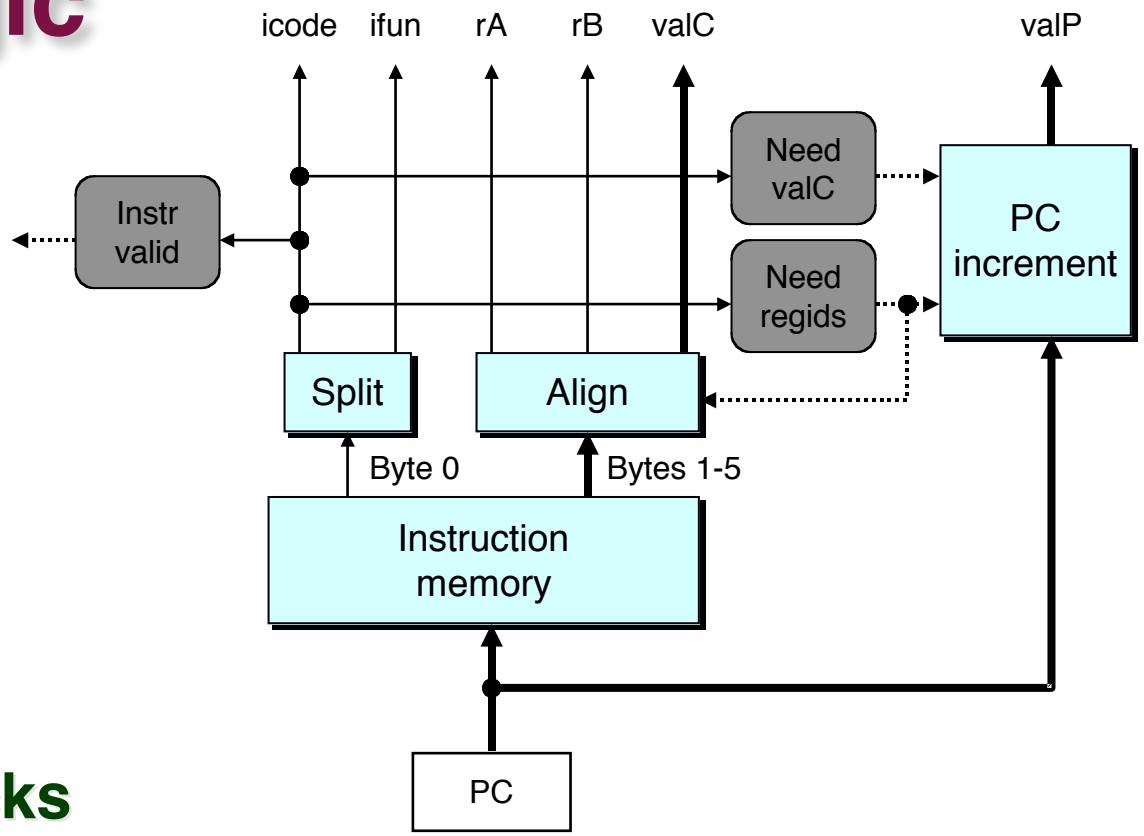
Systems I

Datapath Design III

Topics

- Control logic for instruction execution
- Timing and clocking

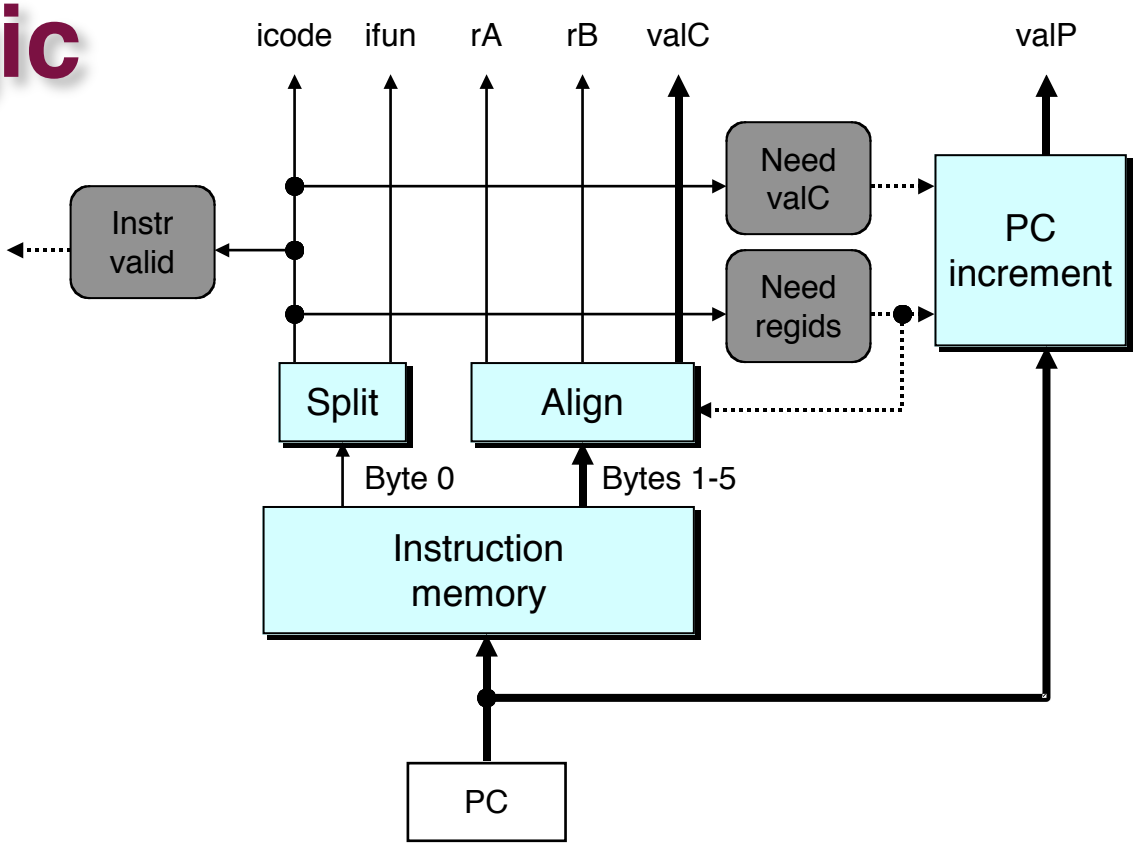
Fetch Logic



Predefined Blocks

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

