# Intro to Architecture of Computers – II
# The Von Neumann Model

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

(Slides include copyright materials from *Computer Systems: A Programmer’s Perspective*, by Bryant and O’Hallaron, and from *The C Programming Language*, by Kernighan and Ritchie)

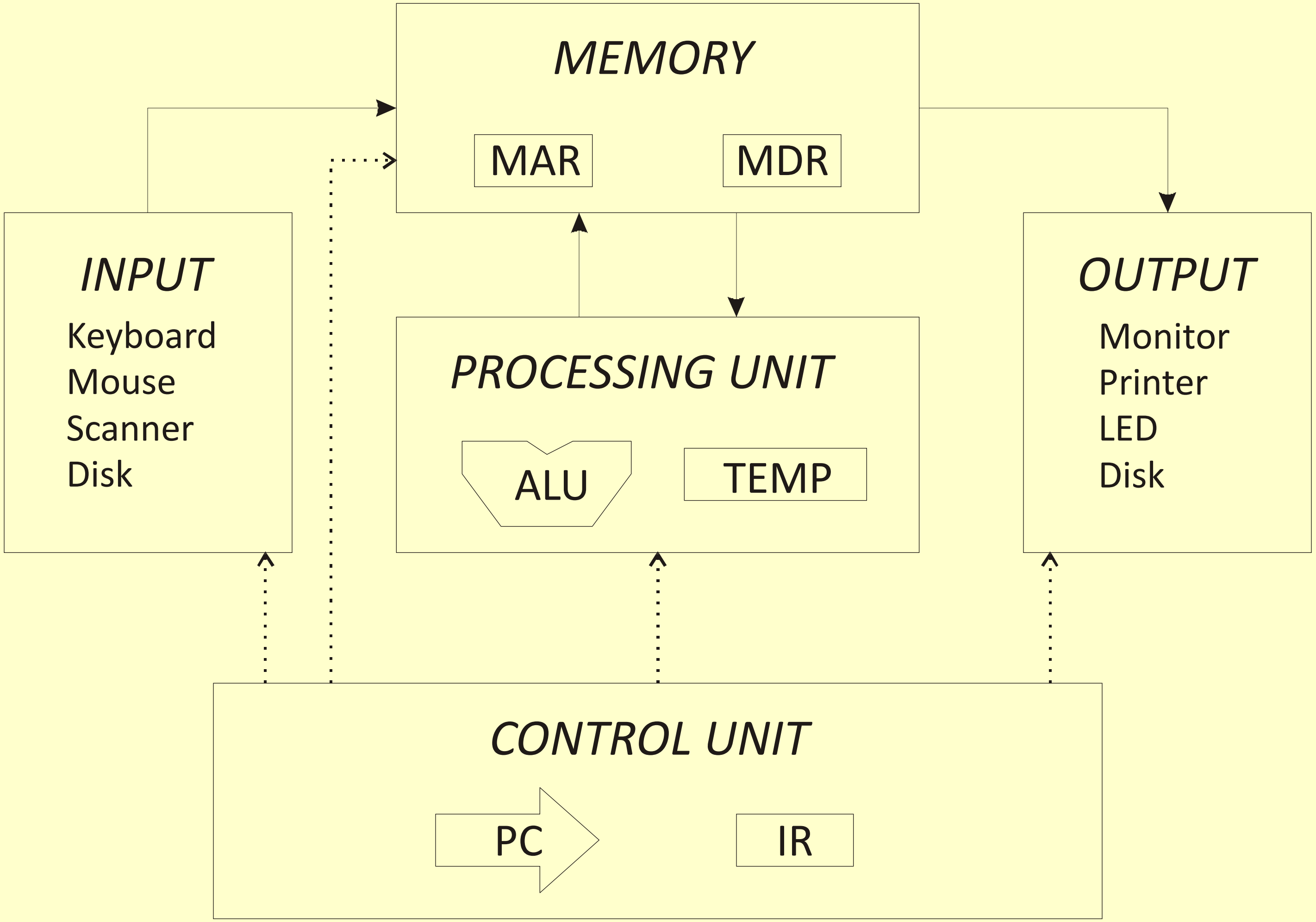
# Prior to John Von Neumann

- **“Programs” for mechanical and electro-mechanical calculators were**
  - Wired into plug-board or
  - Coded onto fixed punched cards or paper tape
  - Executed strictly as finite state machines

- **Von Neumann’s innovation**
  - Store programs in memory of the machine
  - Enable programs to update their own code!

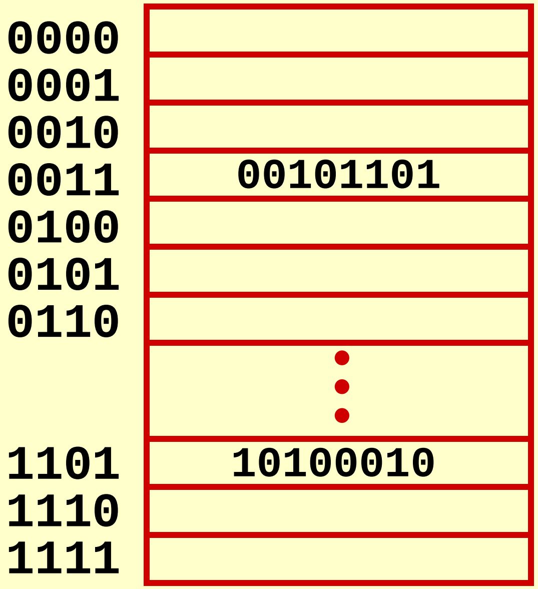
- **Turing’s contribution**
  - These two elements constituted a realistic implementation of his mathematical model of computability

# Von Neumann Model



# Memory

- $2^k$ x *m* array of stored bits
- Address
  - unique (*k*-bit) identifier of location
- Contents
  - *m*-bit value stored in location

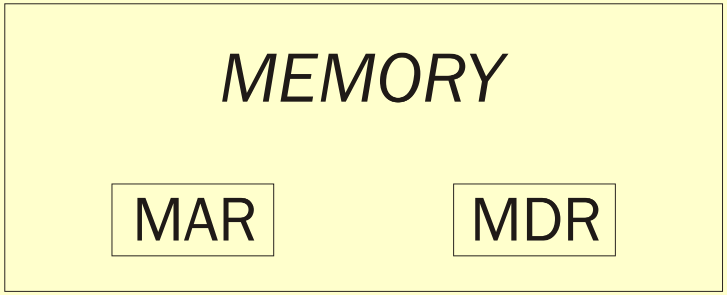


- Basic Operations:
- LOAD
  - read a value from a memory location
- STORE
  - write a value to a memory location

In reality, the Memory Hierarchy

# Interface to Memory

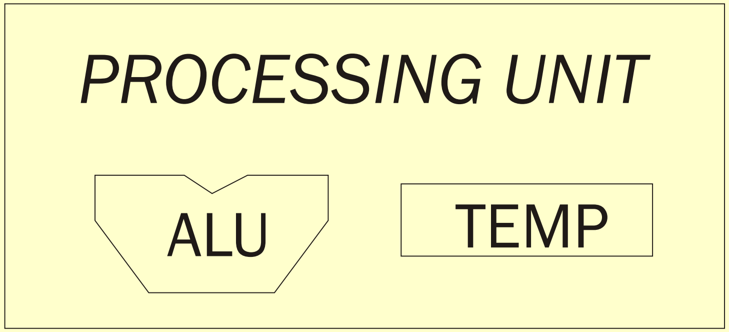
- **How does processing unit get data to/from memory?**
- **MAR: Memory Address Register**
- **MDR: Memory Data Register**



- **To LOAD a location (A):**
  1. Write address (A) into MAR
  2. Send a "read" signal to memory
  3. (After the passage of time) Read data from MDR
- **To STORE a value (X) to a location (A):**
  1. Write data (X) to MDR
  2. Write address (A) into MAR
  3. Send a "write" signal to the memory

# Processing Unit

- **Functional Units**
  - ALU = Arithmetic and Logic Unit
  - could have many functional units. some of them special-purpose (multiply, square root, …)
  - x86-64 has about 6-8 units



- **Registers**
  - Small, temporary storage
  - Operands and results of functional units
  - IA32 has eight — **`%eax`**, **`%ecx`**, **`%edx`**, **`%ebx`**, **`%esi`**, **`%edi`**, **`%ebp`**, **`%esp`**
- **Word Size**
  - number of bits normally processed by ALU in one instruction
  - also width of registers
  - IA32 is 32-bits; x86_64 is 64-bits

# Input and Output

- **Devices for getting data into and out of computer memory**

*INPUT*

**Keyboard**
**Mouse**
**Scanner**
**Disk**

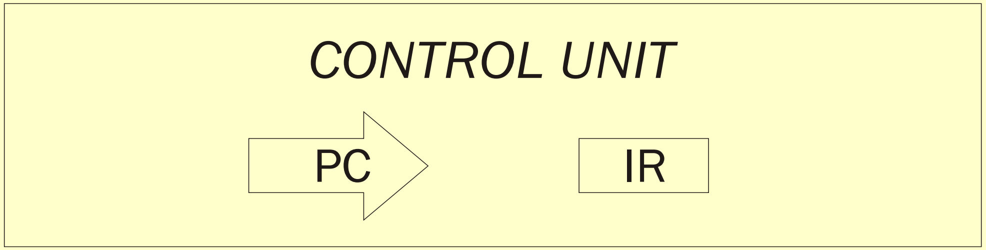
*OUTPUT*

**Monitor**
**Printer**
**LED**
**Disk**

- **Each device has its own interface, usually a set of registers like the memory's MAR and MDR**
  - Laptops and desktop have keyboards and mice
  - Smartphones, game consoles have other devices
  - Embedded systems have digital channels, analog channels, etc.
  - monitor: data register (DDR) and status register (DSR)
- **Some devices provide both input and output**
  - disk, network
- **Program that controls access to a device is usually called a *driver*.**

