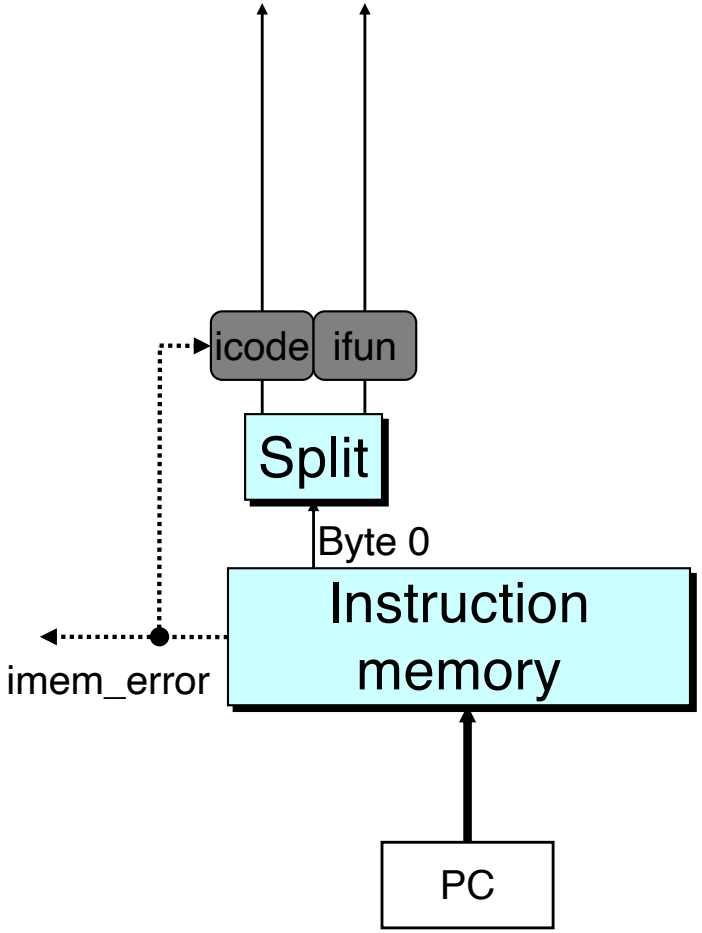


- Predefined Blocks
  - PC: Register containing PC
  - Instruction memory: Read (up to) 10 bytes (PC to PC+9)
    - Signal invalid address
  - Split: Divide instruction byte into icode and ifun
  - Align: Get fields for rA, rB, and valC

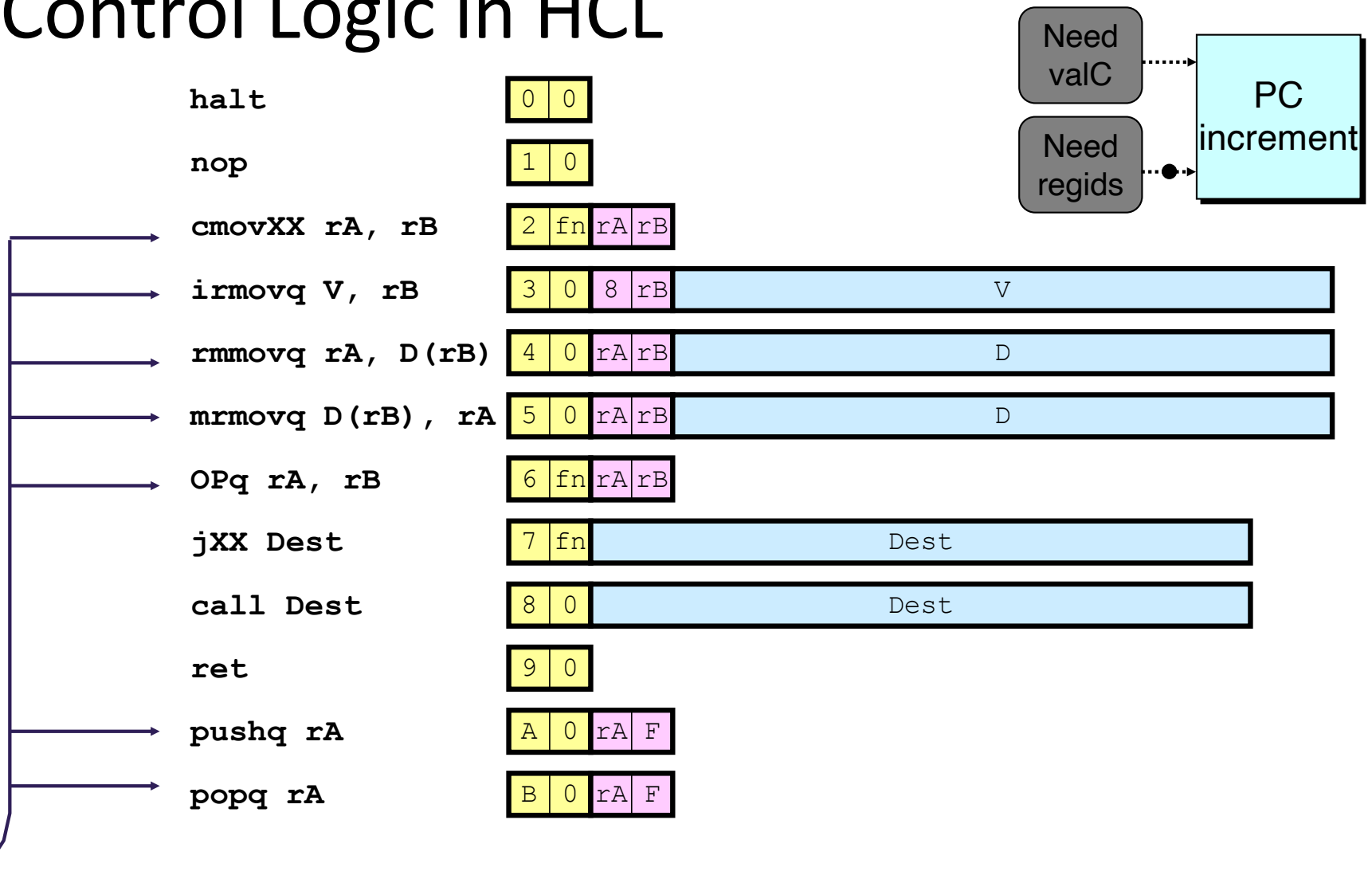
Fetch Control Logic in HCL (gray boxes)

```
# 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 { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                   IIRMOVQ, IRMMOVQ, IMRMOVQ };
bool instr_valid = icode in
        { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
                IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };
```