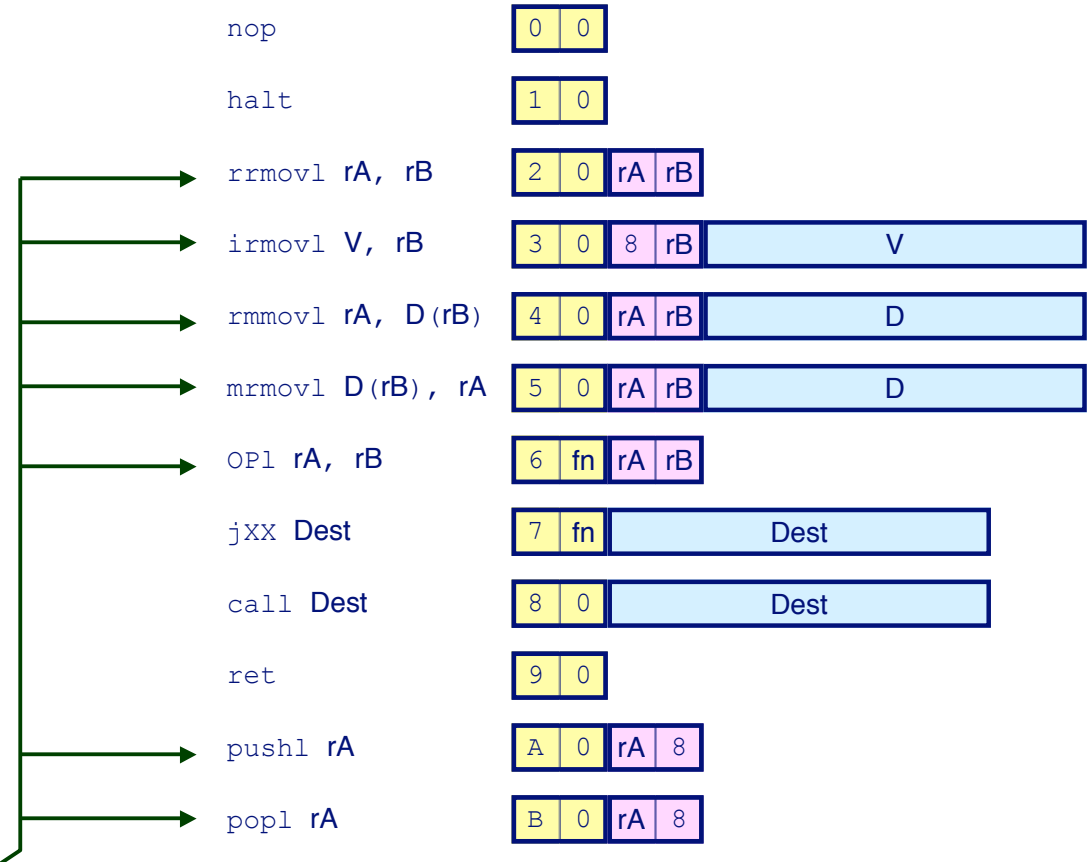
Fetch Logic



Control Logic

- **Instr. Valid: Is this instruction valid?**
- **Need regids: Does this instruction have a register bytes?**
- **Need valC: Does this instruction have a constant word?**