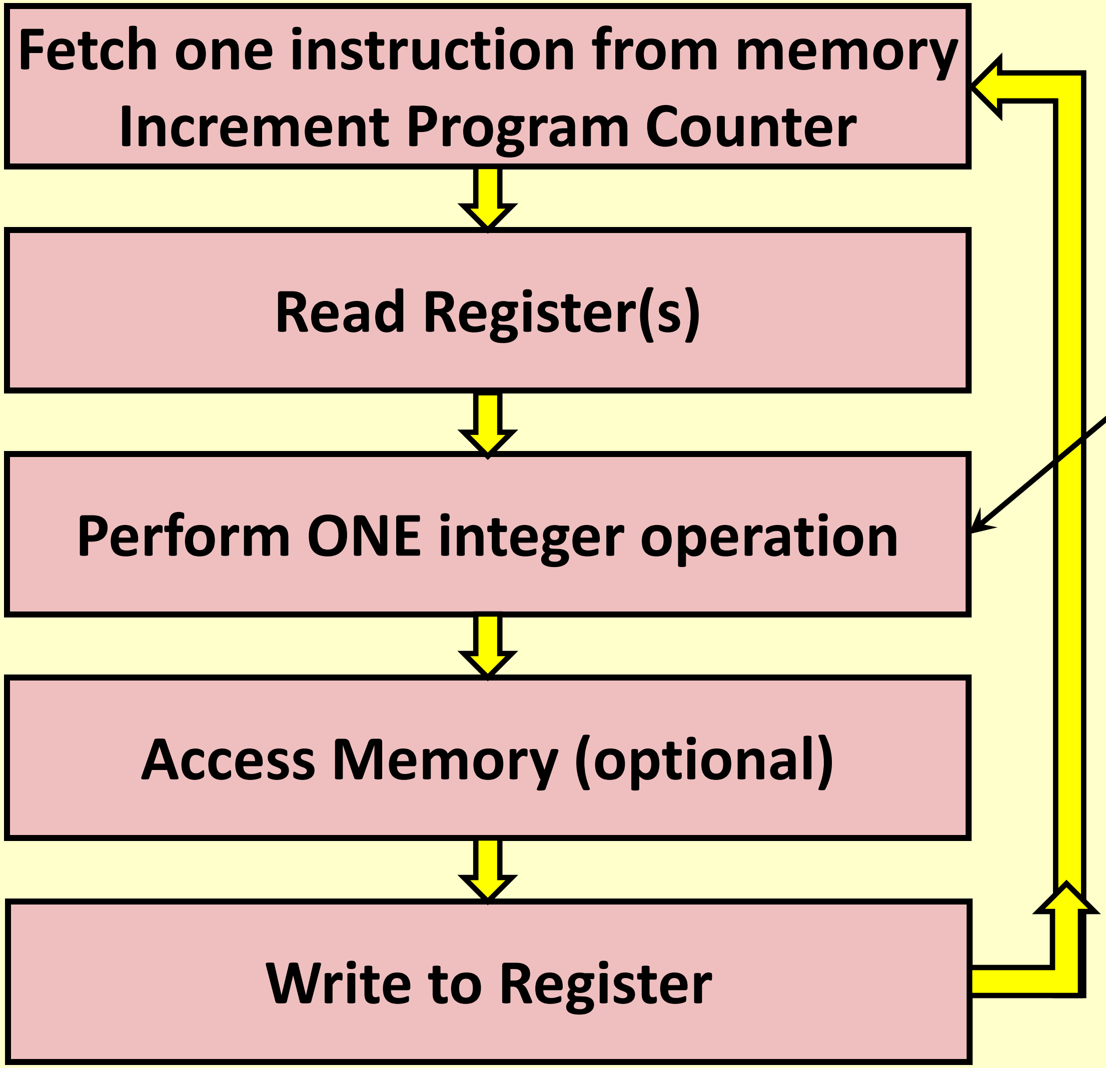
# Control Unit

- **Orchestrates execution of the program**



- **Instruction Register (IR) contains the *current instruction*.**
- **Program Counter (PC) contains the *address* of the next instruction to be executed.**
- **Control unit:**
  - reads an instruction from memory
    - the instruction's address is in the PC
  - interprets the instruction, generating signals that tell the other components what to do
    - an instruction may take many *machine cycles* to complete

# Execution Model for Modern  Computers


Fetch one instruction from memory
Increment Program Counter
Read Register(s)
Perform ONE integer operation
Access Memory (optional)
Write to Register

Often involves and address calculation

From an earlier topic

# Sequential 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

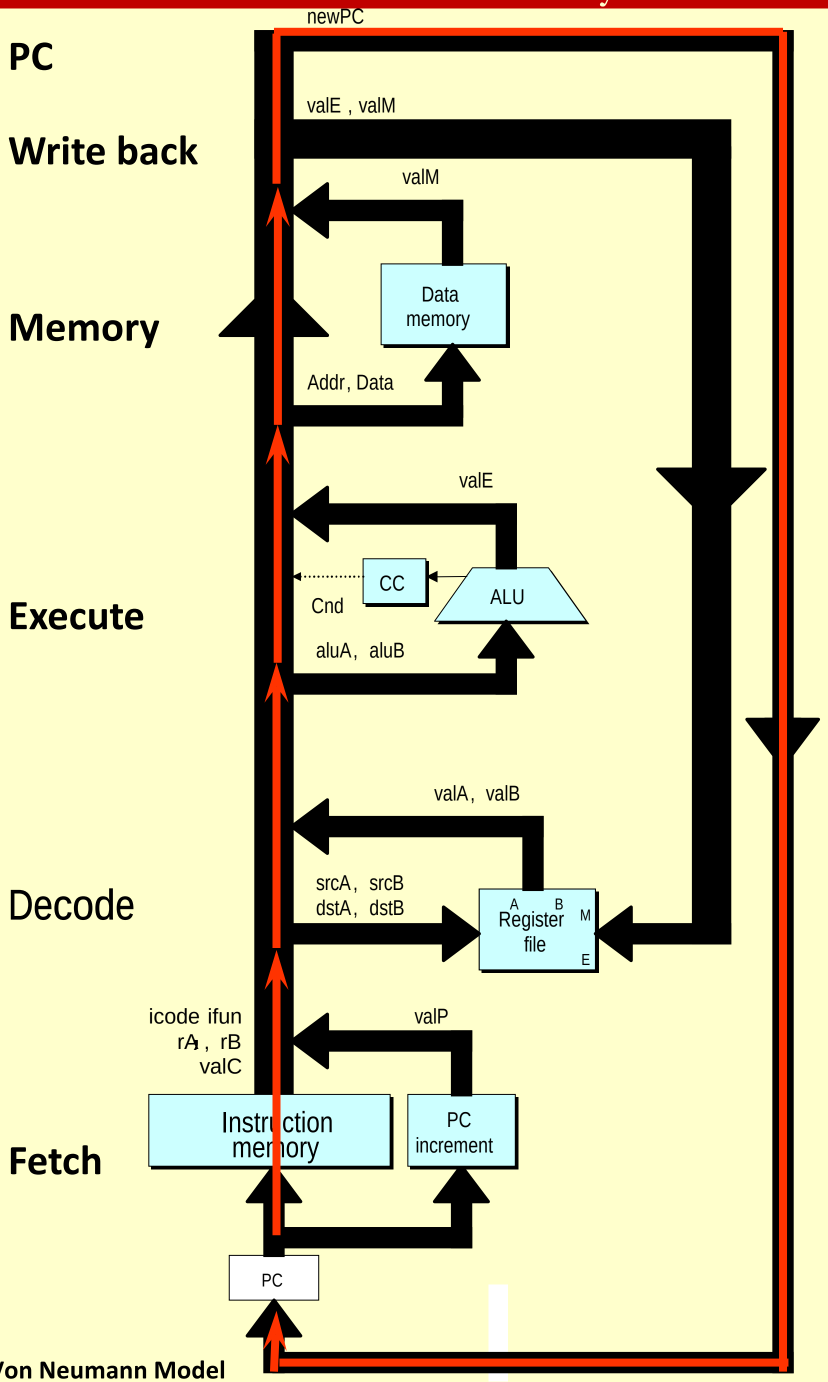
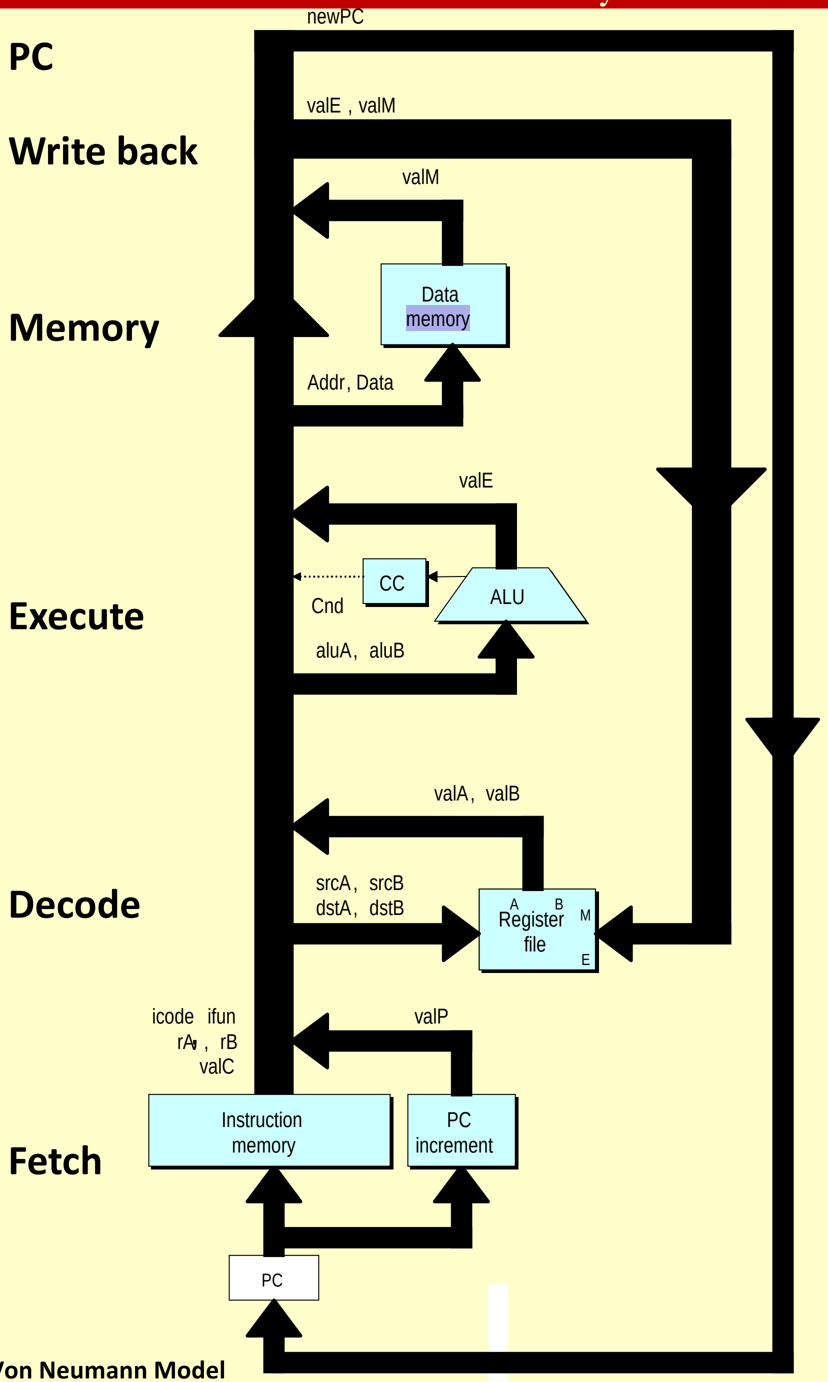


