18-600 Foundations of Computer Systems

Lecture 7: "Processor Architecture & Design"

John P. Shen & Gregory Kesden

September 20, 2017

Lecture #7 – Processor Architecture & Design
Lecture #8 – Pipelined Processor Design
Lecture #9 – Superscalar O3 Processor Design

- Required Reading Assignment:
  - **Chapter 4 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**
- Recommended Reference:
  - **Chapters 1 and 2 of Shen and Lipasti (SnL).**

Electrical & Computer ENGINEERING

Carnegie Mellon University

Computer Systems (ISA & DSI)

PROGRAM

"static"

ARCHITECTURE — Instruction Set Architecture (ISA)

"dynamic"

MACHINE

Instruction Set Definition:
- Architecture State: Reg & Memory
- Op-code & Operand types
- Operand Addressing modes
- Control Flow instructions

Program Execution:
- Load program into Memory
- Fetch instructions from Memory
- Execute Instructions in CPU
- Update Architecture State

Intel x86

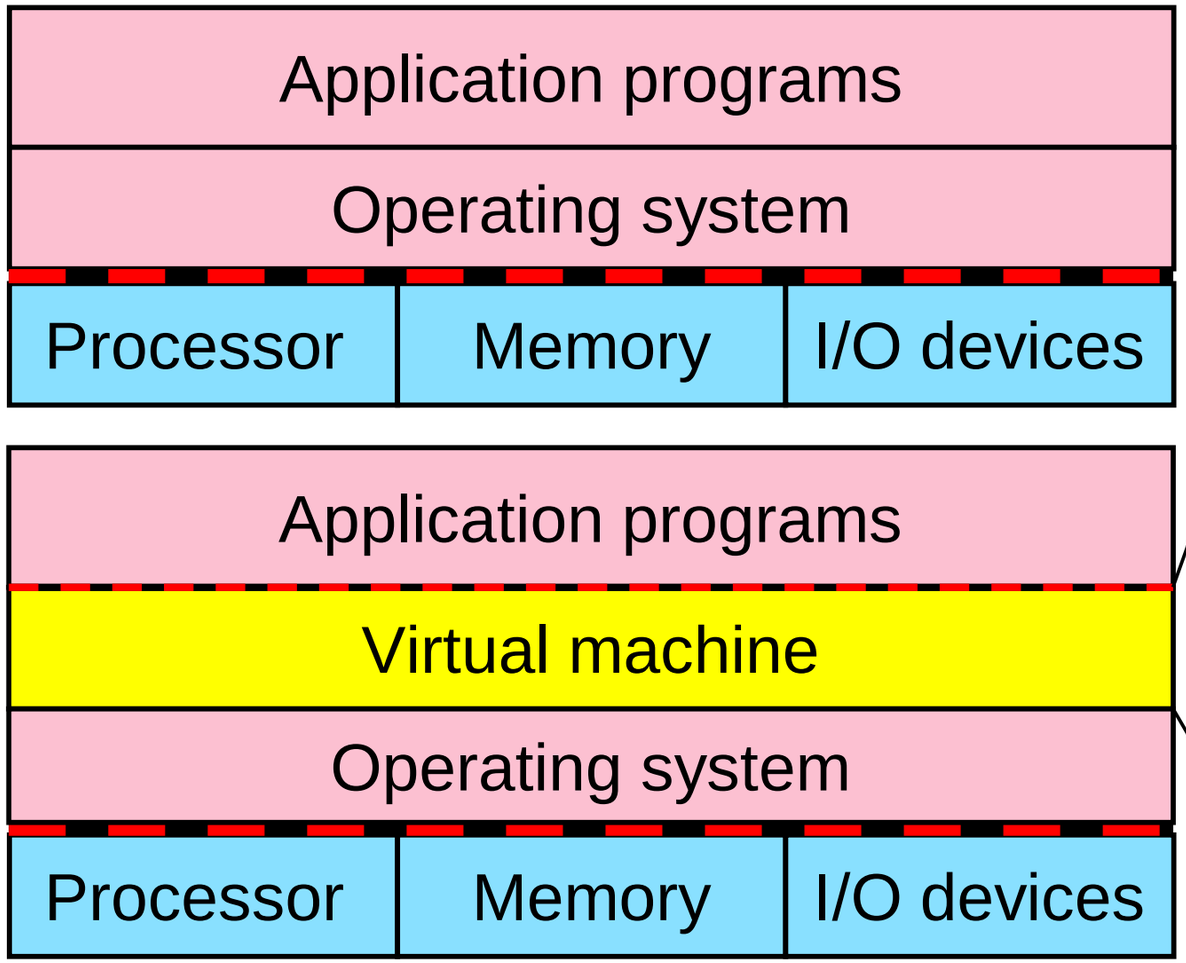
ARM Architecture

MIPS TECHNOLOGIES

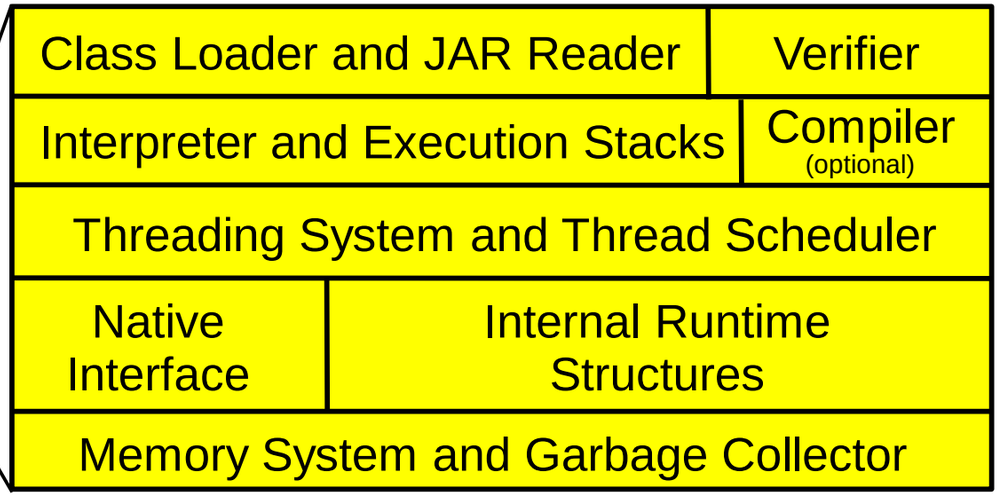
SPARC

PowerPC

Java Programming & Java Virtual Machine



"Will be back!"