Fetch Control Logic




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

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

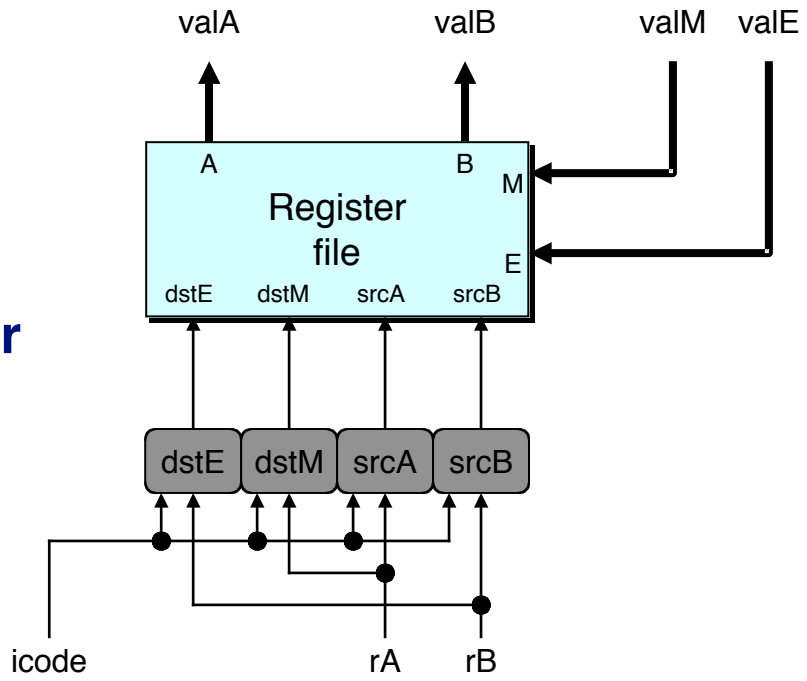
Decode Logic

Register File

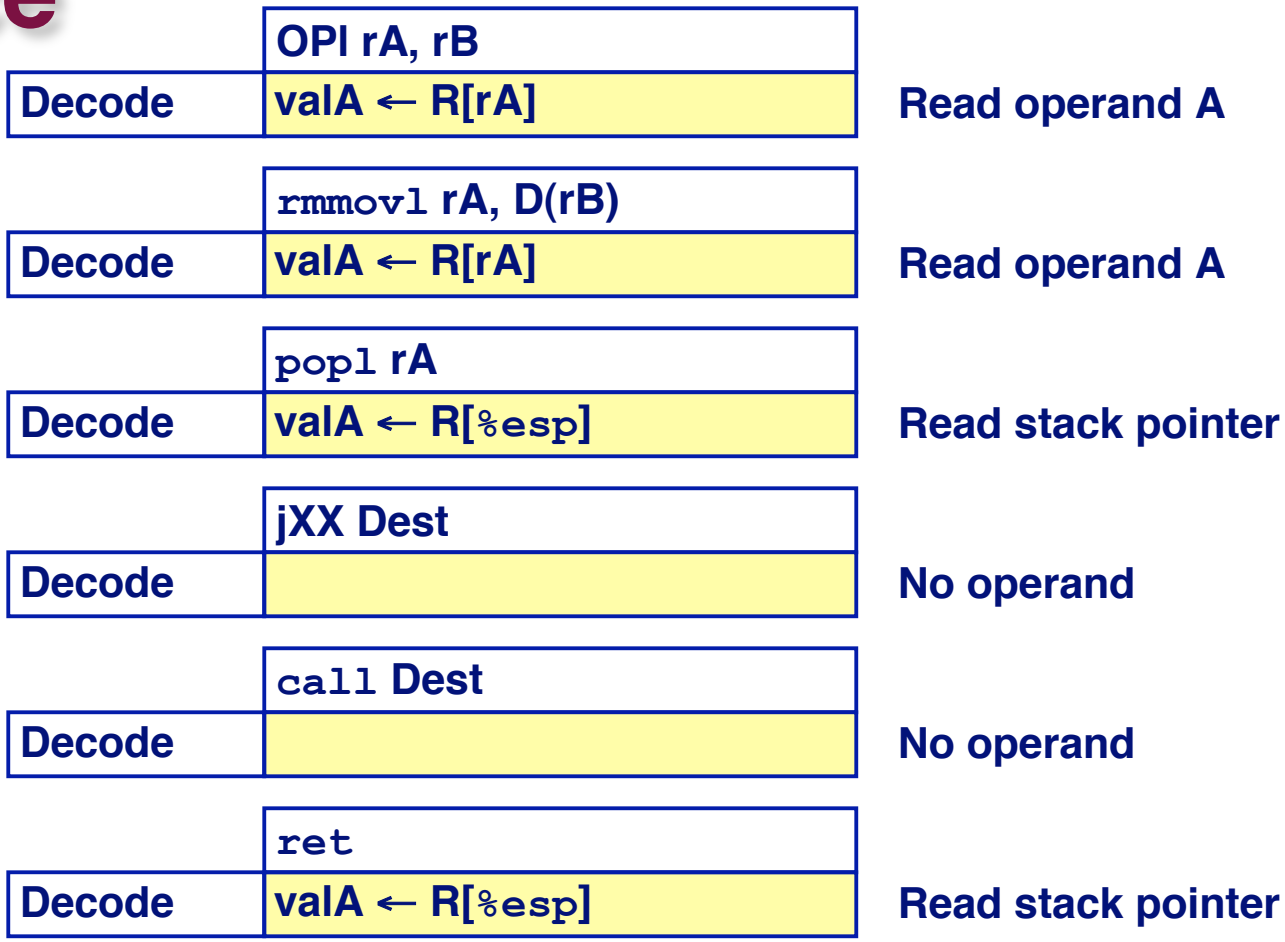
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 8 (no access)