**Figure 4.22, p. 376**

# Sequential Stages

- **Fetch**
  - Read instruction from instruction memory
  - Increment program counter (**`%eip`**)
- **Decode**
  - Read registers named in instruction
- **Execute**
  - Compute value or address
- **Memory**
  - Read or write data
- **Write Back**
  - Write program registers
- **PC**
  - Update program counter (incremented or jump/call/ret target)



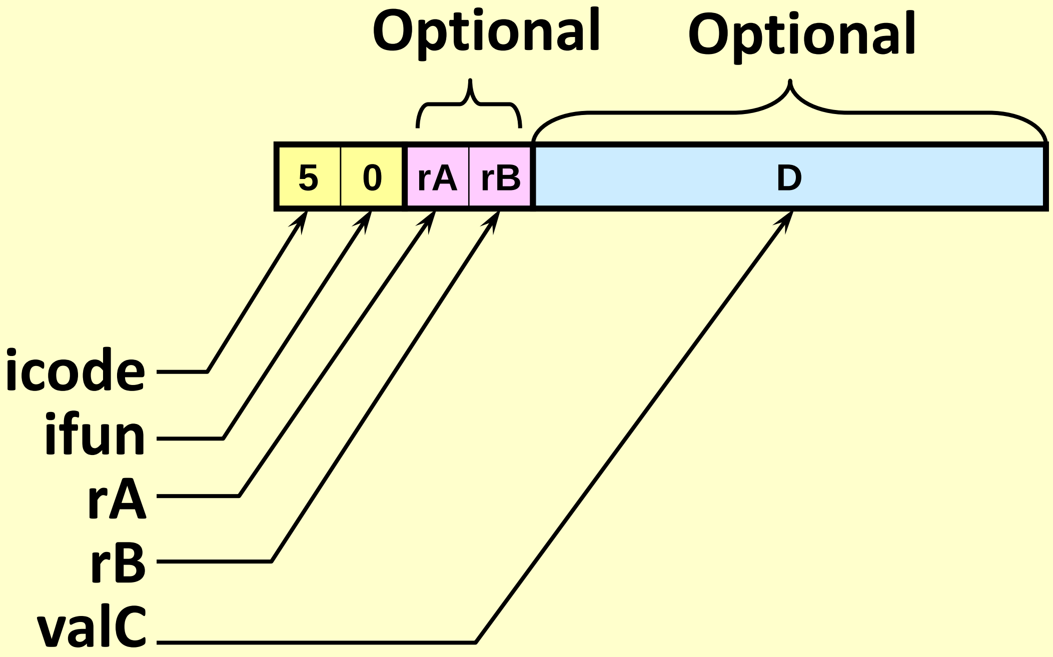
# Instruction

- **The instruction is the fundamental unit of work.**
- **Specifies two things:**
  - *opcode*: operation to be performed
  - *operands*: data/locations to be used for operation

- **An instruction is encoded as a sequence of bits.**
  ***(Just like data!)***
  - May be fixed or variable length
  - Control unit interprets instruction;
    generates sequence of control signals to carry out operation.
  - Operation is either executed completely, or not at all.

- **A computer's instructions and their formats is known as its**
  *Instruction Set Architecture* *(ISA)*.

# Instruction Decoding



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

# Y86 Instruction Set #1

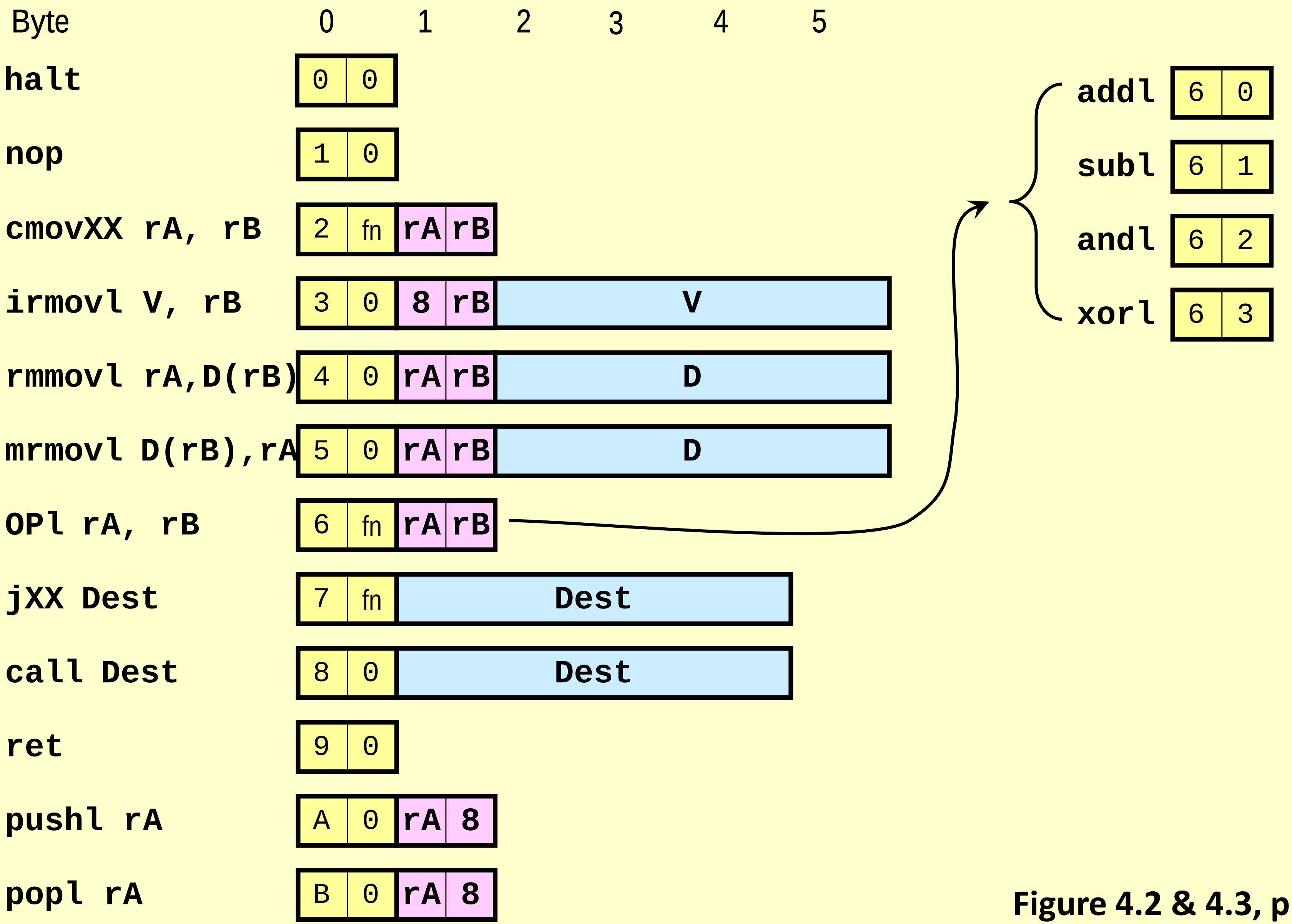
| Byte | 0 | | 1 | | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|
| halt | 0 | 0 | | | | | | |
| nop | 1 | 0 | | | | | | |
| cmovXX rA, rB | 2 | fn | rA | rB | | | | |
| irmovl V, rB | 3 | 0 | 8 | rB | V | | | |
| rmmovl rA,D(rB) | 4 | 0 | rA | rB | D | | | |
| mrmovl D(rB),rA | 5 | 0 | rA | rB | D | | | |
| OPl rA, rB | 6 | fn | rA | rB | | | | |
| jXX Dest | 7 | fn | Dest | | | | | |
| call Dest | 8 | 0 | Dest | | | | | |
| ret | 9 | 0 | | | | | | |
| pushl rA | A | 0 | rA | 8 | | | | |
| popl rA | B | 0 | rA | 8 | | | | |

| | | |
|---|---|---|
| rrmovl | 2 | 0 |
| cmovle | 2 | 1 |
| cmovl | 2 | 2 |
| cmove | 2 | 3 |
| cmovne | 2 | 4 |
| cmovge | 2 | 5 |
| cmovg | 2 | 6 |