SEQ Hardware

- Key
  - Blue boxes:

    predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes:

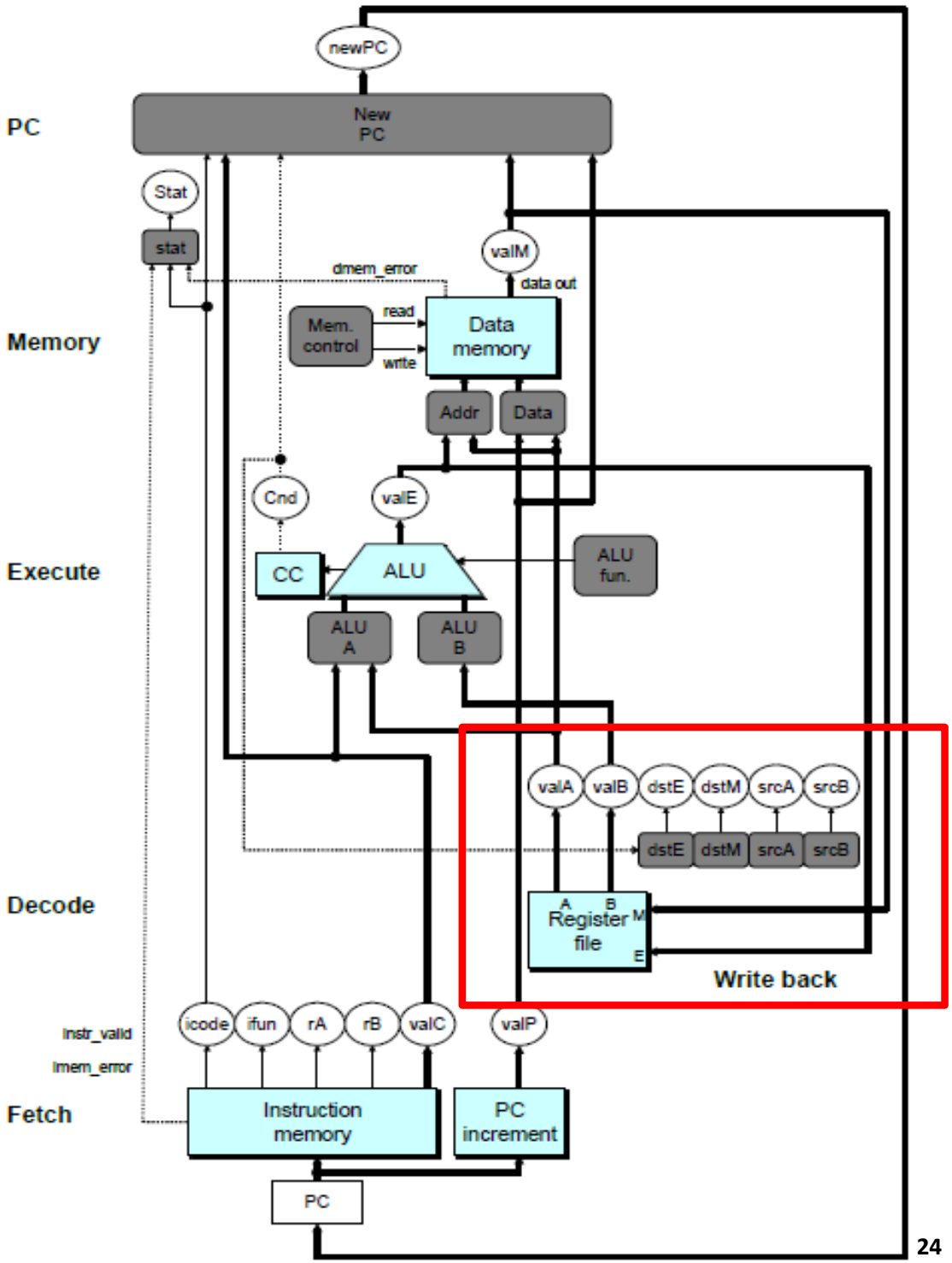
    control logic
    - Describe in HCL
  - White ovals:

    labels for signals
  - Thick lines:

    64-bit word values
  - Thin lines:

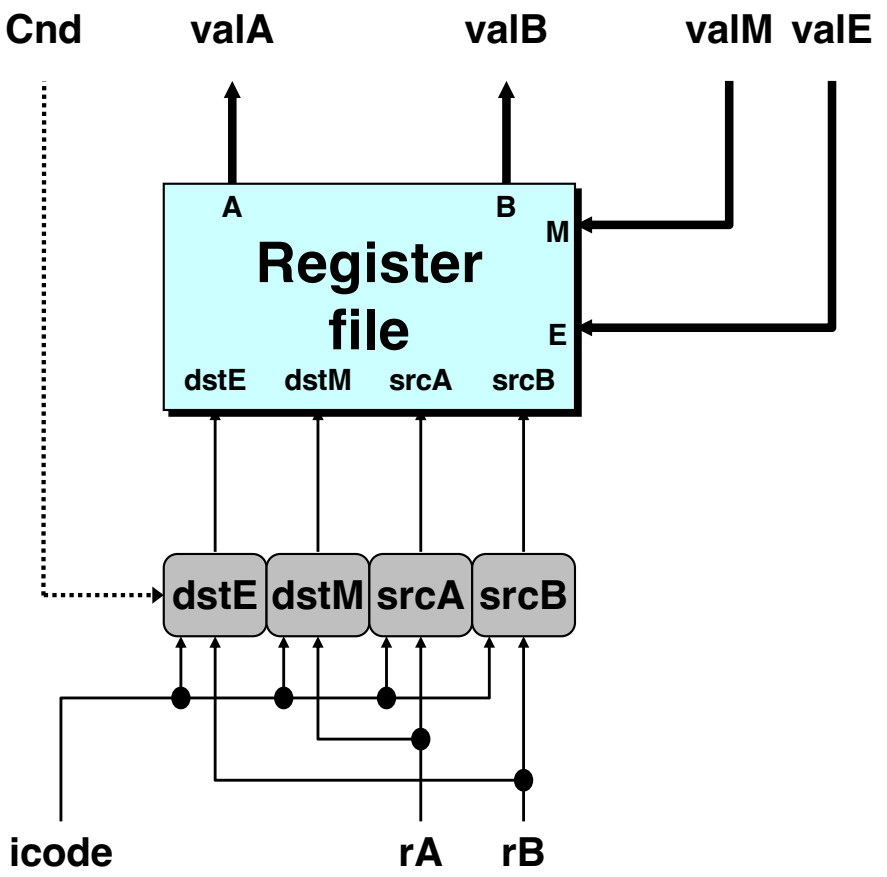
    4-8 bit values
  - Dotted lines:

    1-bit values



Decode/Write-back 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



Source A

	OPq rA, rB	
Decode	valA ← R[rA]	Read operand A
	cmovXX rA, rB	
Decode	valA ← R[rA]	Read operand A
	`rmmovq` rA, D(rB)	
Decode	valA ← R[rA]	Read operand A
	`popq` rA	
Decode	valA ← R[`%rsp`]	Read stack pointer
	jXX Dest	
Decode		No operand
	`call` Dest	
Decode		No operand
	`ret`	
Decode	valA ← R[`%rsp`]	Read stack pointer

```
int srcA = [
        icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
        icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];
```