Control Logic

- srcA, srcB: read port addresses
- dstA, dstB: write port addresses

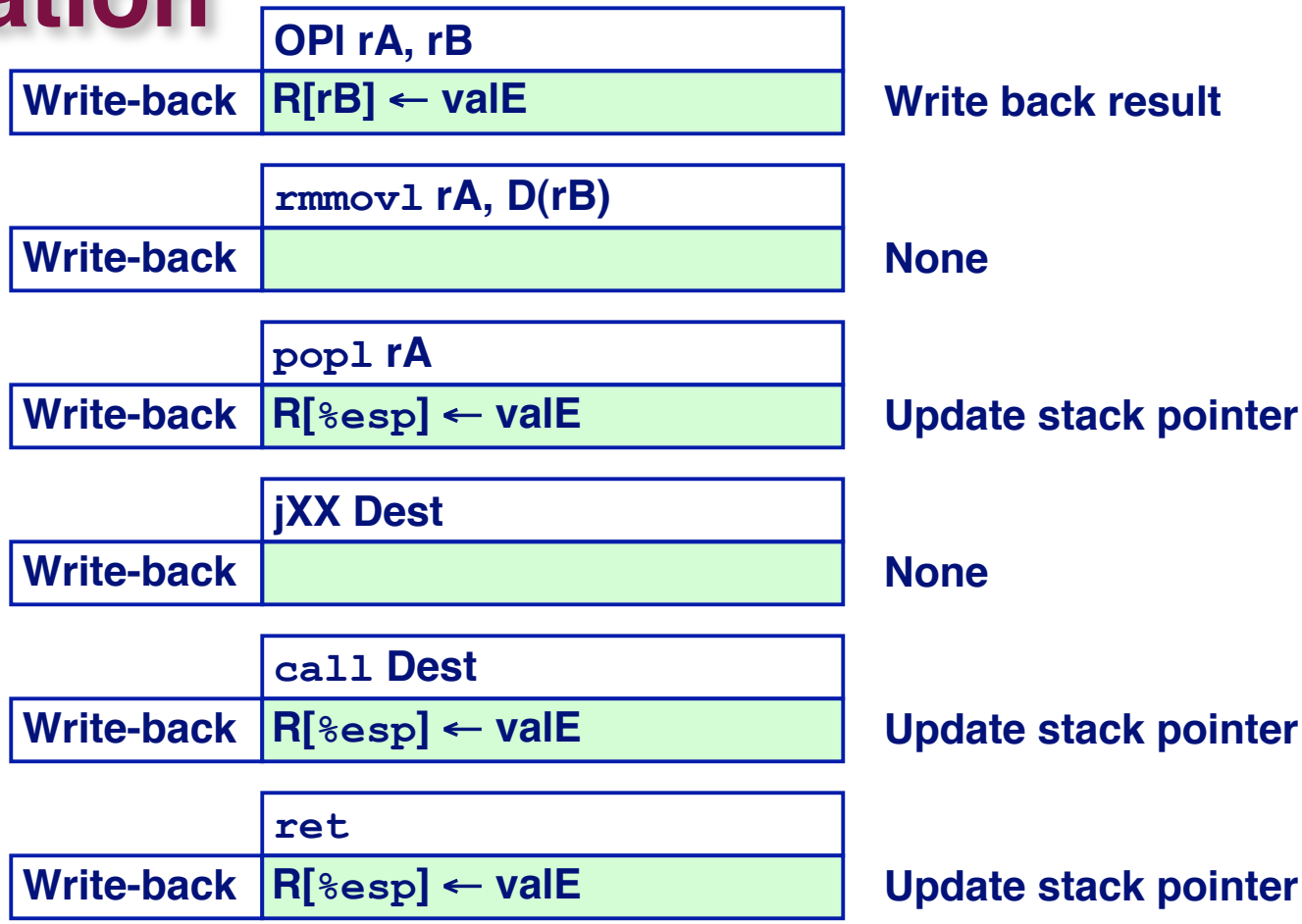


A Source




```
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, IIRMOVL, IOPL} : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];
```

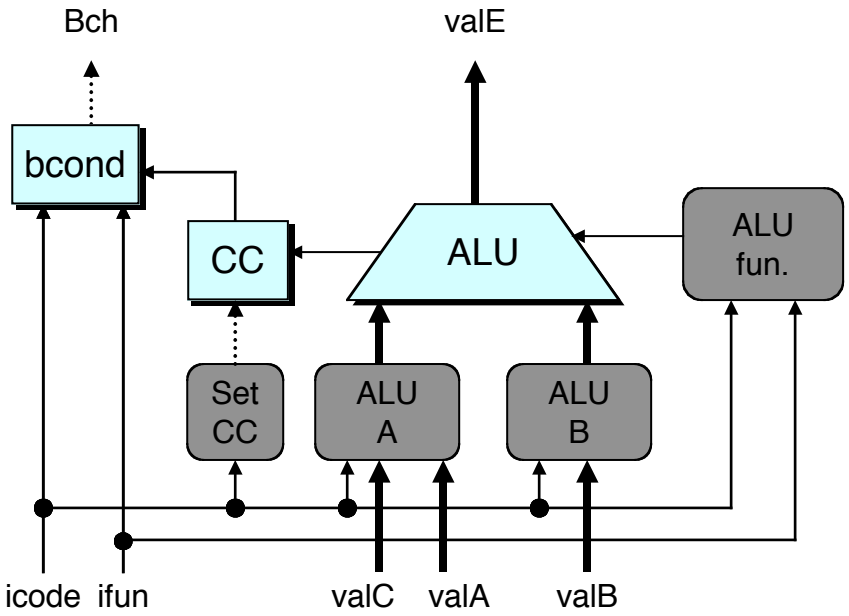
Execute Logic

Units

- ALU
  - Implements 4 required functions
  - Generates condition code values
- CC
  - Register with 3 condition code bits
- bcond
  - Computes branch 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