Note: the Y86 is Bryant's & O'Hallaron's stripped down IA32 example architecture

Figure 4.2 & 4.3, p. 338

# Y86 Instruction Set #2

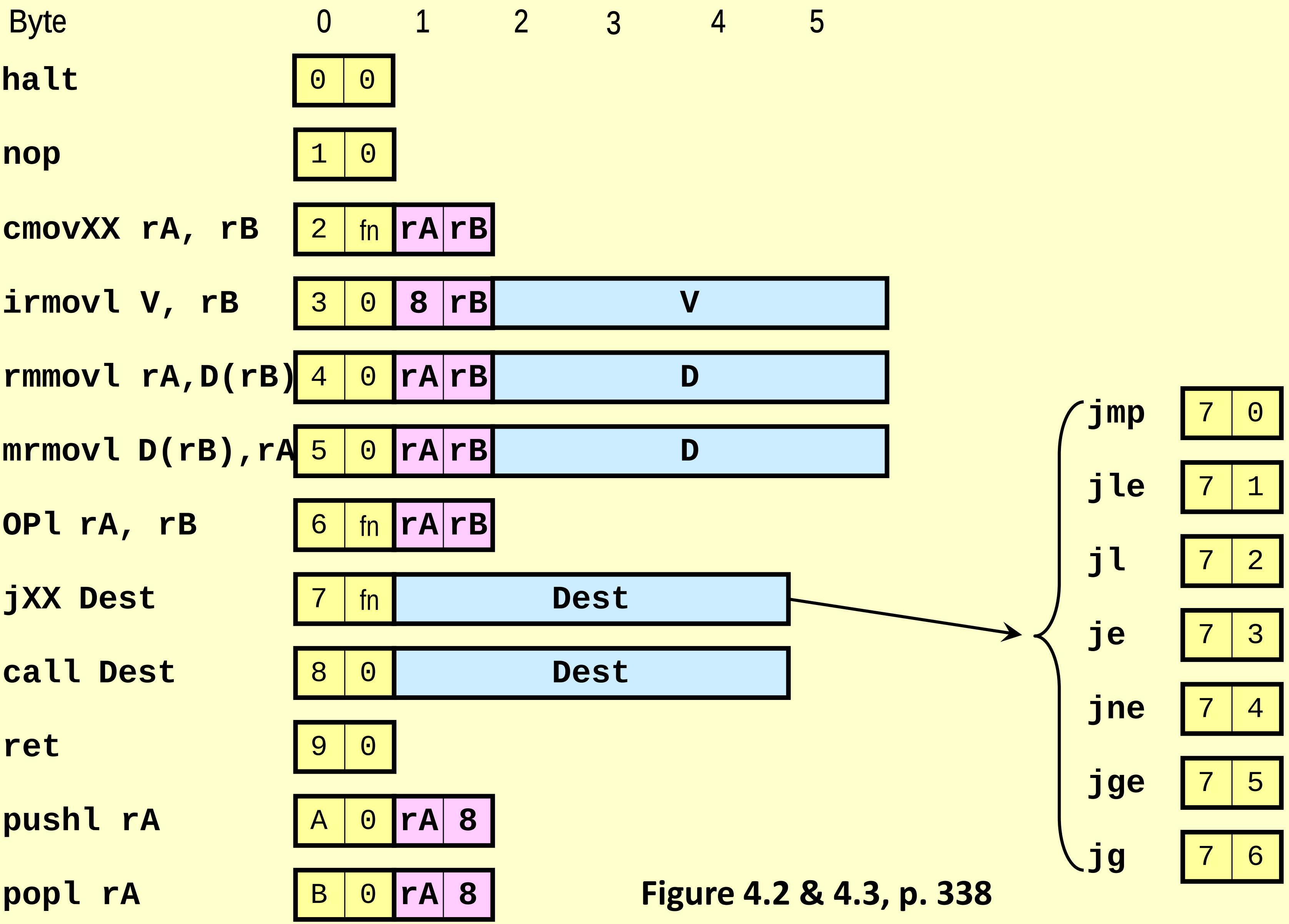


| Byte | 0 | | 1 | | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|
| halt | 0 | 0 | | | | | | |
| nop | 1 | 0 | | | | | | |
| cmovXX rA, rB | 2 | fn | rA | rB | | | | |
| irmovl V, rB | 3 | 0 | 8 | rB | V | | | |
| rmmovl rA,D(rB) | 4 | 0 | rA | rB | D | | | |
| mrmovl D(rB),rA | 5 | 0 | rA | rB | D | | | |
| OPl rA, rB | 6 | fn | rA | rB | | | | |
| jXX Dest | 7 | fn | Dest | | | | | |
| call Dest | 8 | 0 | Dest | | | | | |
| ret | 9 | 0 | | | | | | |
| pushl rA | A | 0 | rA | 8 | | | | |
| popl rA | B | 0 | rA | 8 | | | | |

| | | |
|---|---|---|
| addl | 6 | 0 |
| subl | 6 | 1 |
| andl | 6 | 2 |
| xorl | 6 | 3 |

Figure 4.2 & 4.3, p. 338

# Y86 Instruction Set #3



| Byte | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| halt | 0 0 | | | | | |
| nop | 1 0 | | | | | |
| cmovXX rA, rB | 2 fn | rA rB | | | | |
| irmovl V, rB | 3 0 | 8 rB | V | | | |
| rmmovl rA,D(rB) | 4 0 | rA rB | D | | | |
| mrmovl D(rB),rA | 5 0 | rA rB | D | | | |
| OPl rA, rB | 6 fn | rA rB | | | | |
| jXX Dest | 7 fn | Dest | | | | |
| call Dest | 8 0 | Dest | | | | |
| ret | 9 0 | | | | | |
| pushl rA | A 0 | rA 8 | | | | |
| popl rA | B 0 | rA 8 | | | | |

| jXX | Byte 0 |
|---|---|
| jmp | 7 0 |
| jle | 7 1 |
| jl | 7 2 |
| je | 7 3 |
| jne | 7 4 |
| jge | 7 5 |
| jg | 7 6 |

Figure 4.2 & 4.3, p. 338

# Executing Arithmetic/Logical Operation

`OPl rA, rB`

|   |    |    |    |
|---|----|----|----|
| 6 | fn | rA | rB |

- **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: Arith/Logical Ops

| | OPl rA, rB | |
|---|---|---|
| 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[rA] | Read operand A |
| | valB ← R[rB] | Read operand B |
| Execute | valE ← valB OP valA | Perform ALU operation |
| | Set CC | 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`

`rmmovl rA, D(rB)`

| | | | | |
|---|---|---|---|---|
| 4 | 0 | rA | rB | D |

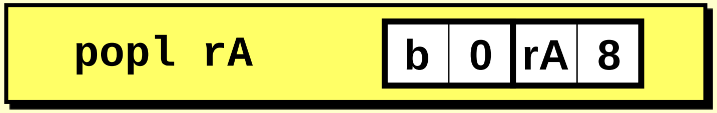
- **Fetch**
  - Read 6 bytes
- **Decode**
  - Read operand registers
- **Execute**
  - Compute effective address
- **Memory**
  - Write to memory
- **Write back**
  - Do nothing
- **PC Update**
  - Increment PC by 6

# Stage Computation: rmmovl

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

- Use ALU for address computation

# Executing `popl`



`popl rA`

| b | 0 | rA | 8 |
|---|---|---|---|

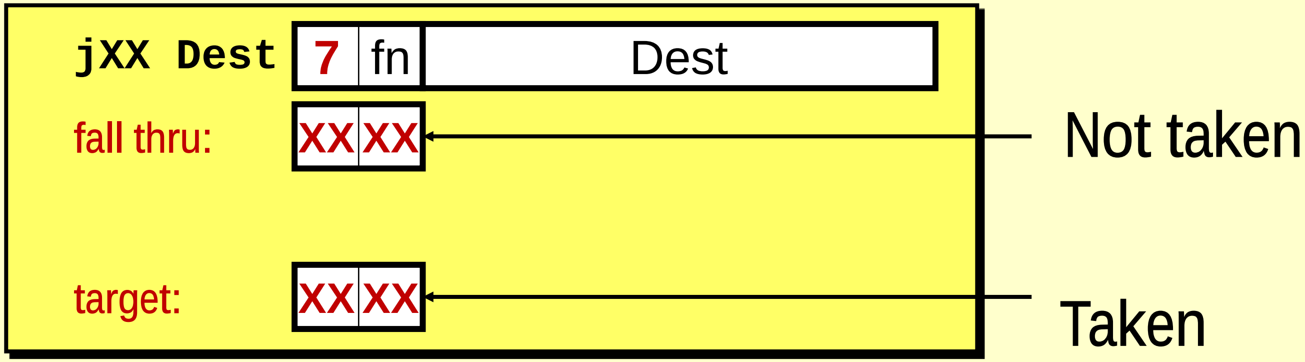
- **Fetch**
  - Read 2 bytes
- **Decode**
  - Read stack pointer
- **Execute**
  - Increment stack pointer by 4
- **Memory**
  - Read from old stack pointer
- **Write back**
  - Update stack pointer
  - Write result to register
- **PC Update**
  - Increment PC by 2

# Stage Computation: `popl`

| | **`popl`** rA | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC]<br>rA:rB ← $M_1$[PC+1]<br>valP ← PC+2 | Read instruction byte<br>Read register byte<br>Compute next PC |
| Decode | valA ← R[**%esp**]<br>valB ← R [**%esp**] | Read stack pointer<br>Read stack pointer |
| Execute | valE ← valB + 4 | Increment stack pointer |
| Memory | valM ← $M_4$[valA] | Read from stack |
| Write<br>back | R[**%esp**] ← valE<br>R[rA] ← valM | Update stack pointer<br>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 Jumps



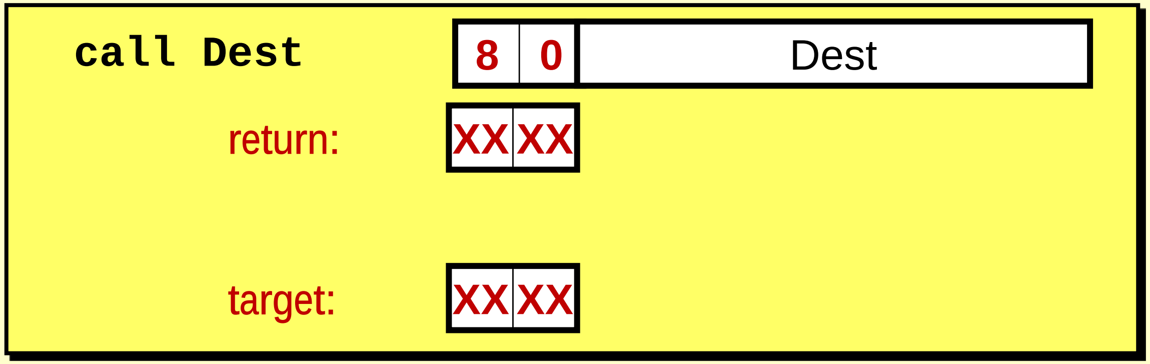
- **Fetch**
  - Read 5 bytes
  - Increment PC by 5
- **Decode**
  - Do nothing
- **Execute**
  - Determine whether to take branch based on jump condition and condition codes
- **Memory**
  - Do nothing
- **Write back**
  - Do nothing
- **PC Update**
  - Set PC to Dest if branch taken or to incremented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

# Stage Computation: Jumps

| | jXX Dest | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC] | Read instruction byte |
| | valC ← $M_4$[PC+1] | Read destination address |
| | valP ← PC+5 | 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`



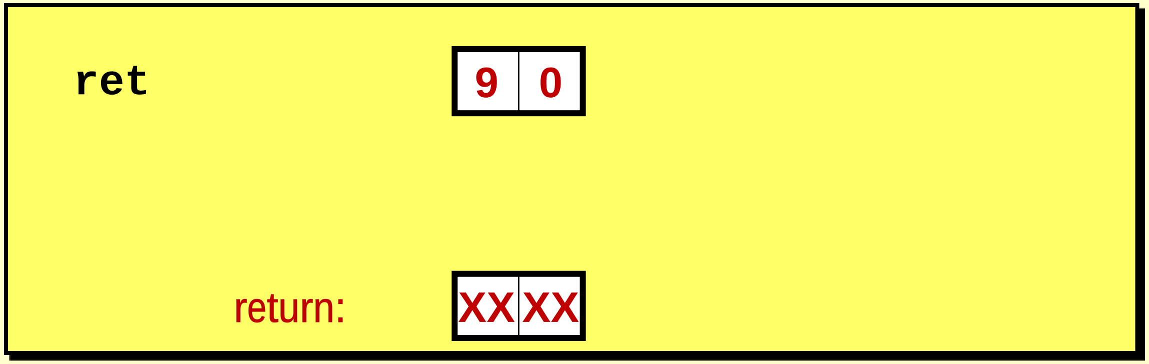
- **Fetch**
  - Read 5 bytes
  - Increment PC by 5
- **Decode**
  - Read stack pointer
- **Execute**
  - Decrement stack pointer by 4
- **Memory**
  - Write incremented PC to new value of stack pointer
- **Write back**
  - Update stack pointer
- **PC Update**
  - Set PC to Dest