18-600 Foundations of Computer Systems

Lecture 7: "Processor Architecture & Design"

1. **Processor Architecture**
   a. Instruction Set Architecture (ISA)
   b. Y86-64 Instruction Set Architecture
   c. Logic Design Revisited/Simplified
2. **Processor Implementation**
   a. Instruction Set Processor (CPU)
   b. Processor Organization Design
   c. Y86-64 Sequential Processor Design
3. **Motivation for Pipelining**

Electrical & Computer ENGINEERING

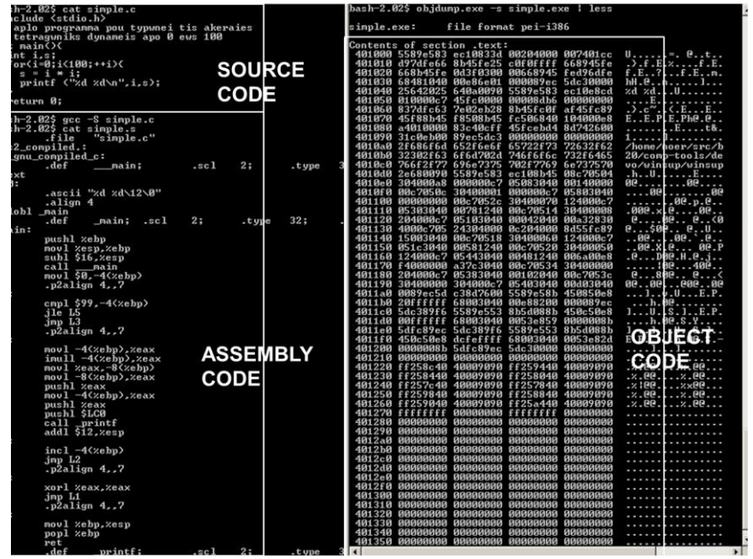
Carnegie Mellon University

Computer Systems (SE & CE)

Software Engineering

Program Development

Application Software
- Program development
- Program compilation

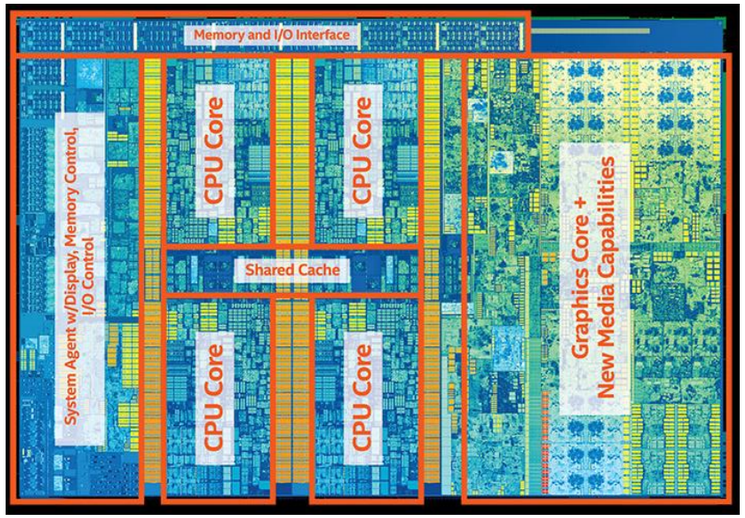


SPECIFICATION — Instruction Set Architecture (ISA)

Computer Engineering

Processor Design

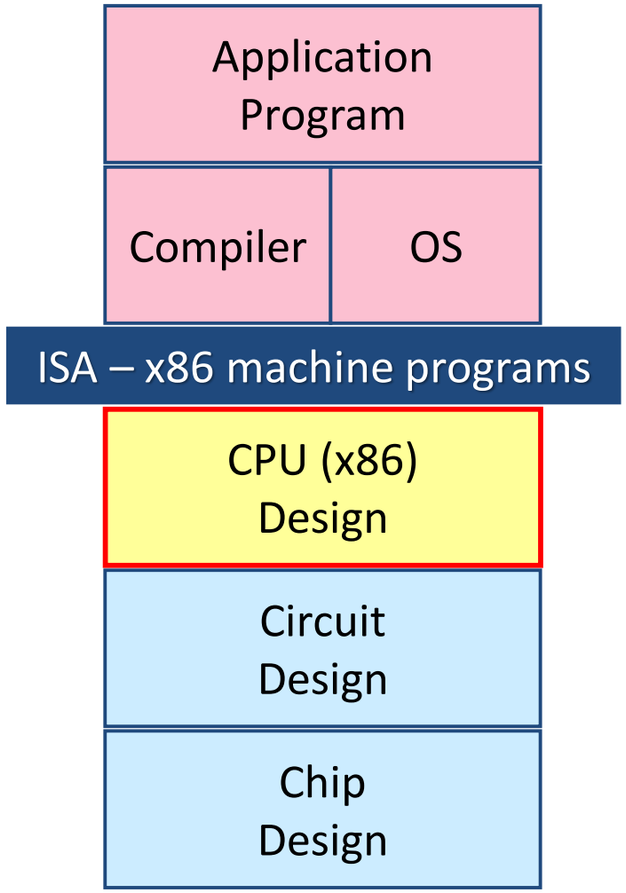
Hardware Technology
- Program execution
- Computer performance