	OPl rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	`rmmovl` rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	`popl` rA	
Execute	valE ← valB + 4	Increment stack pointer
	jXX Dest	
Execute		No operation
	`call` Dest	
Execute	valE ← valB + –4	Decrement stack pointer
	`ret`	
Execute	valE ← valB + 4	Increment stack pointer

```
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

	OPl rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation

	`rmmovl` rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address

	`popl` rA	
Execute	valE ← valB + 4	Increment stack pointer

	jXX Dest	
Execute		No operation

	`call` Dest	
Execute	valE ← valB + –4	Decrement stack pointer

	`ret`	
Execute	valE ← valB + 4	Increment stack pointer

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

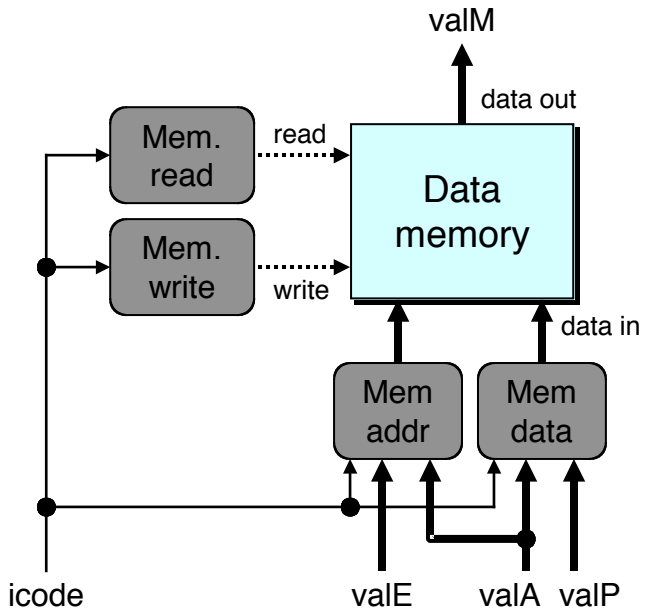
Memory Logic

Memory

- Reads or writes memory word

Control Logic

- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



Memory Address

	OPl rA, rB	
Memory		No operation

	`rmmovl` rA, D(rB)	
Memory	M_4[valE] ← valA	Write value to memory

	`popl` rA	
Memory	valM ← M_4[valA]	Read from stack

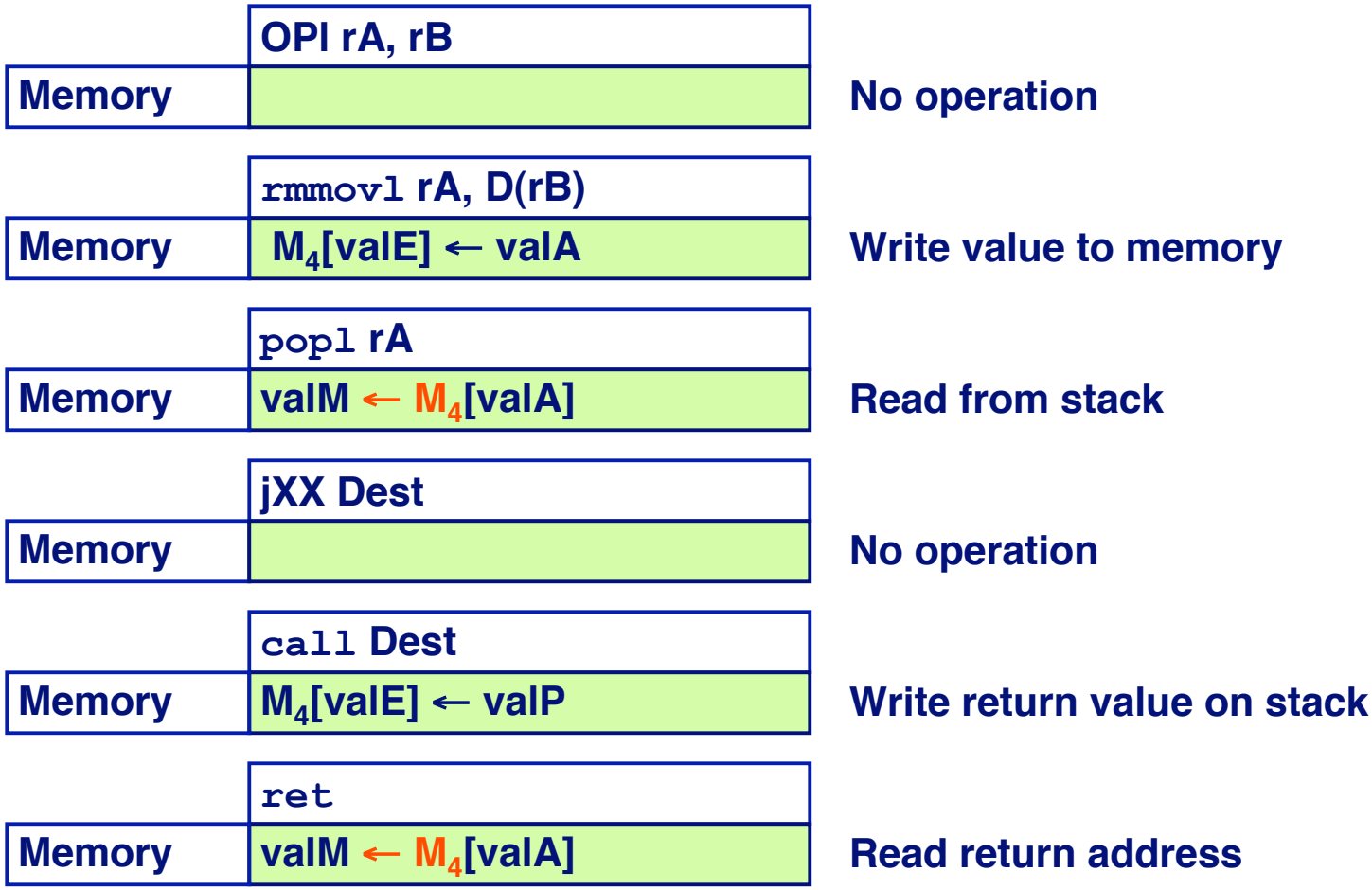
	jXX Dest	
Memory		No operation

	`call` Dest	
Memory	M_4[valE] ← valP	Write return value on stack

	`ret`	
Memory	valM ← M_4[valA]	Read return address

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

Memory Read

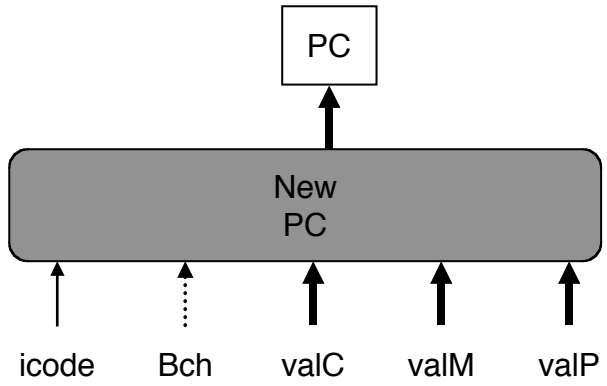