# Stage Computation: `call`

| | **`call`** Dest | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC] | Read instruction byte |
| | valC ← $M_4$[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
- Store incremented PC

# Executing `ret`



ret 9 0

return: XX XX

- **Fetch**
  - Read 1 byte
- **Decode**
  - Read stack pointer
- **Execute**
  - Increment stack pointer by 4
- **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[**%esp**]<br>valB ← R[**%esp**] | Read operand stack pointer<br>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

# 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 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 ← $M_4$[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 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 |

# Sequential Hardware Implementation

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

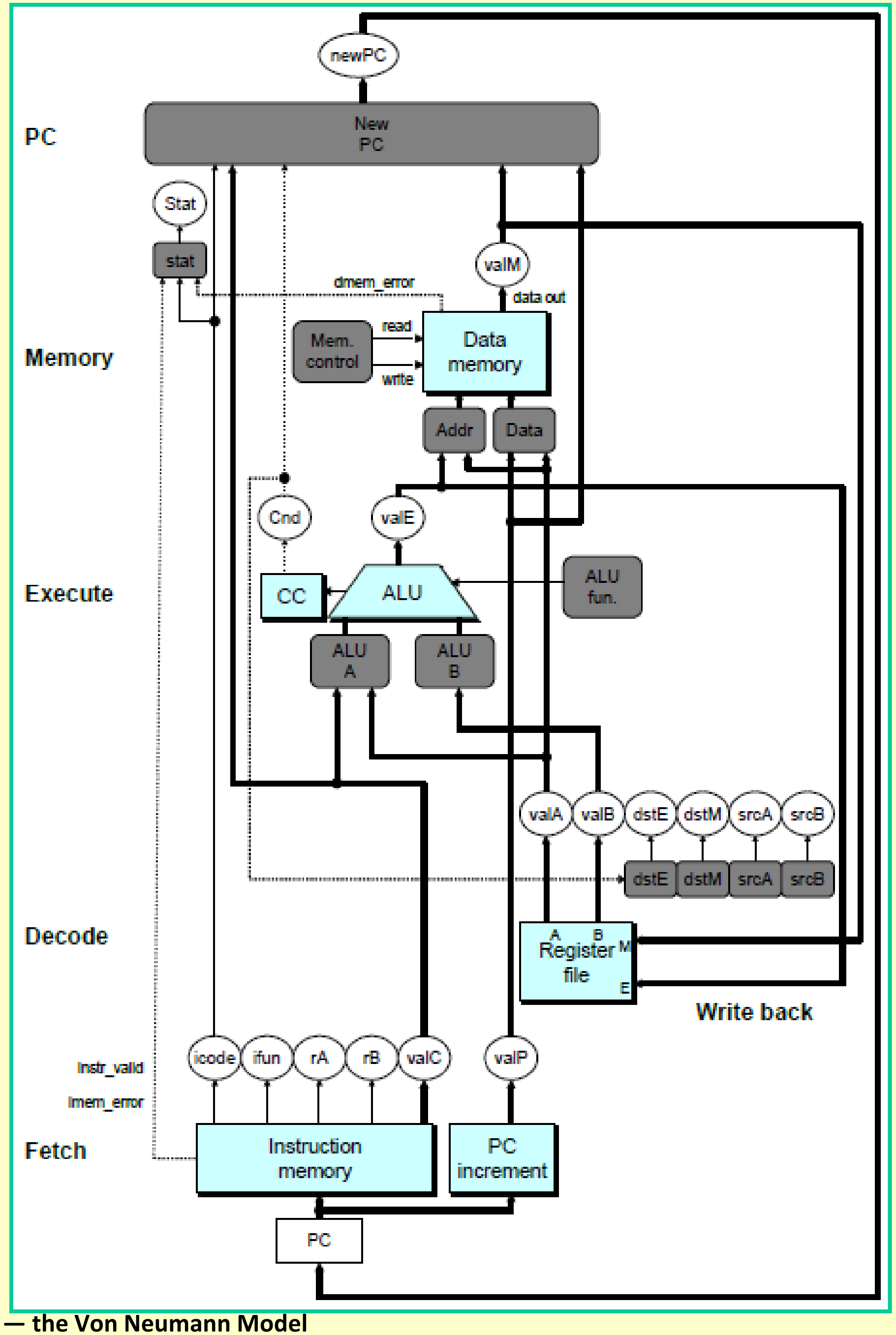
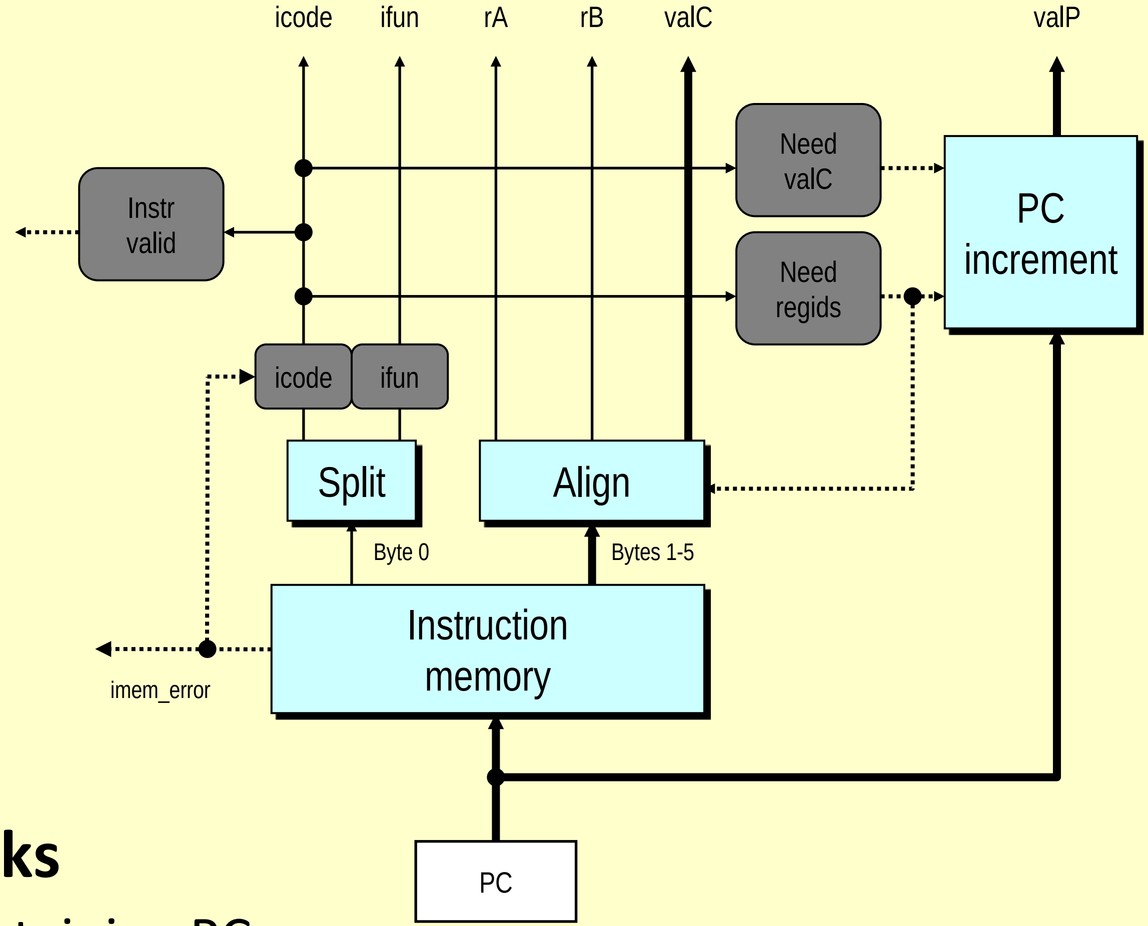