Dest E

	OPq rA, rB	
Write-back	R[rB] ← valE	Write back result
	cmovXX rA, rB	
Write-back	R[rB] ← valE	Conditionally write back result
	`rmmovq` rA, D(rB)	
Write-back		None
	`popq` rA	
Write-back	R[`%rsp`] ← valE	Update stack pointer
	jXX Dest	
Write-back		None
	`call` Dest	
Write-back	R[`%rsp`] ← valE	Update stack pointer
	`ret`	
Write-back	R[`%rsp`] ← valE	Update stack pointer

```
int dstE = [
        icode in { IRRMOVQ } && Cnd : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```

SEQ Hardware

- Key
  - Blue boxes:

    predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes:

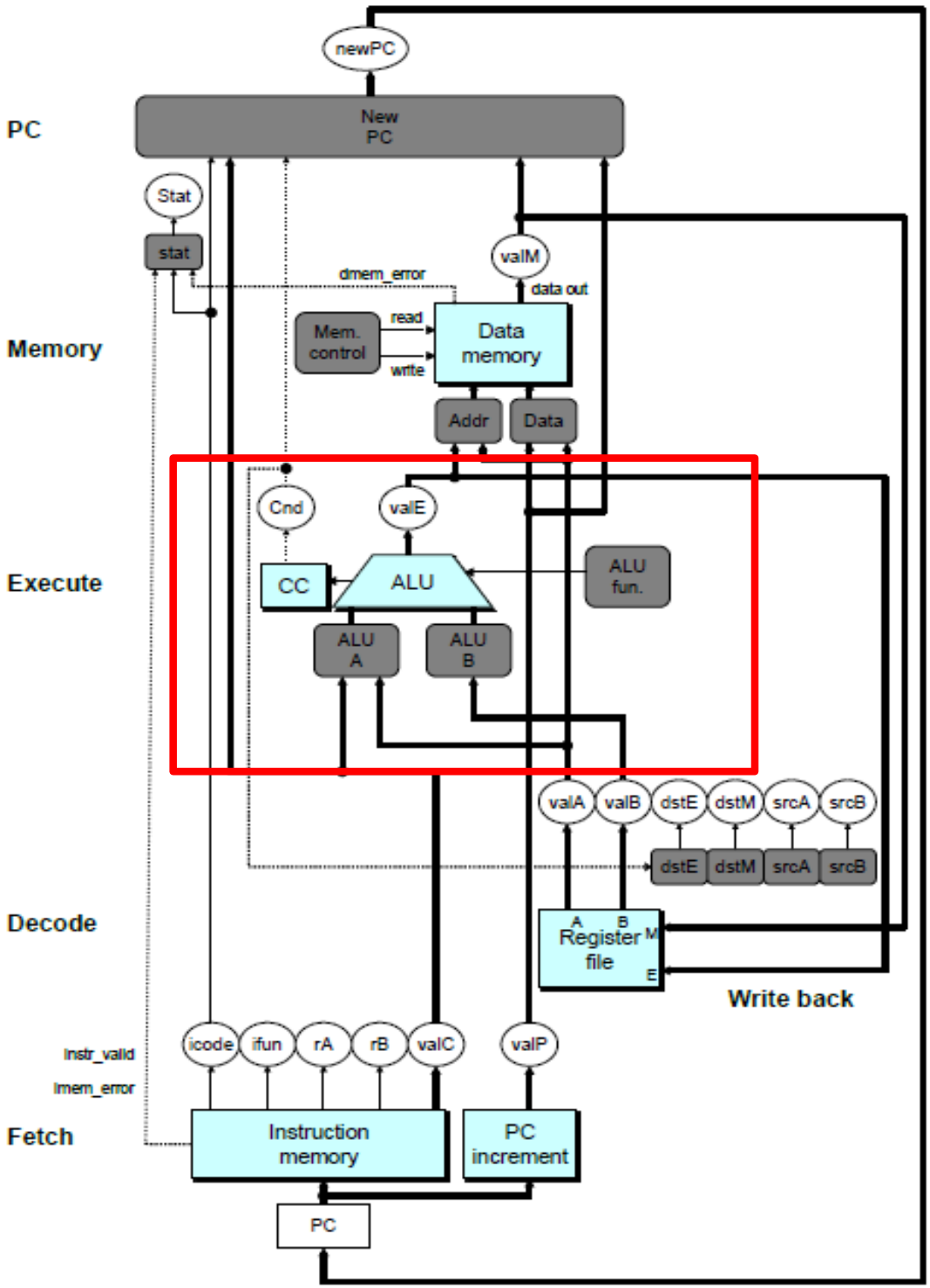
    control logic
    - Describe in HCL
  - White ovals:

    labels for signals
  - Thick lines:

    64-bit word values
  - Thin lines:

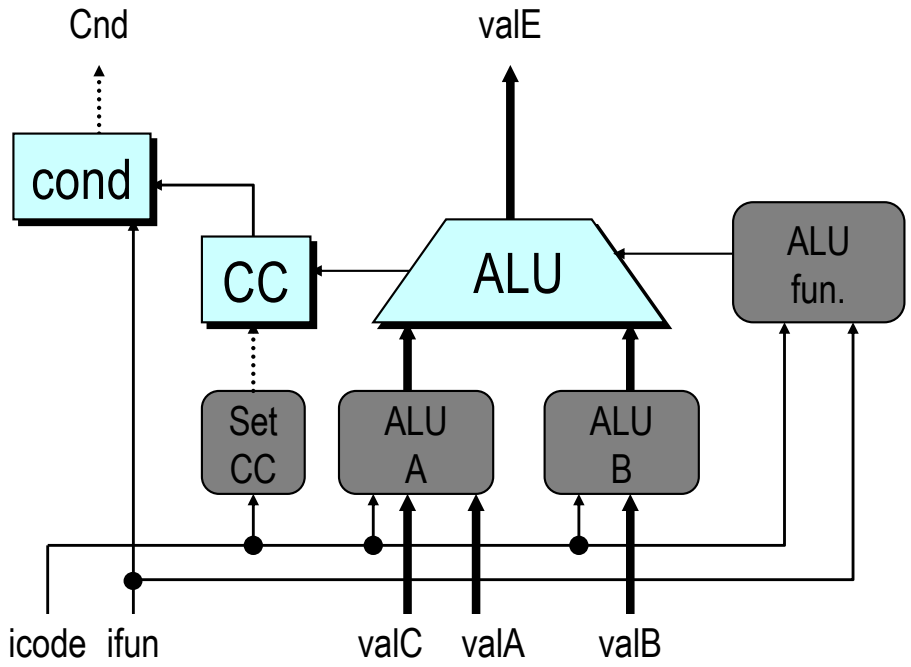
    4-8 bit values
  - Dotted lines:

    1-bit values

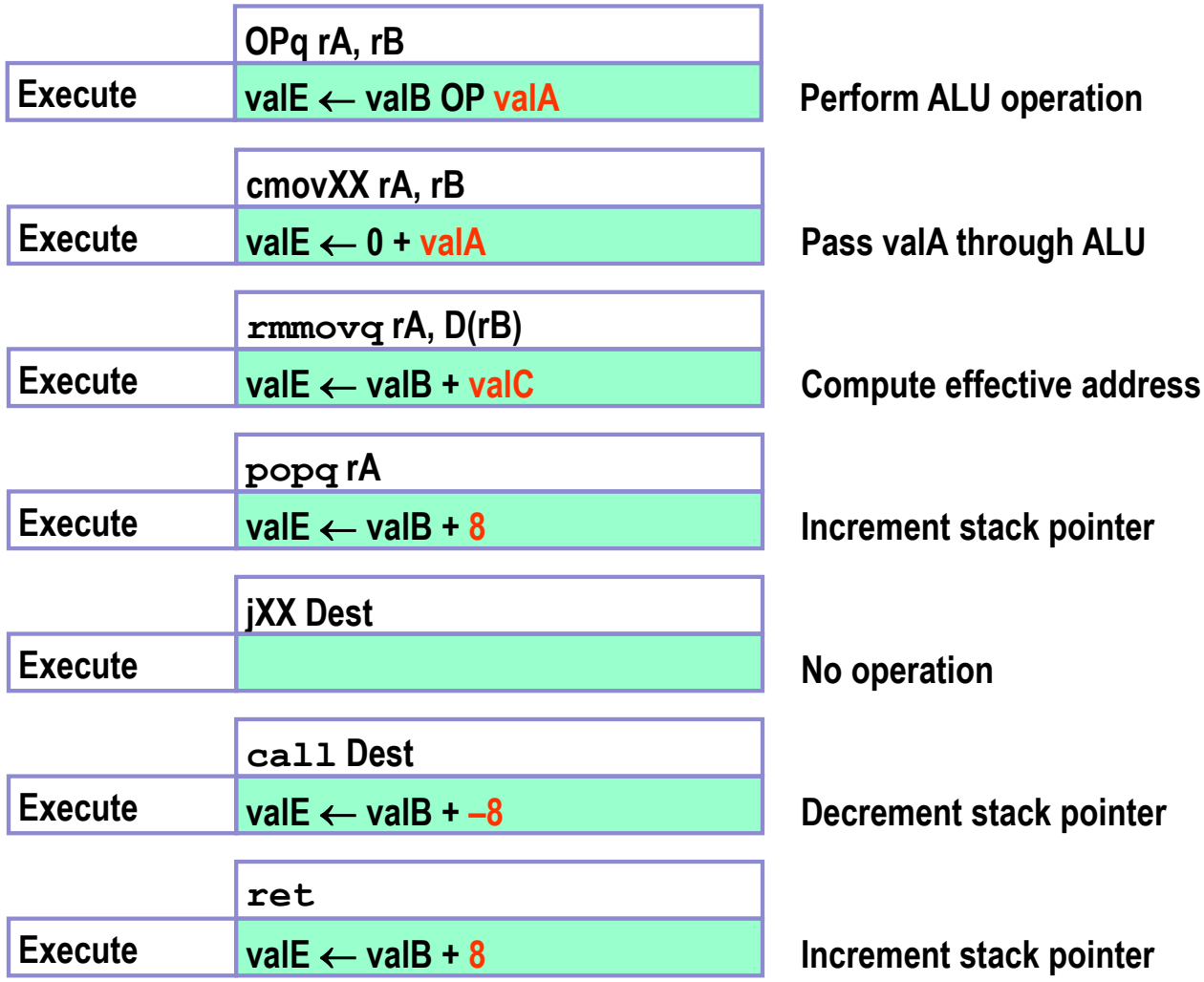


Execute Logic

- 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