```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

PC Update Logic

New PC

- Select next value of PC



PC Update

	OPl rA, rB	
PC update	PC ← valP	Update PC

	`rmmovl` rA, D(rB)	
PC update	PC ← valP	Update PC

	`popl` rA	
PC update	PC ← valP	Update PC

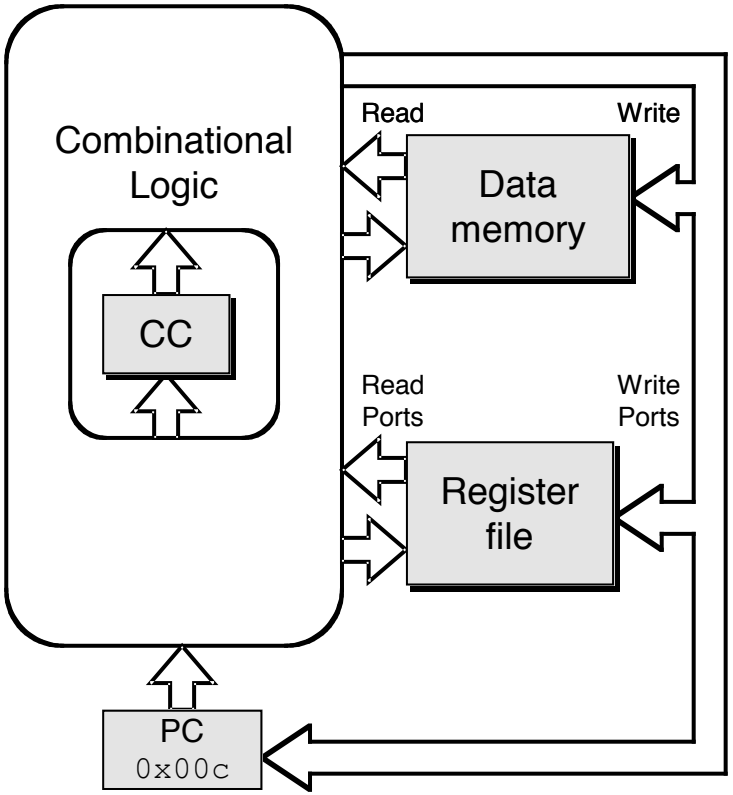
	jXX Dest	
PC update	PC ← Bch ? valC : valP	Update PC