IMPLEMENTATION

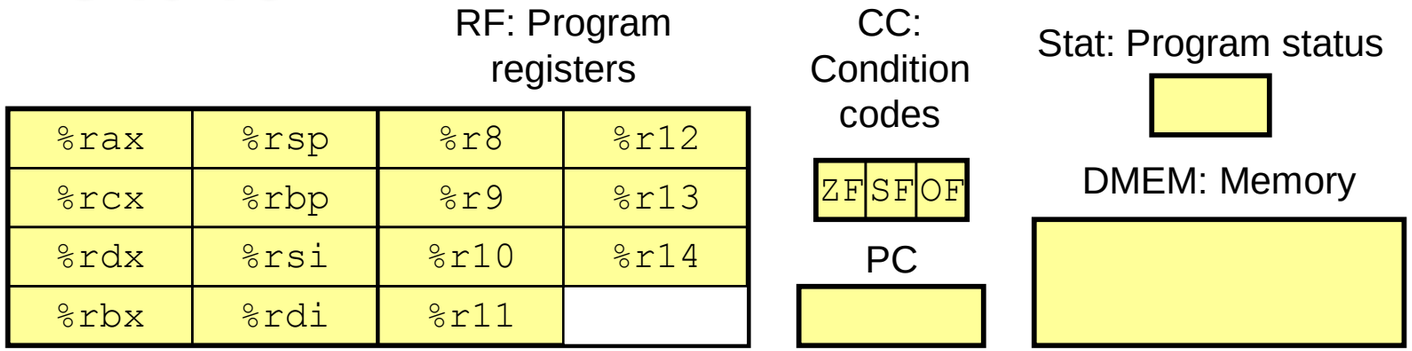
Instruction Set Architecture (x86-64)

- Assembly Language View
  - Processor state (architecture state)
    - Registers, Memory, …
  - Instructions (specify operations, operands)
    - `addq`, `pushq`, `ret`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be implemented
    - Use variety of uarch techniques to make it run fast
    - E.g., execute multiple instructions simultaneously



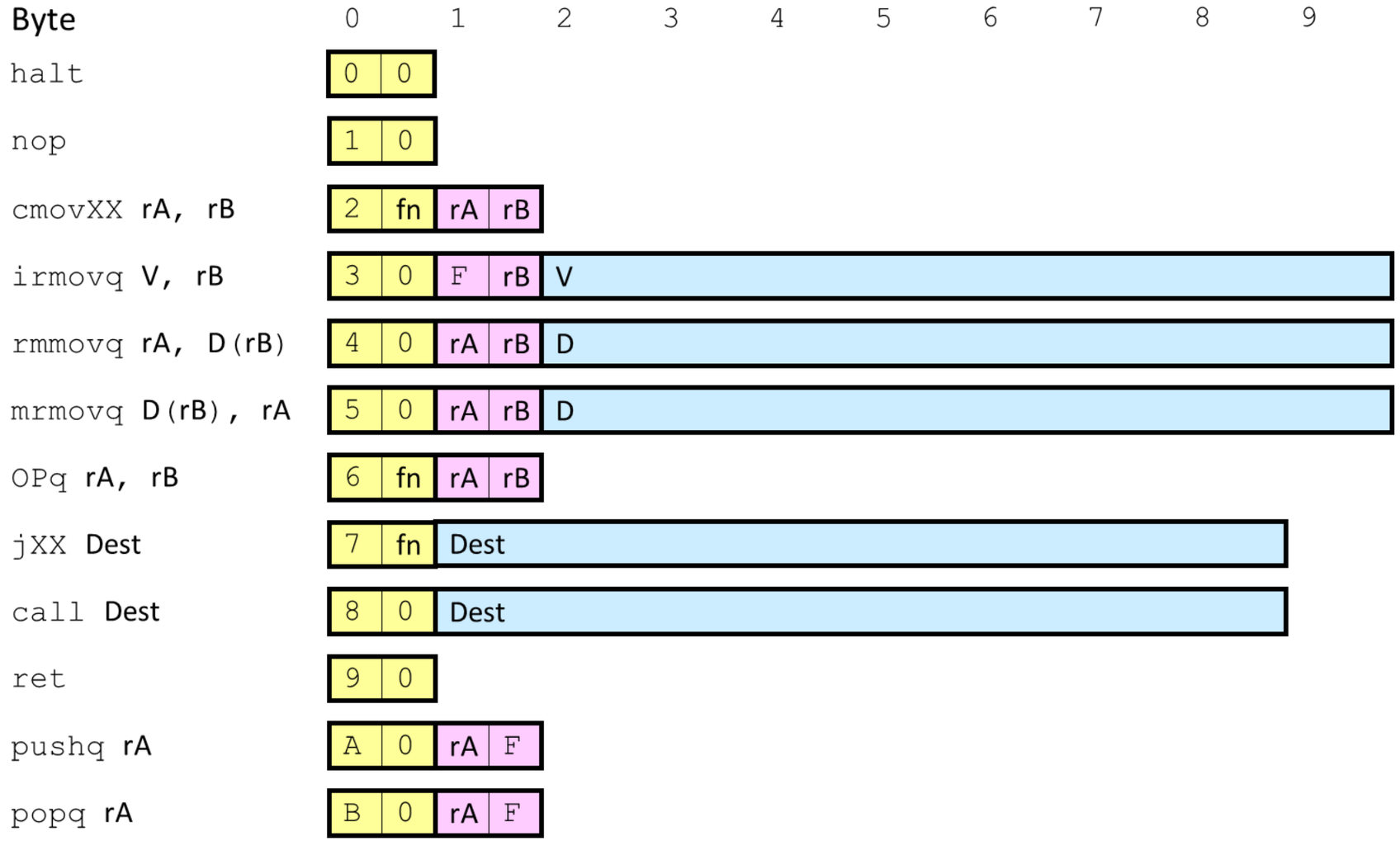
Y86-64 Processor State

- Program Registers
  - 15 registers (omit `%r15`).
    Each 64 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - ZF: Zero SF: Negative OF: Overflow
- Program Counter
  - Indicates address of **next** instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order