```
int aluA = [
        icode in { IRRMOVQ, IOPQ } : valA;
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
        icode in { ICALL, IPUSHQ } : -8;
        icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];
```

ALU Op

	OPl rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	cmovXX rA, rB	
Execute	valE ← 0 + valA	Pass valA through ALU
	rmmovl rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	popq rA	
Execute	valE ← valB + 8	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	valE ← valB + –8	Decrement stack pointer
	ret	
Execute	valE ← valB + 8	Increment stack pointer

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

SEQ Hardware

- Key
  - Blue boxes:

    predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes:

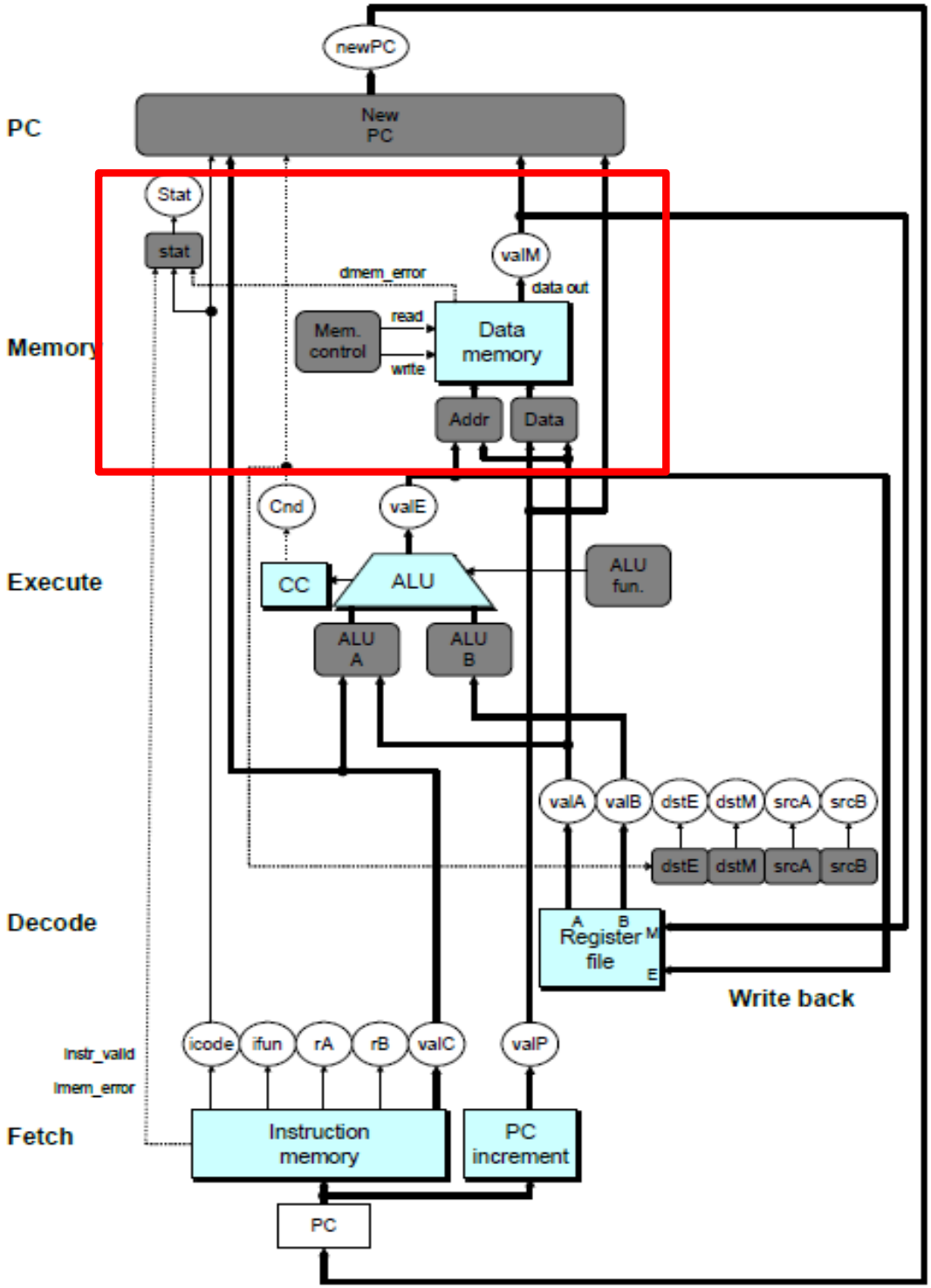
    control logic
    - Describe in HCL
  - White ovals:

    labels for signals
  - Thick lines:

    64-bit word values
  - Thin lines:

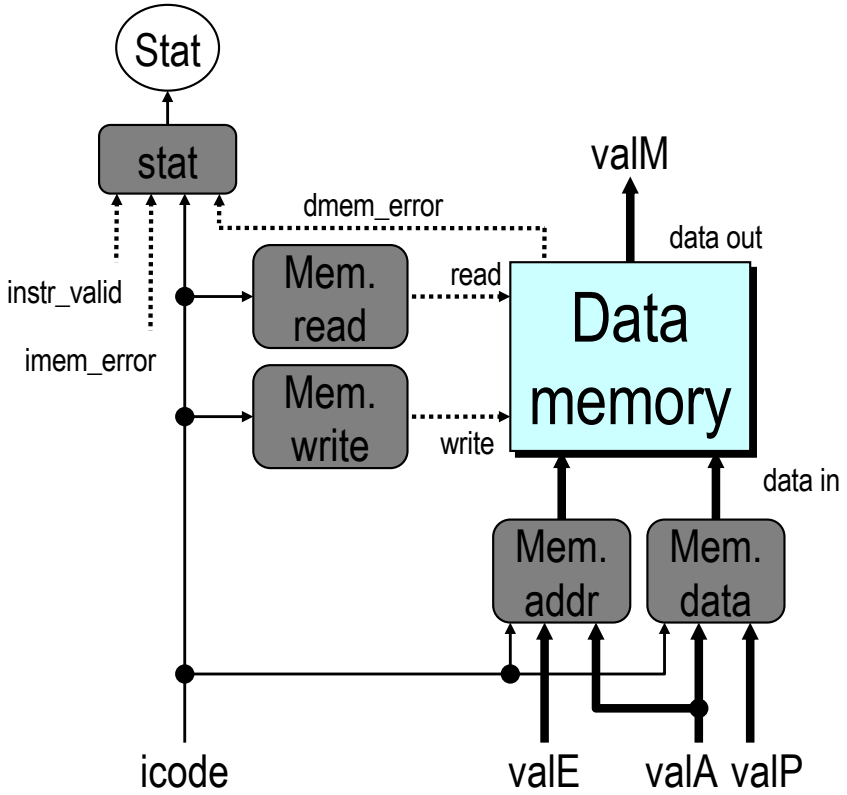