	`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 && Bch : valC;
        icode == IRET : valM;
        1 : valP;
];
```

SEQ Operation



State

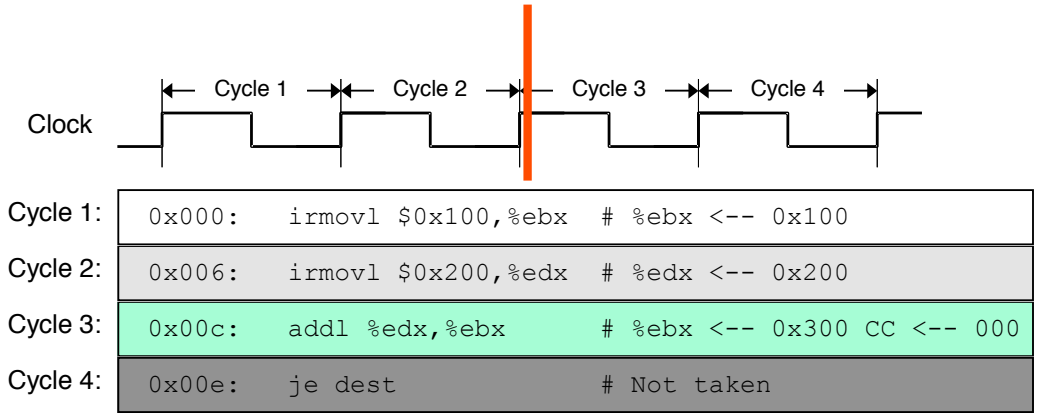
- PC register
- Cond. Code register
- Data memory
- Register file

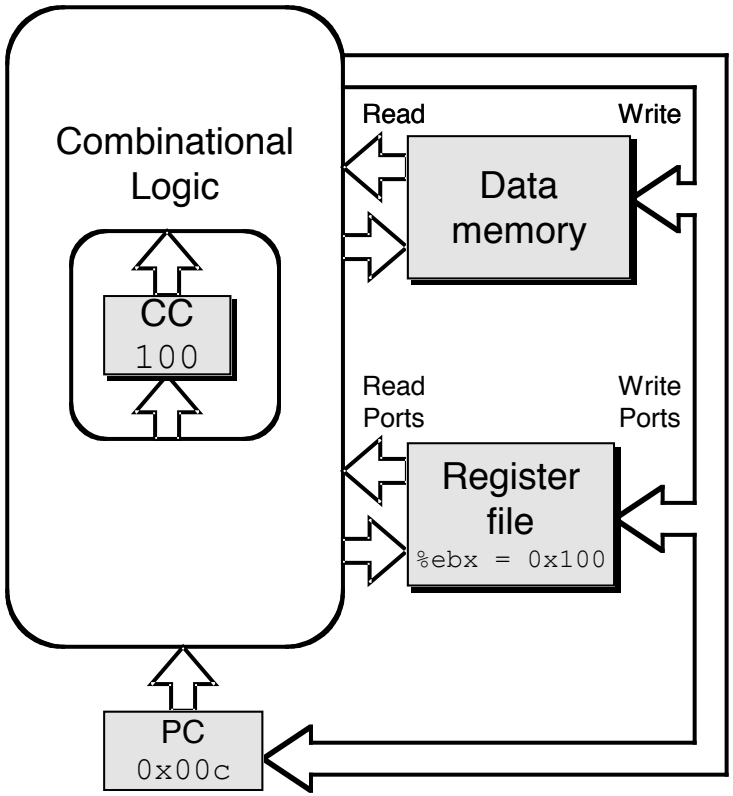
All updated as clock rises

Combinational Logic

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

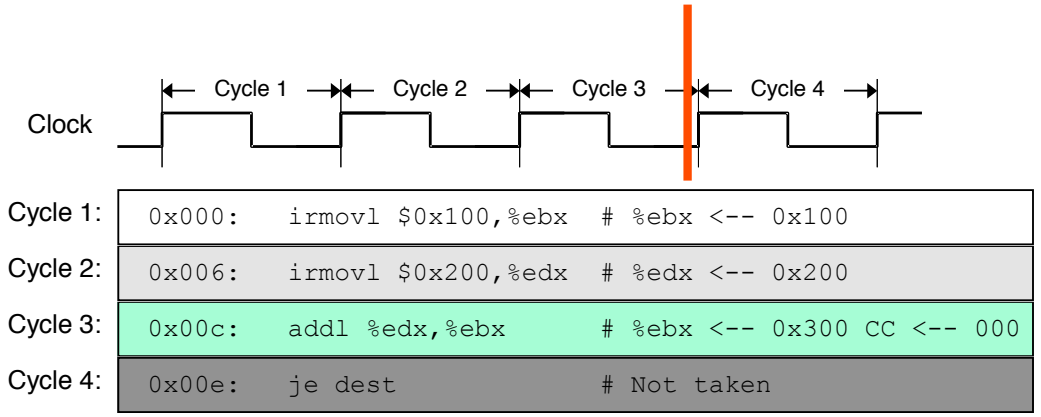
SEQ Operation #2

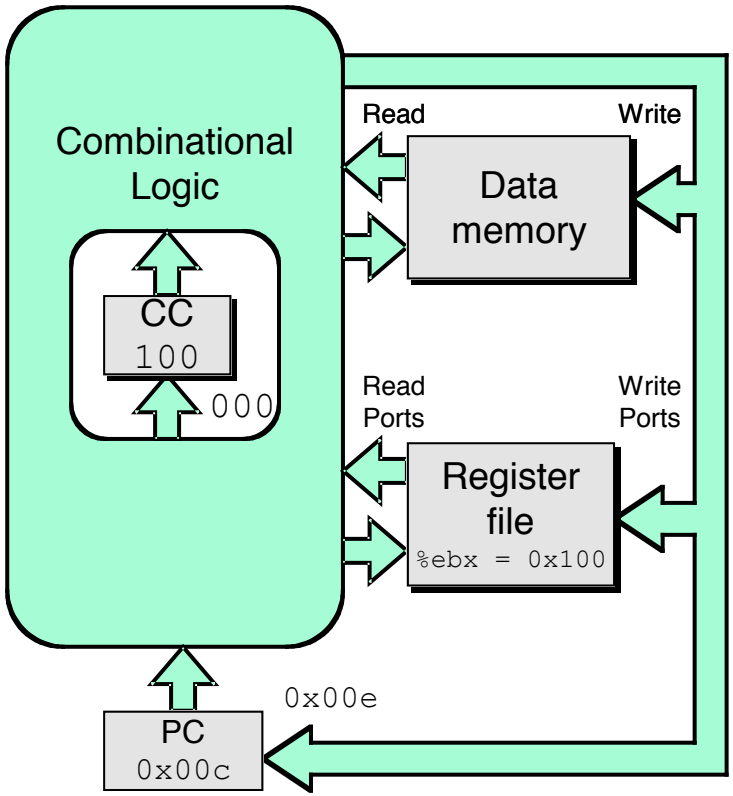




- state set according to second `irmovl` instruction
- combinational logic starting to react to state changes

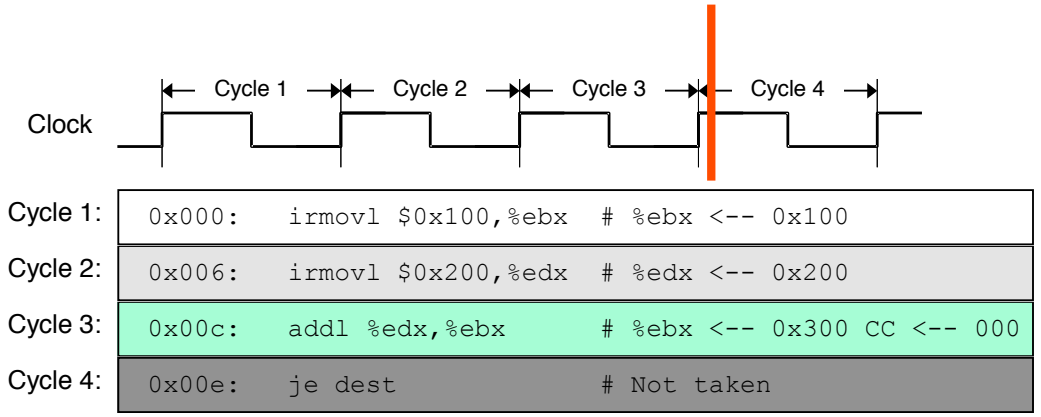
SEQ Operation #3