Y86-64 Instruction Set Architecture
- Similar state and instructions as x86-64
- Simpler encodings
- Somewhere between CISC and RISC

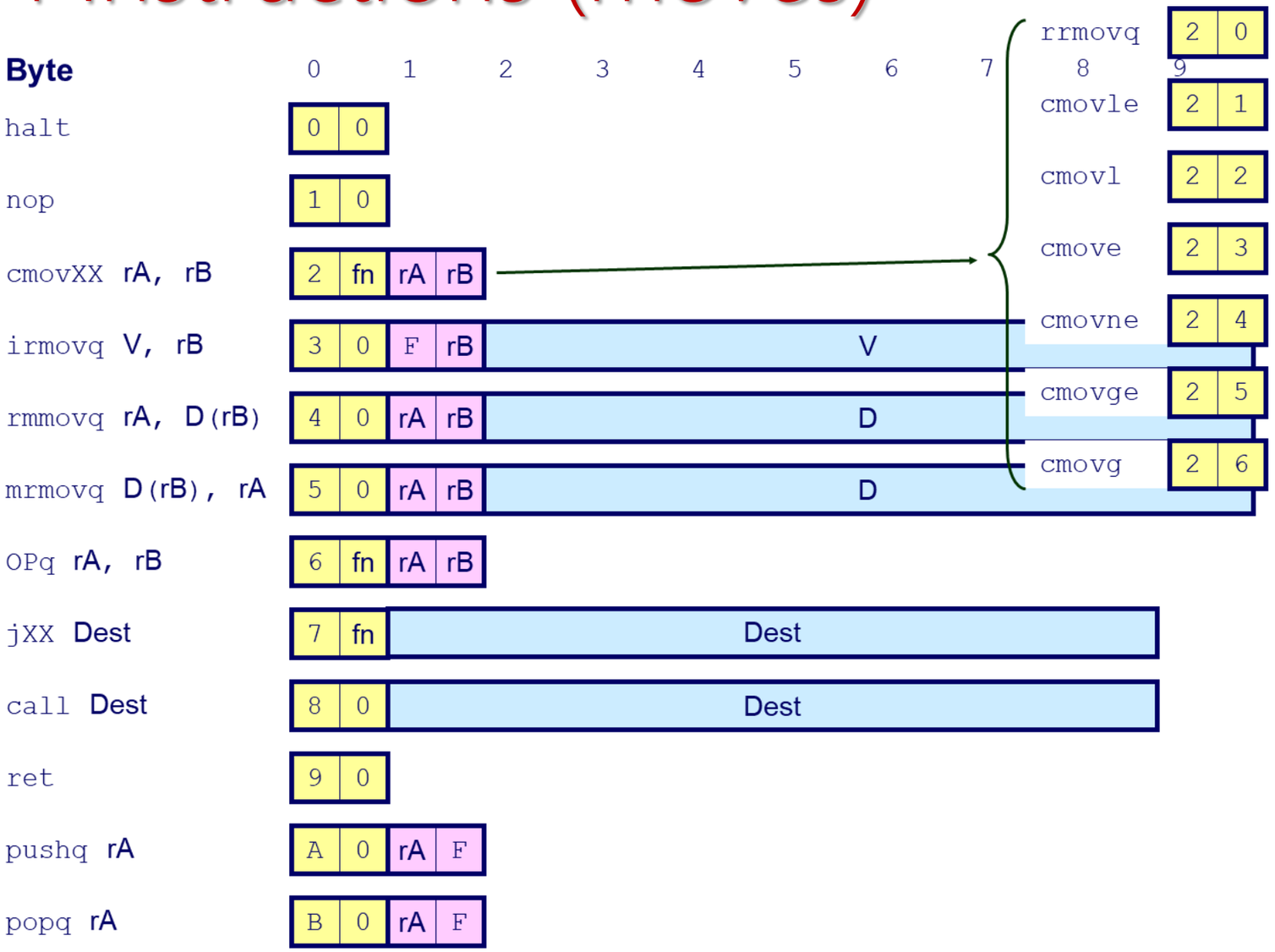
Y86-64 Instruction Set



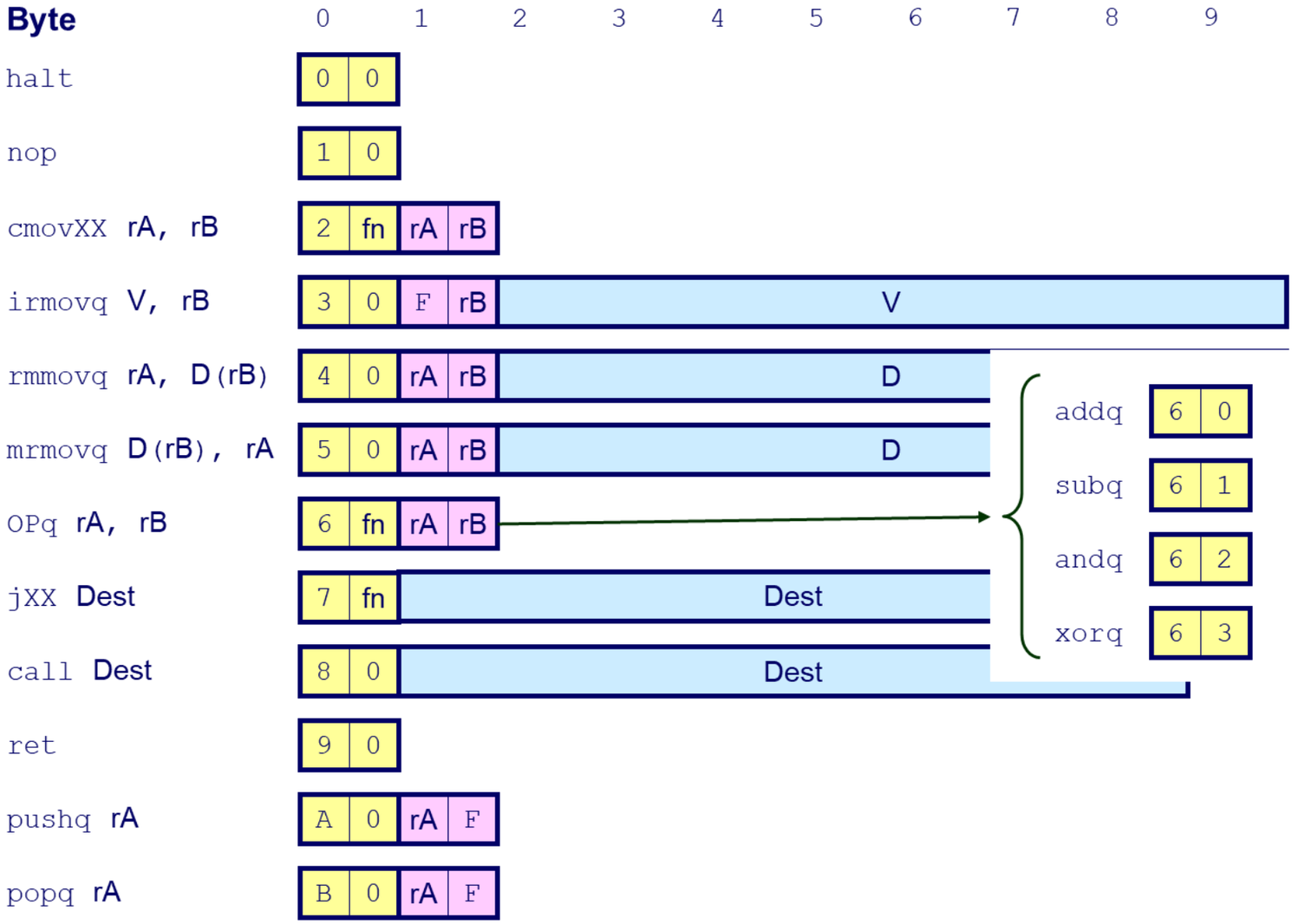
Y86-64 Instruction Formats

- Format