**Figure 4.23, p. 378**

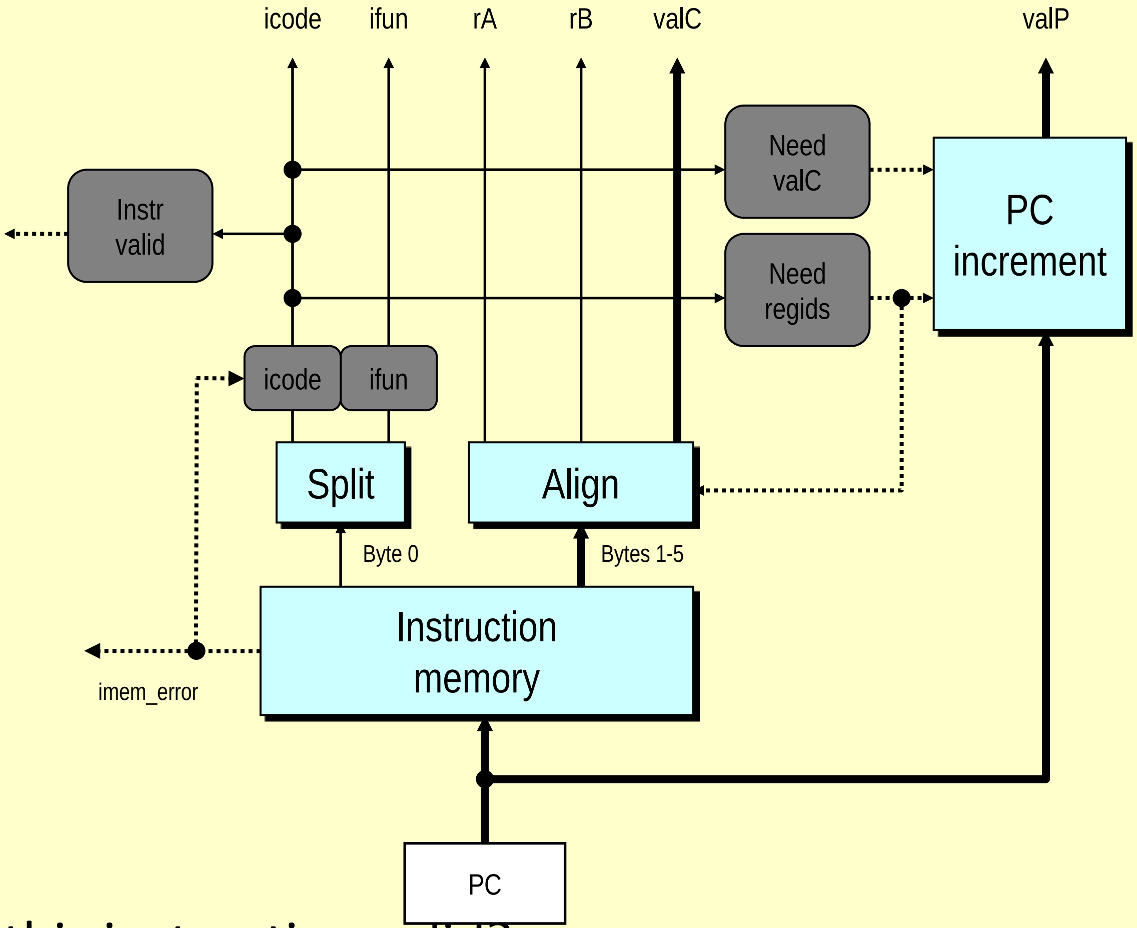
# Fetch Logic



## Predefined Blocks

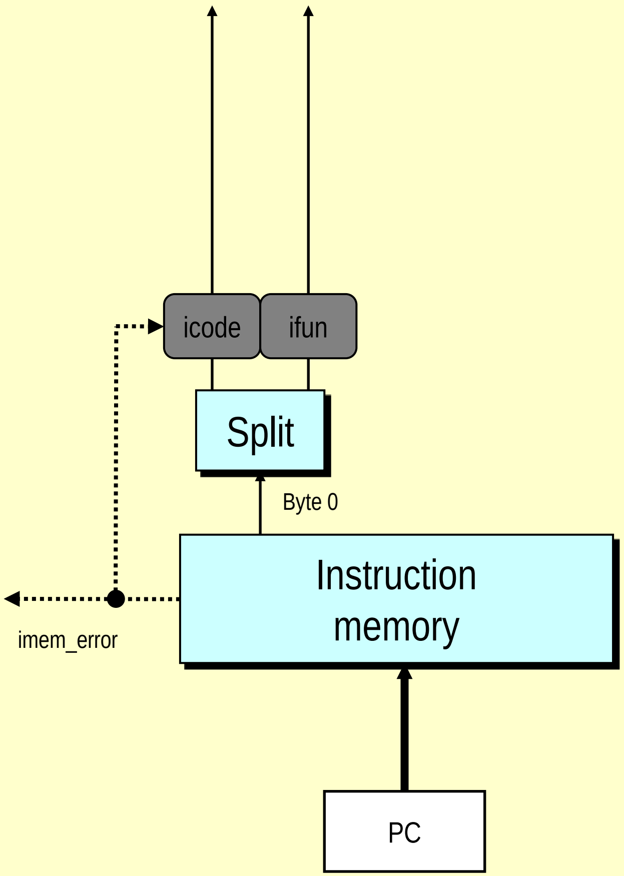
- PC: Register containing PC
- Instruction memory: Read 6 bytes (PC to PC+5)
  - Signal invalid address
- Split: Divide instruction byte into icode and ifun
- Align: Get fields for rA, rB, and valC

# Fetch Logic



- **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?

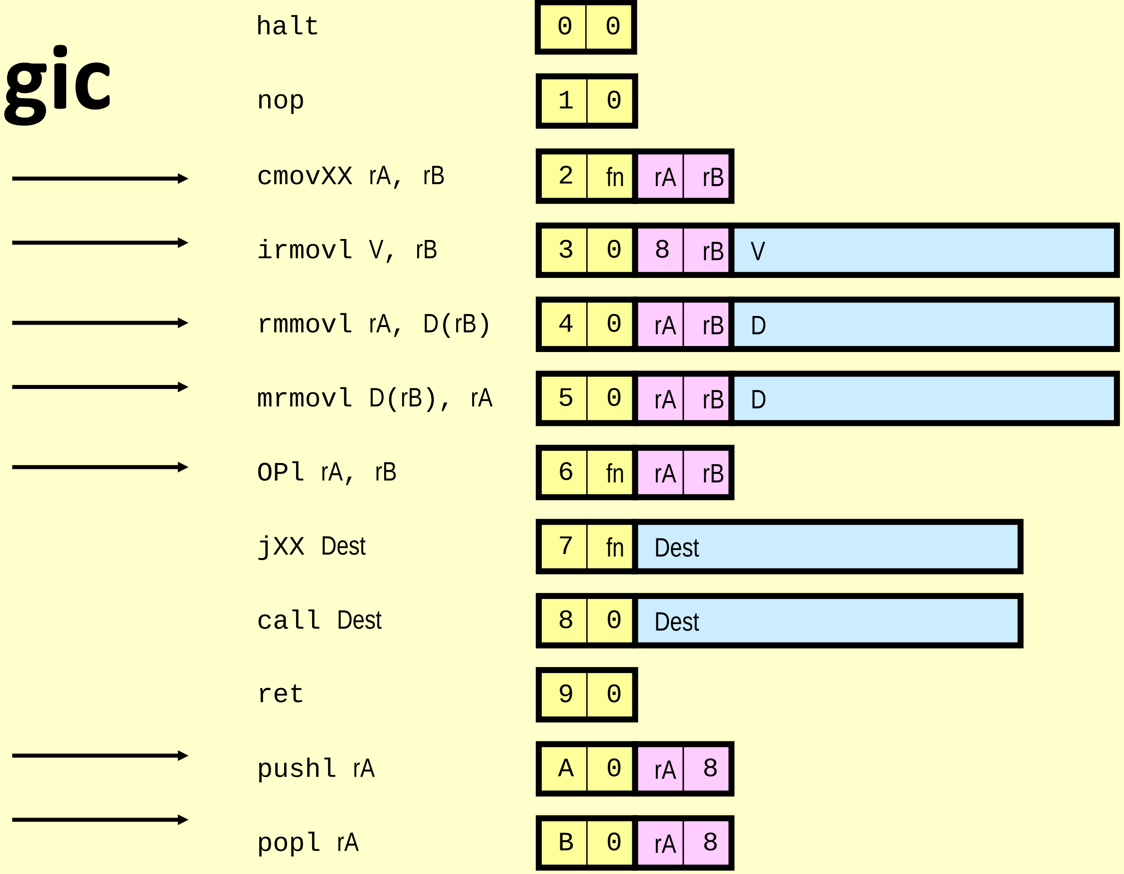
# Fetch Control Logic in HCL



```
# 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



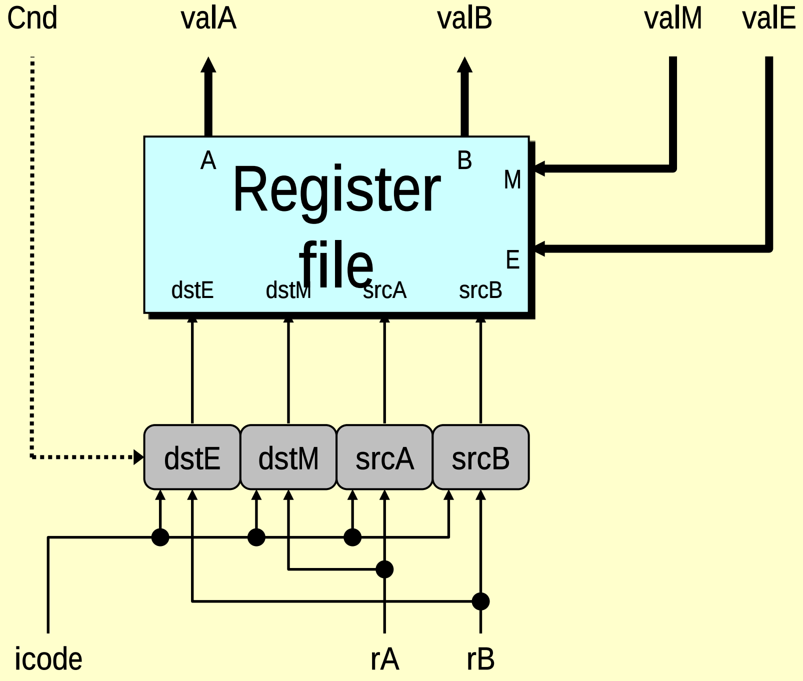
| | | | | | |
|---|---|---|---|---|---|
| halt | 0 | 0 | | | |
| nop | 1 | 0 | | | |
| cmovXX rA, rB | 2 | fn | rA | rB | |
| irmovl V, rB | 3 | 0 | 8 | rB | V |
| rmmovl rA, D(rB) | 4 | 0 | rA | rB | D |
| mrmovl D(rB), rA | 5 | 0 | rA | rB | D |
| OPl rA, rB | 6 | fn | rA | rB | |
| jXX Dest | 7 | fn | Dest | | |
| call Dest | 8 | 0 | Dest | | |
| ret | 9 | 0 | | | |
| pushl rA | A | 0 | rA | 8 | |
| popl rA | B | 0 | rA | 8 | |

```
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 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

| | | |
|---|---|---|
| | OPl rA, rB | |
| Decode | valA ← R[rA] | Read operand A |
| | cmovXX rA, rB | |
| Decode | valA ← R[rA] | Read operand A |
| | `rmmovl` rA, D(rB) | |
| Decode | valA ← R[rA] | Read operand A |
| | `popl` rA | |
| Decode | valA ← R[`%esp`] | Read stack pointer |
| | jXX Dest | |
| Decode | | No operand |
| | `call` Dest | |
| Decode | | No operand |
| | `ret` | |
| Decode | valA ← R[`%esp`] | Read stack pointer |