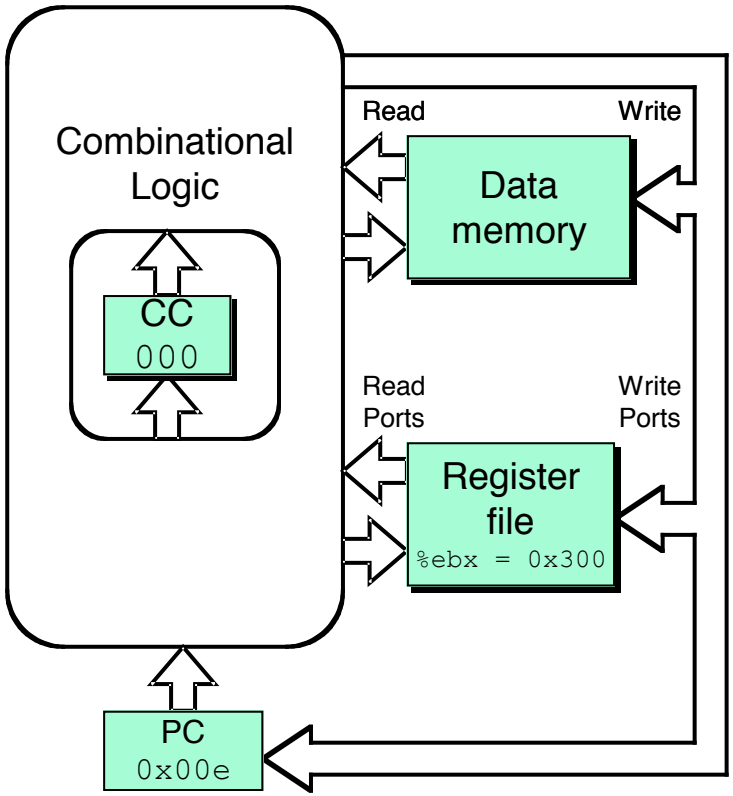




- **state set according to second `irmovl` instruction**
- **combinational logic generates results for `addl` instruction**

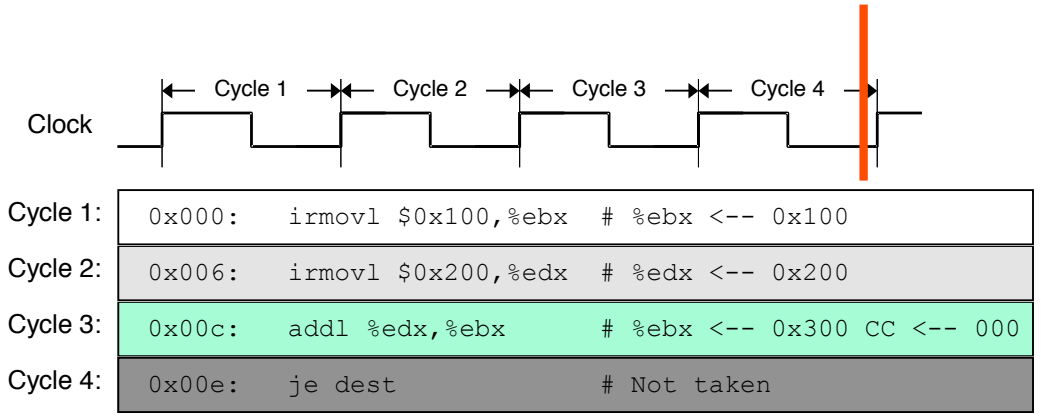
SEQ Operation #4

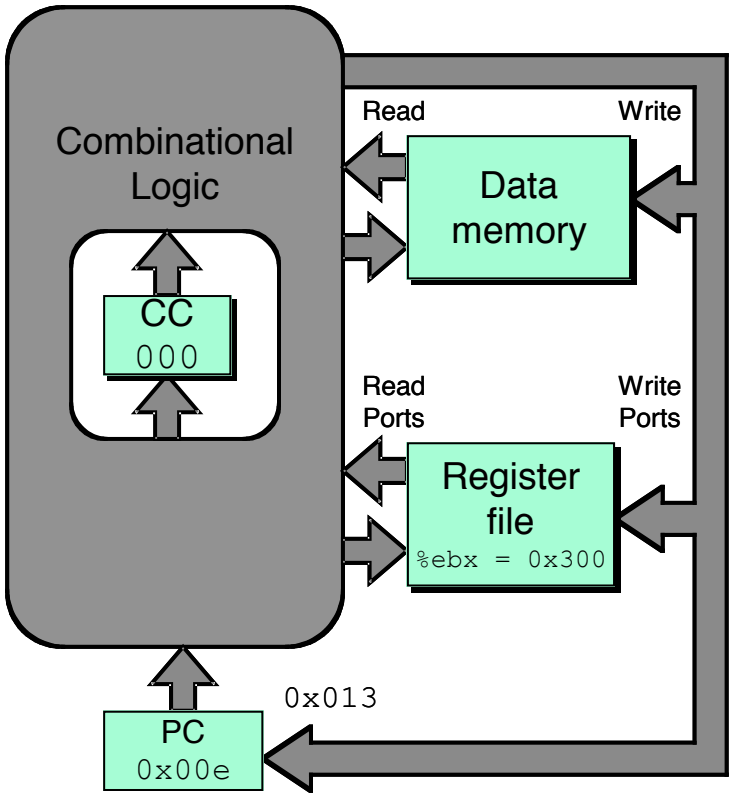




- **state set according to `addl` instruction**
- **combinational logic starting to react to state changes**

SEQ Operation #5





- **state set according to `addl` instruction**
- **combinational logic generates results for `je` instruction**

SEQ Summary

Implementation

- Express every instruction as series of simple steps
- Follow 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