  - 1–10 bytes of information read from memory
    - Can determine instruction length from first byte
    - Fewer instruction types and simpler encoding than x86-64
  - Each instruction accesses and modifies some part(s) of the program state

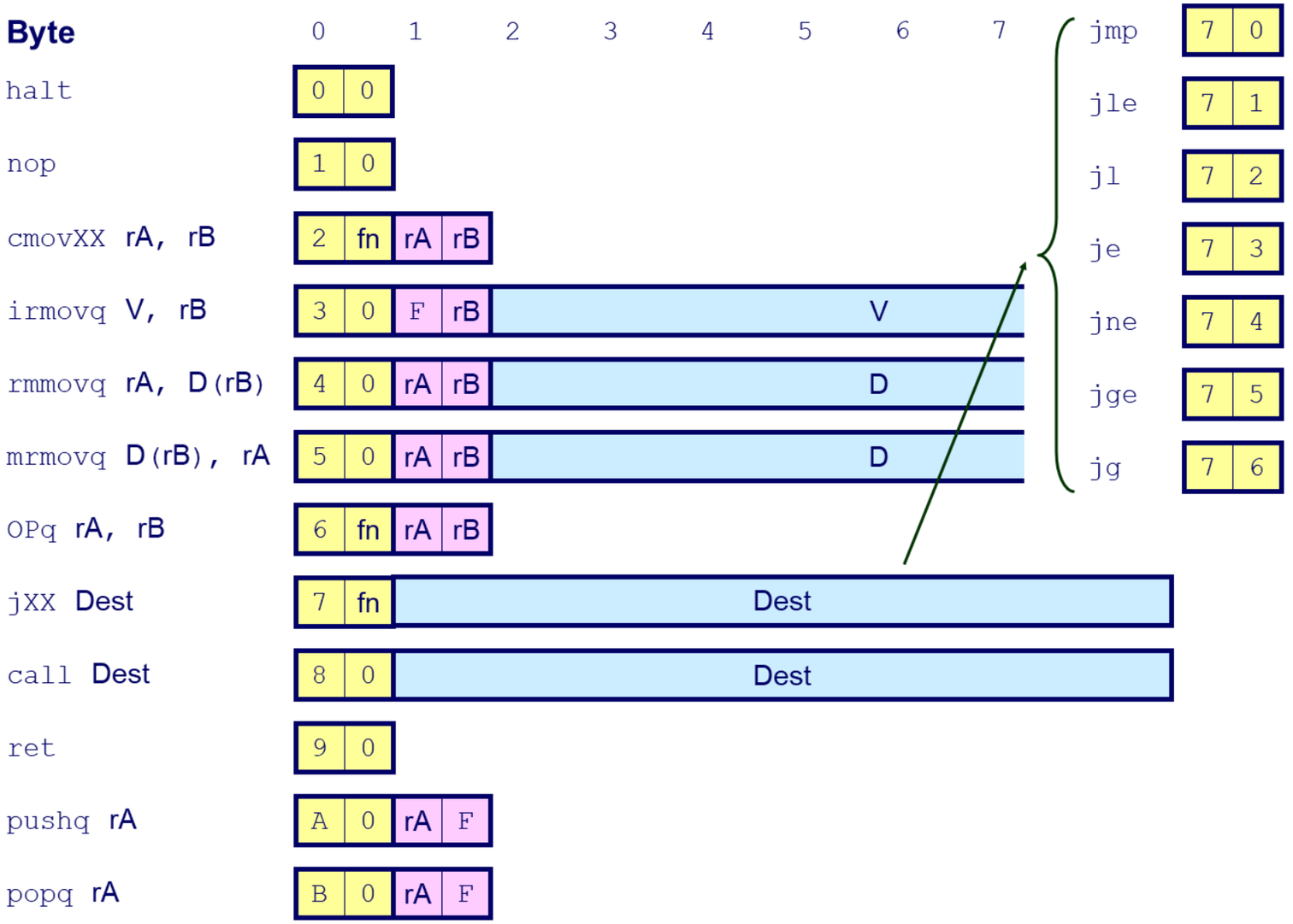
Y86-64 Instructions (moves)