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

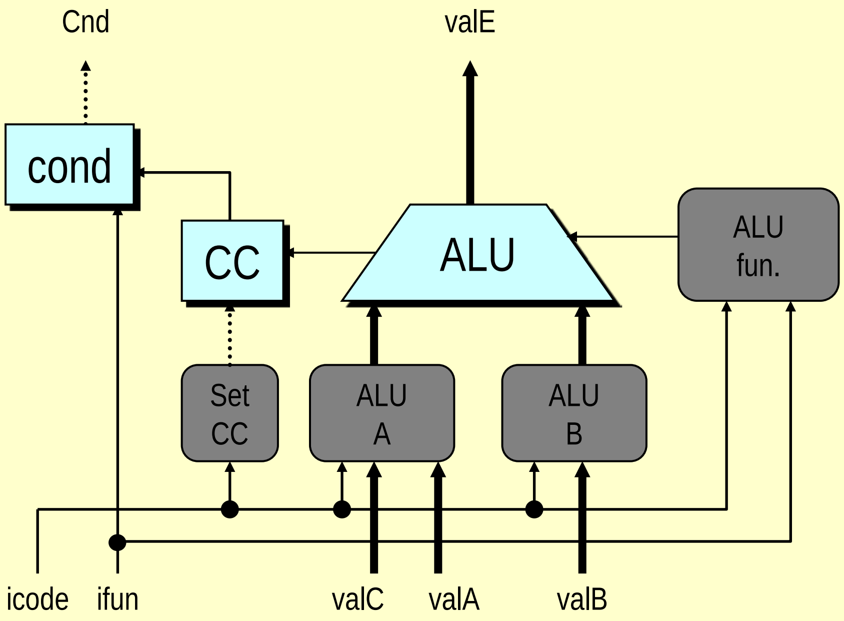
# E Destination

| | | |
|---|---|---|
| | OPl rA, rB | |
| Write-back | R[rB] ← valE | Write back result |
| | cmovXX rA, rB | |
| Write-back | R[rB] ← valE | Conditionally write back result |
| | **`rmmovl`** rA, D(rB) | |
| Write-back | | None |
| | **`popl`** rA | |
| Write-back | R[**`%esp`**] ← valE | Update stack pointer |
| | jXX Dest | |
| Write-back | | None |
| | **`call`** Dest | |
| Write-back | R[**`%esp`**] ← valE | Update stack pointer |
| | **`ret`** | |
| Write-back | R[**`%esp`**] ← valE | Update stack pointer |

```
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

| | | |
|---|---|---|
| | 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 |
| | `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 |

| | cmovXX rA, rB | |
|---|---|---|
| Execute | valE ← 0 + valA | Pass valA through ALU |

| | `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 |

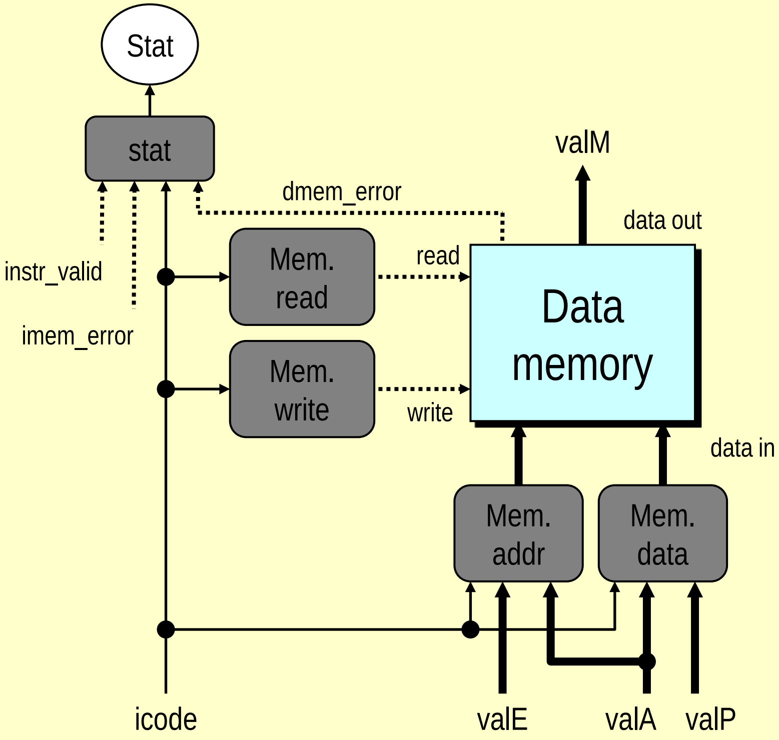
| | `call` Dest | |
|---|---|---|
| Execute | valE ← valB + –4 | Decrement stack pointer |

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

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

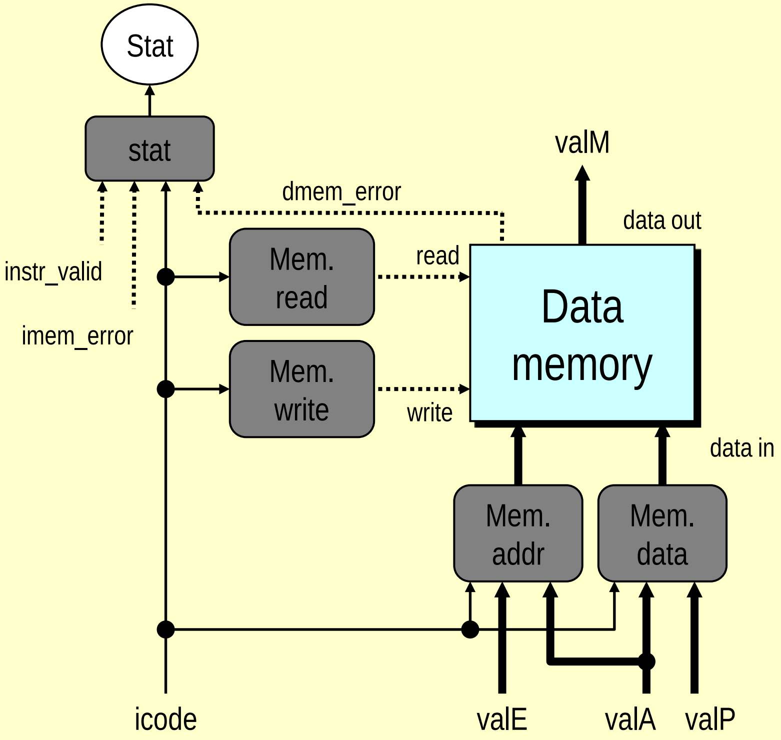
# Memory Logic

- **Memory**
  - Reads or writes memory word
- **Control Logic**
  - 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

| | | |
|---|---|---|
| | 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
];
```

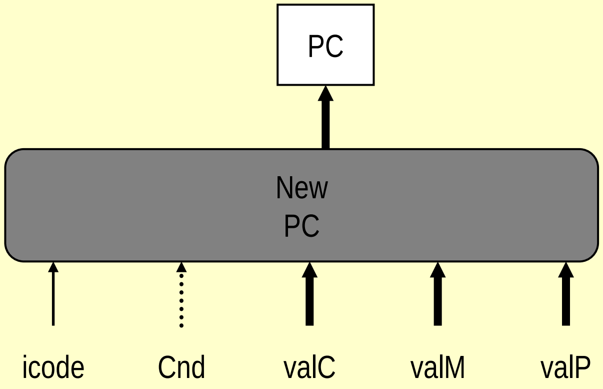
# Memory Read

| | | |
|---|---|---|
| | 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 |