    4-8 bit values
  - Dotted lines:

    1-bit values



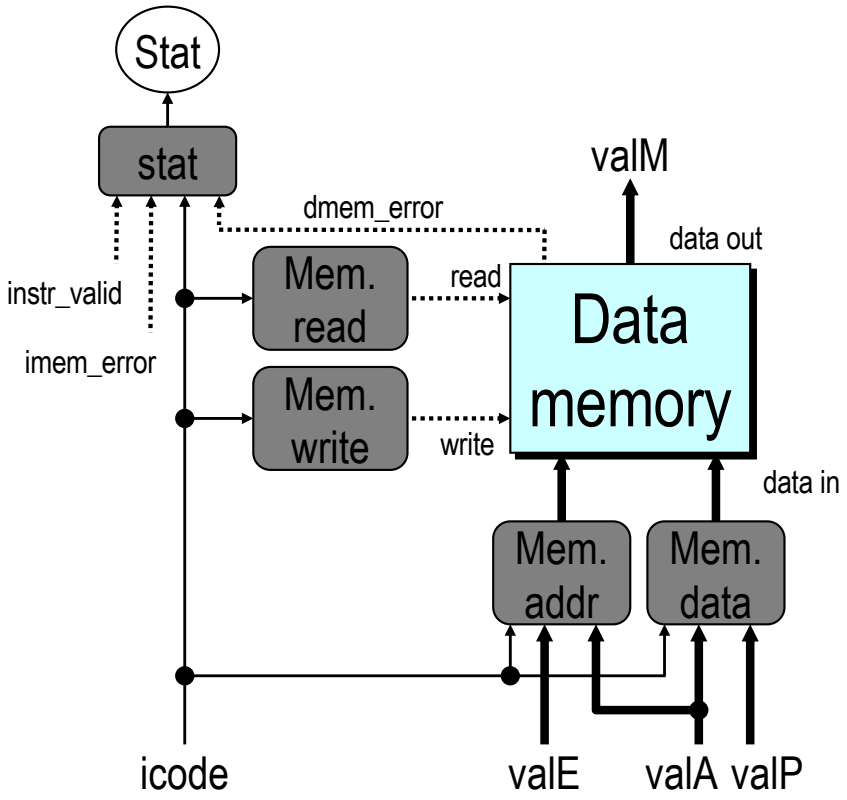
Memory Logic

- 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

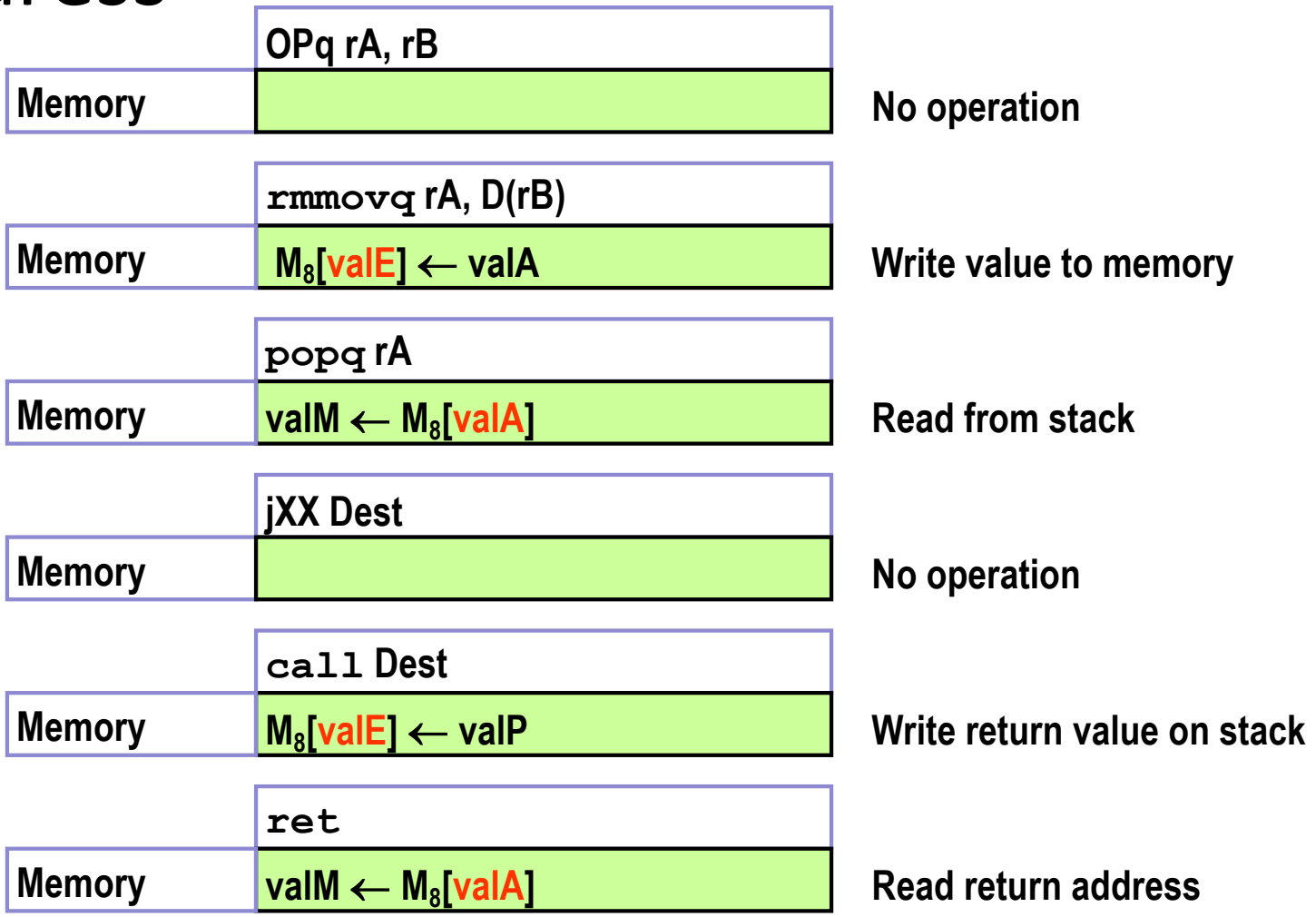
- Control Logic
  - stat: What is instruction status?




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

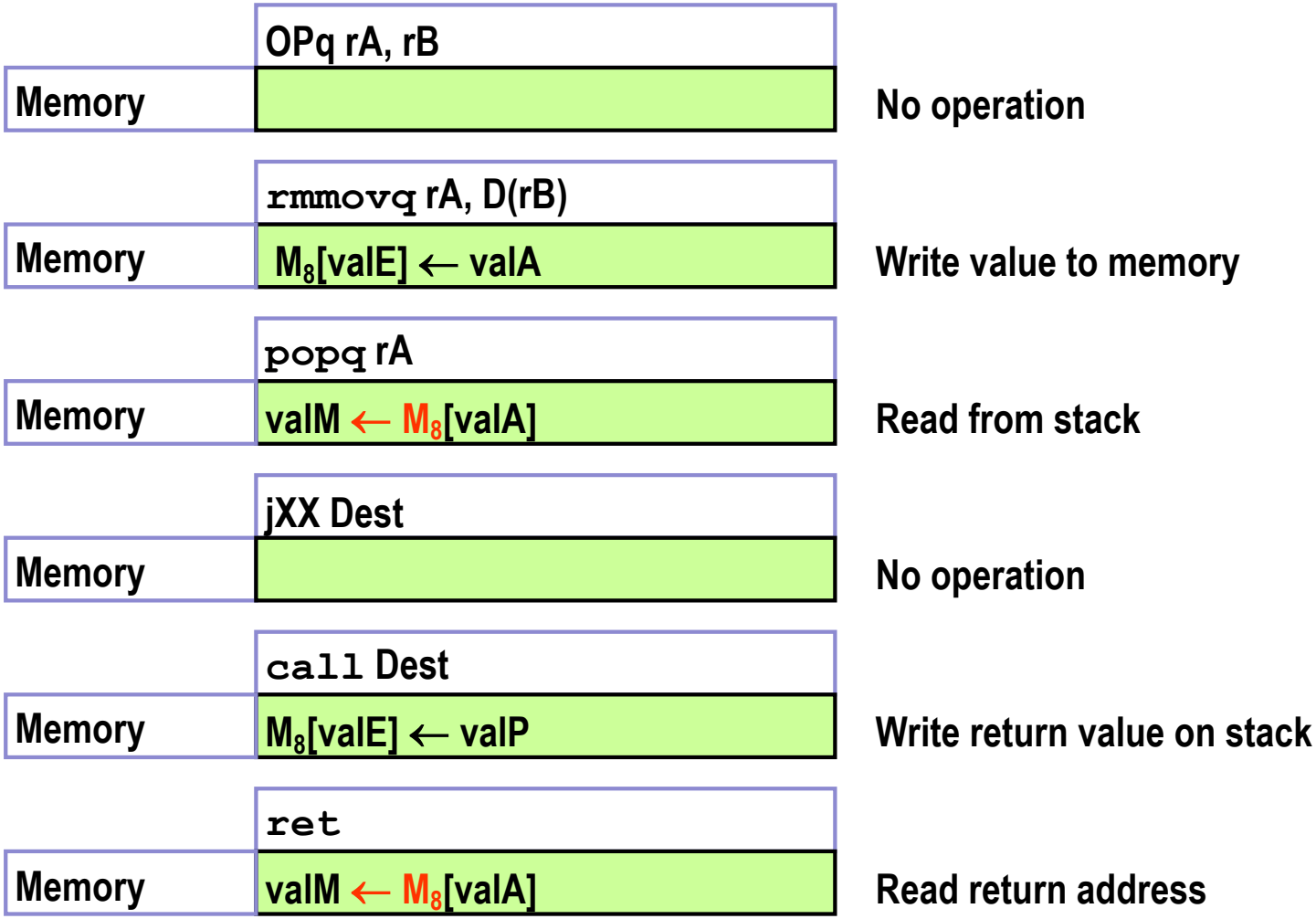
```
SADR = bad address
SINS = invalid instr
SHLT = halt
SAOK = good to go!
```

Memory Address