Y86-64 Instructions (ALUs)



Y86-64 Instructions (branches)



Encoding Register Specifiers

- Each register has 4-bit ID

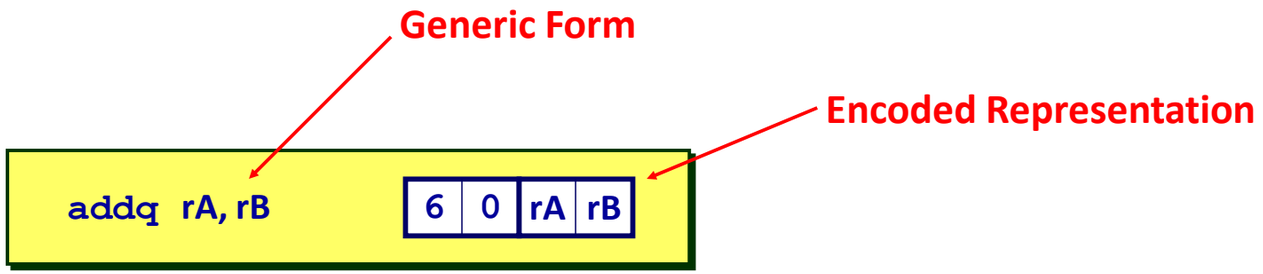
%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7

%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

  - Same encoding as in x86-64
- Register ID 15 (`0xF`) indicates "no register"
  - Will use this in our hardware design in multiple places