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

# PC Update Logic

- **New PC**
  - Select next value of PC

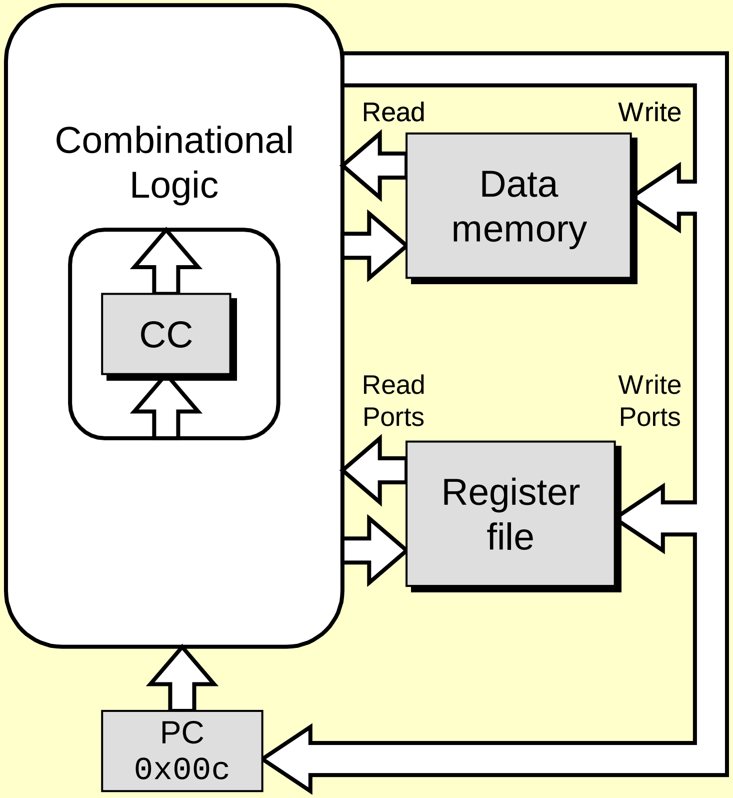


# PC Update

| | | |
|---|---|---|
| | 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 ← 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;
];
```

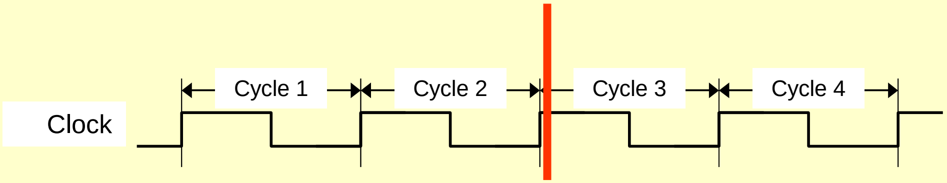
# Sequential Operation



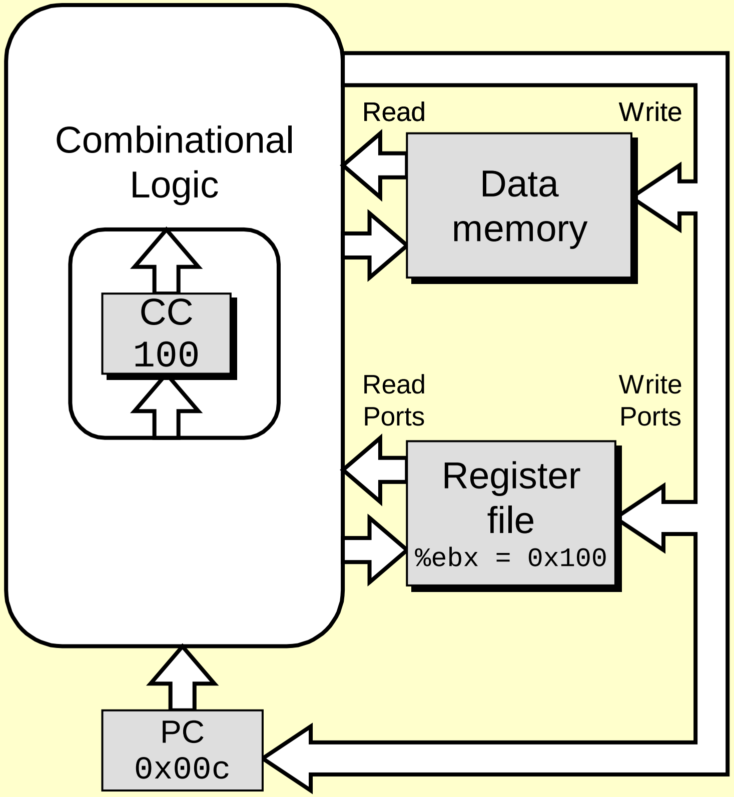
- **State**
  - PC register
  - Cond. Code register
  - Data memory
  - Register file

  *All updated as clock rises*
- **Combinational Logic**
  - ALU
  - Control logic
  - Memory reads
    - Instruction memory
    - Register file
    - Data memory

# SEQ Operation #2

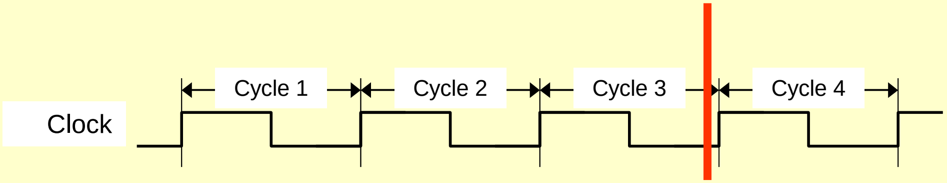


| | | | |
|---|---|---|---|
| Cycle 1: | 0x000: | irmovl $0x100,%ebx | # %ebx <-- 0x100 |
| Cycle 2: | 0x006: | irmovl $0x200,%edx | # %edx <-- 0x200 |
| Cycle 3: | 0x00c: | addl %edx,%ebx | # %ebx <-- 0x300 CC <-- 000 |
| Cycle 4: | 0x00e: | je dest | # Not taken |

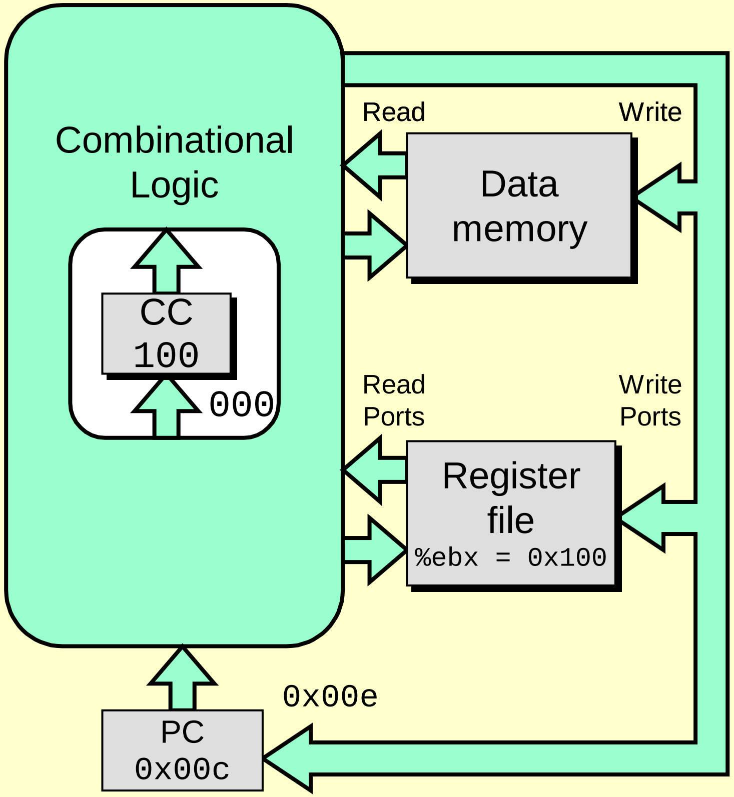


- state set according to second `irmovl` instruction
- combinational logic starting to react to state changes

# SEQ Operation #3

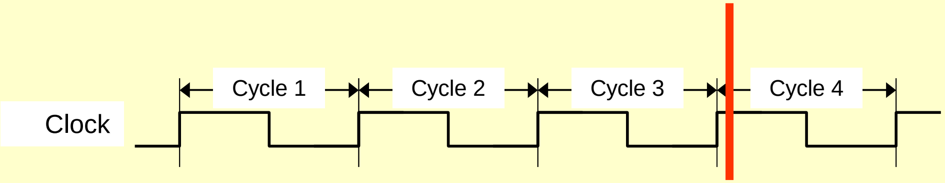


```
Cycle 1:  0x000:  irmovl $0x100,%ebx  # %ebx <-- 0x100
Cycle 2:  0x006:  irmovl $0x200,%edx  # %edx <-- 0x200
Cycle 3:  0x00c:  addl %edx,%ebx      # %ebx <-- 0x300 CC <-- 000
Cycle 4:  0x00e:  je dest             # Not taken
```

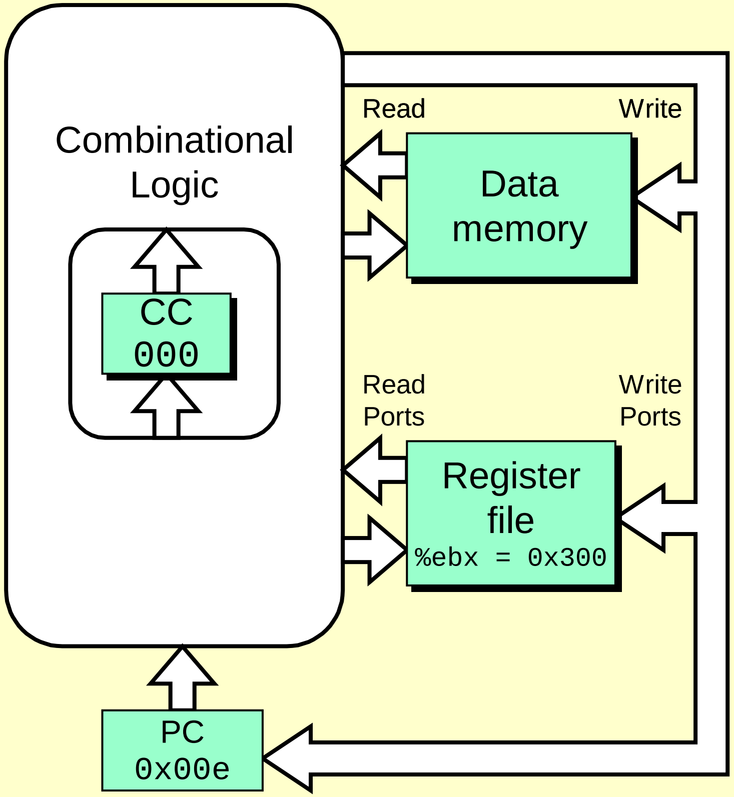


- state set according to second `irmovl` instruction
- combinational logic generates results for `addl` instruction

# SEQ Operation #4

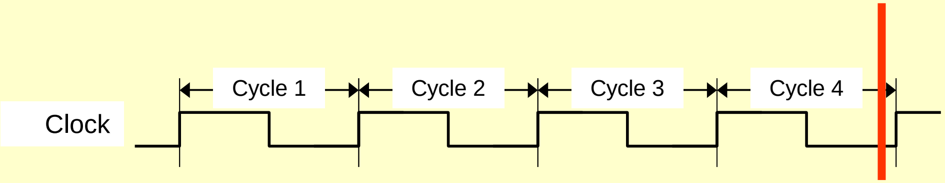


```
Cycle 1:  0x000:  irmovl $0x100,%ebx  # %ebx <-- 0x100
Cycle 2:  0x006:  irmovl $0x200,%edx  # %edx <-- 0x200
Cycle 3:  0x00c:  addl %edx,%ebx      # %ebx <-- 0x300 CC <-- 000
Cycle 4:  0x00e:  je dest             # Not taken
```

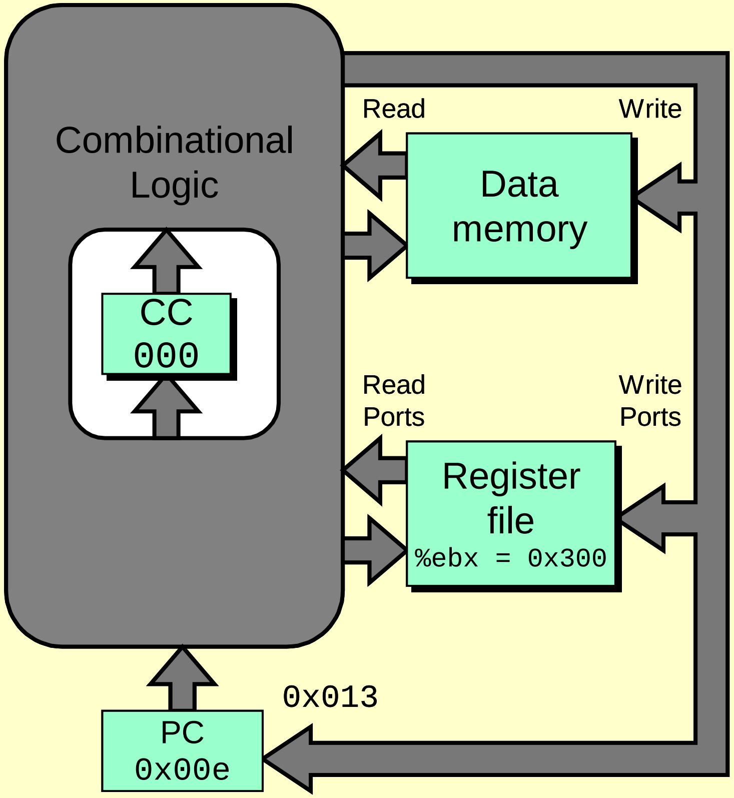


- state set according to `addl` instruction
- combinational logic starting to react to state changes

# SEQ Operation #5



```
Cycle 1:  0x000:   irmovl $0x100,%ebx   # %ebx <-- 0x100
Cycle 2:  0x006:   irmovl $0x200,%edx   # %edx <-- 0x200
Cycle 3:  0x00c:   addl %edx,%ebx       # %ebx <-- 0x300 CC <-- 000
Cycle 4:  0x00e:   je dest              # Not taken
```



- 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

# Questions?