```
int mem_addr = [
        icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
        icode in { IPOPQ, IRET } : valA;
        # Other instructions don't need address
];
```

Memory Read




```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
```

SEQ Hardware

- Key
  - Blue boxes:

    predesigned hardware blocks
    - E.g., memories, ALU
  - Gray boxes:

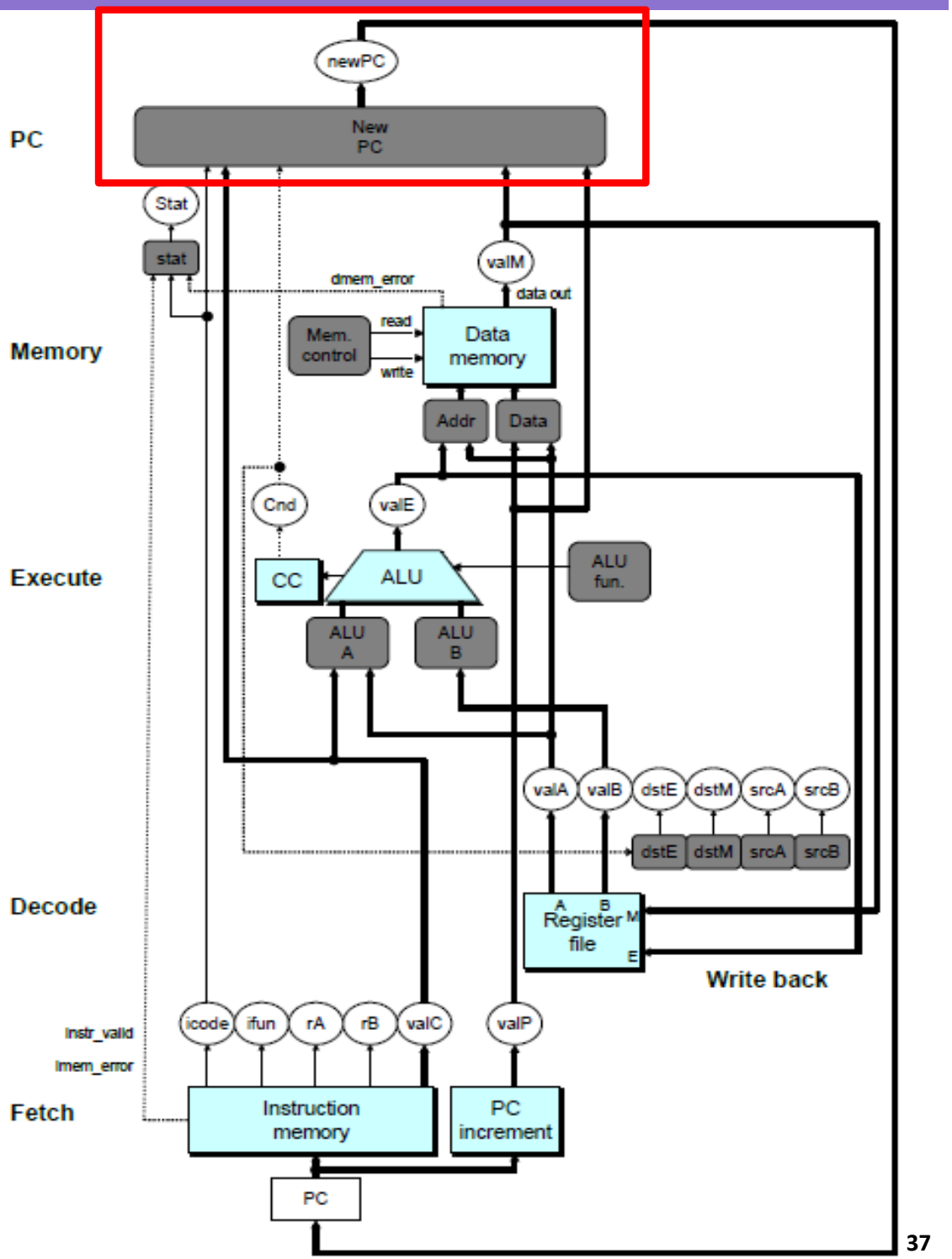
    control logic
    - Describe in HCL
  - White ovals:

    labels for signals
  - Thick lines:

    64-bit word values
  - Thin lines:

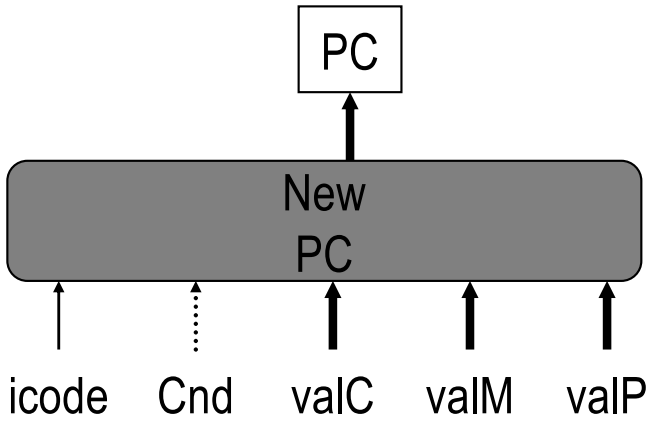
    4-8 bit values
  - Dotted lines:

    1-bit values



PC Update Logic

- New PC
  - Select next value of PC



PC Update

	OPq rA, rB	
PC update	PC ← valP	Update PC
	`rmmovq` rA, D(rB)	
PC update	PC ← valP	Update PC
	`popq` rA	
PC update	PC ← valP	Update PC
	jXX Dest	
PC update	PC ← Cnd ? valC : valP	Update PC
	`call` Dest	
PC update	PC ← valC	Set PC to destination
	`ret`	
PC update	PC ← valM	Set PC to return address

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

HAVE A GREAT SPRING BREAK!