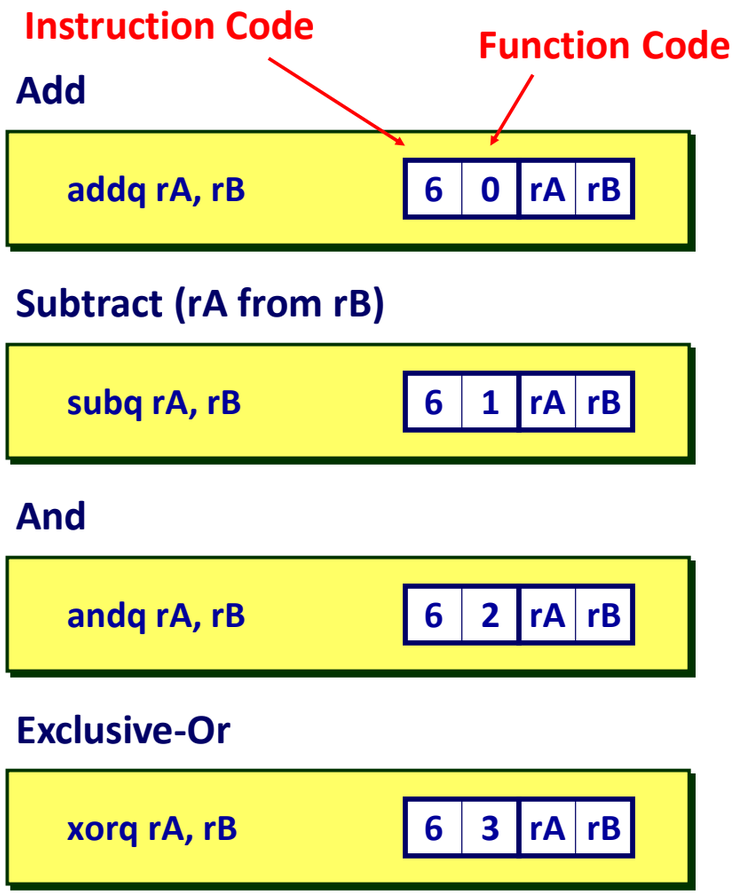
Instruction Format Example

Addition Instruction



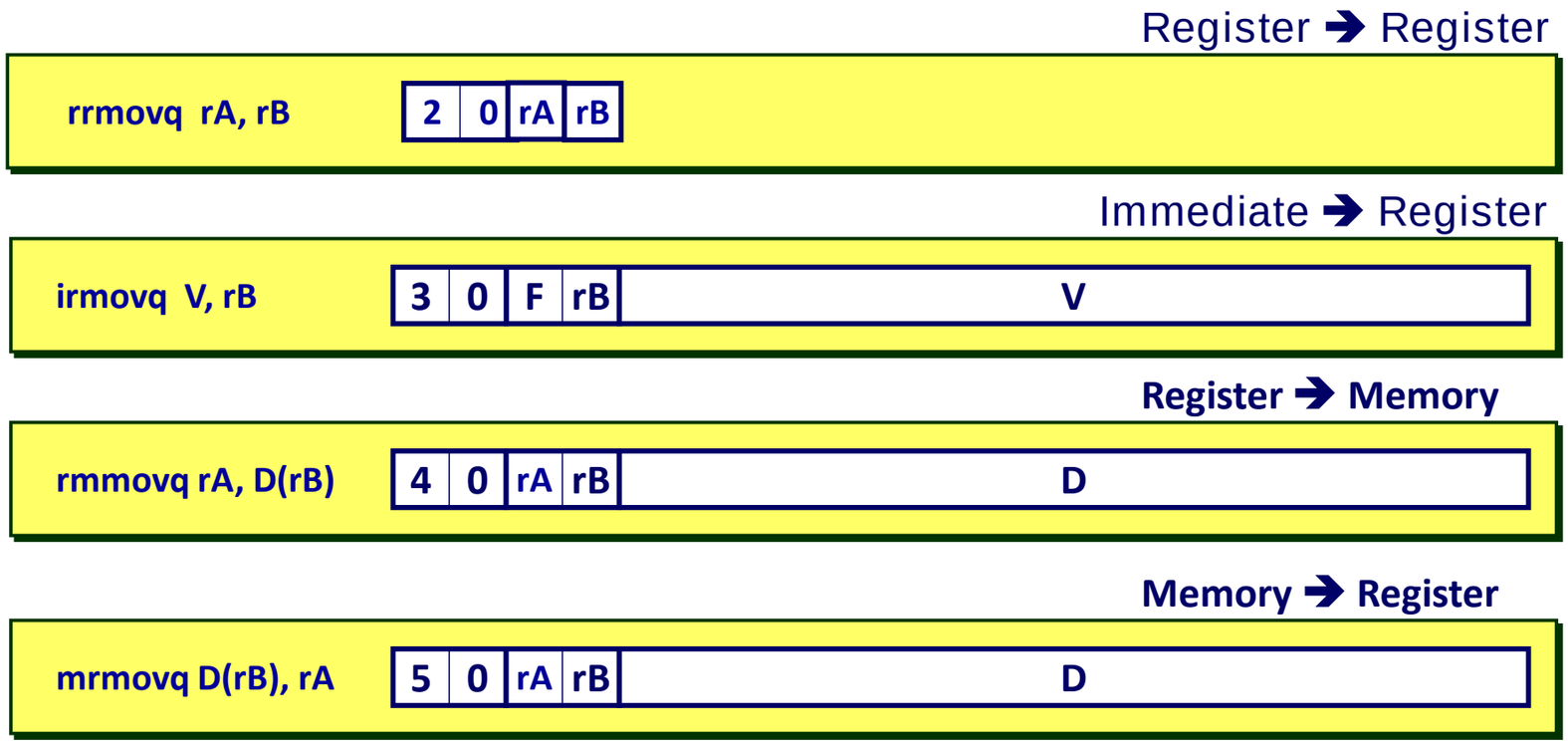
- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- e.g., addq %rax,%rsi Encoding: 60 06
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

Arithmetic and Logical Operations



- Refer to generically as "OPq"
- Encodings differ only by "function code"
  - Low-order 4 bits in first instruction byte
- Set condition codes as side effect

Move Operations

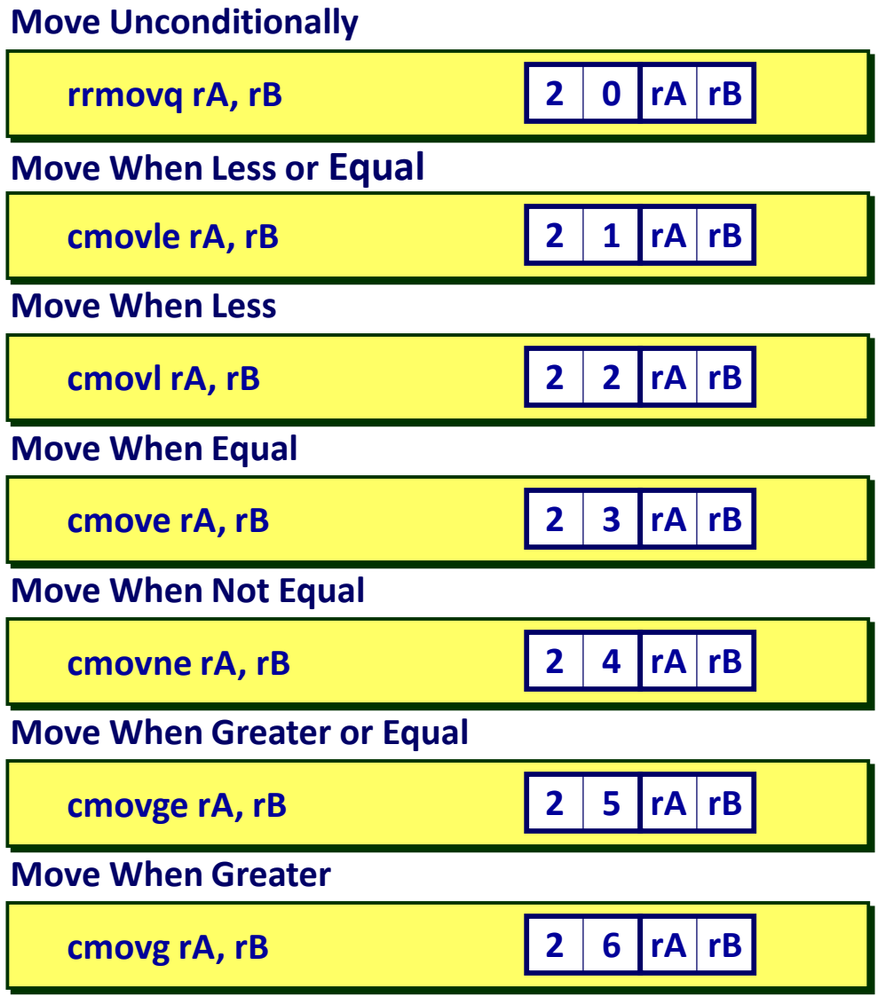


- Like the x86-64 movq instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

X86-64	Y86-64
movq $0xabcd, %rdx	irmovq $0xabcd, %rdx
Encoding:	30 82 cd ab 00 00 00 00 00 00
movq %rsp, %rbx	rrmovq %rsp, %rbx
Encoding:	20 43
movq -12(%rbp),%rcx	mrmovq -12(%rbp),%rcx
Encoding:	50 15 f4 ff ff ff ff ff ff ff
movq %rsi,0x41c(%rsp)	rmmovq %rsi,0x41c(%rsp)
Encoding:	40 64 1c 04 00 00 00 00 00 00

Conditional Move Instructions



- Refer to generically as "cmovXX"
- Encodings differ only by "function code"
- Based on values of condition codes
- Variants of rrmovq instruction
  - (Conditionally) copy value from source to destination register

Jump Instructions

Jump (Conditionally)

- **Refer to generically as "jXX"**
- **Encodings differ only by "function code" fn**
- **Based on values of condition codes**
- **Same as x86-64 counterparts**
- **Encode full destination address**
  - **Unlike PC-relative addressing seen in x86-64**

Jump Instructions

Jump Unconditionally

Jump When Less or Equal

Jump When Less

Jump When Equal

Jump When Not Equal

Jump When Greater or Equal

Jump When Greater

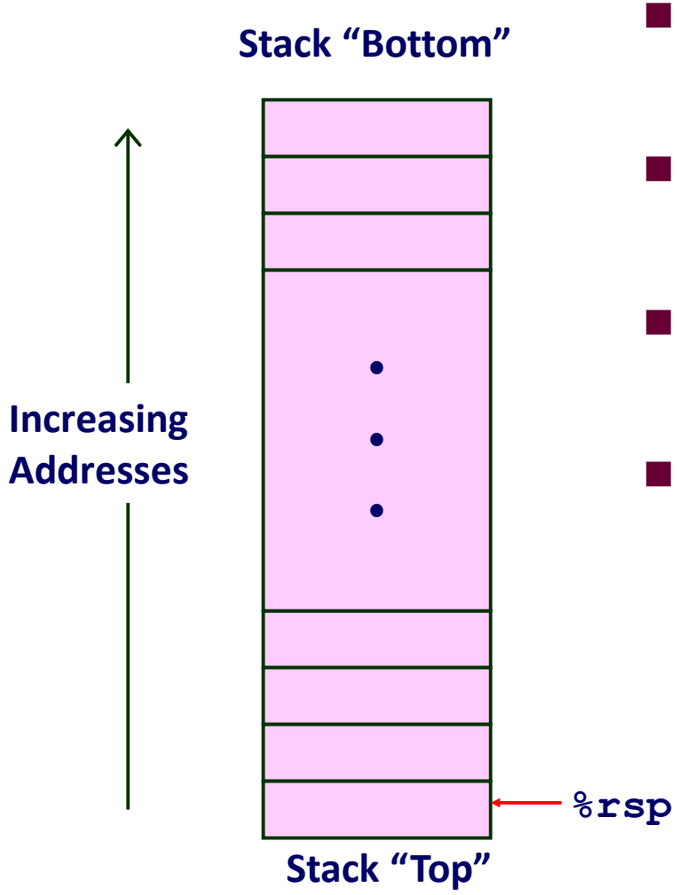
Stack Operations

pushq rA | A | 0 | rA | F |

- **Decrement %rsp by 8**
- **Store word from rA to memory at %rsp**
- **Like x86-64**

popq rA | B | 0 | rA | F |

- **Read word from memory at %rsp**
- **Save in rA**
- **Increment %rsp by 8**
- **Like x86-64**



- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by **%rsp**
  - Address of top stack element
- Stack grows toward lower addresses
  - Bottom element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer

Subroutine Call and Return

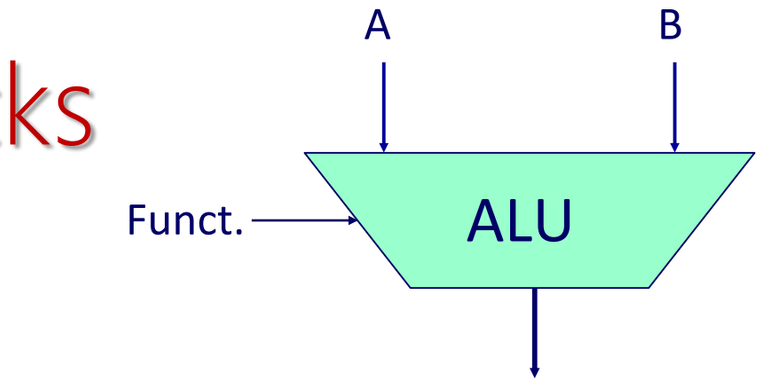
- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64

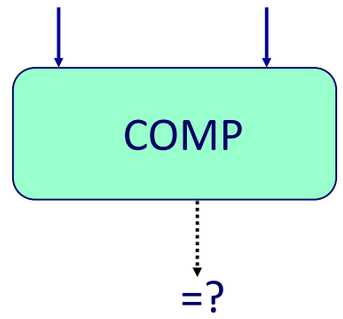
- Pop value from stack
- Use as address for next instruction
- Like x86-64

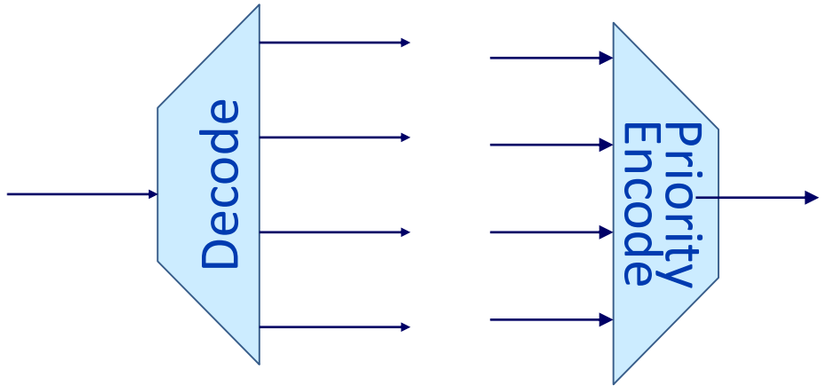
HW Building Blocks

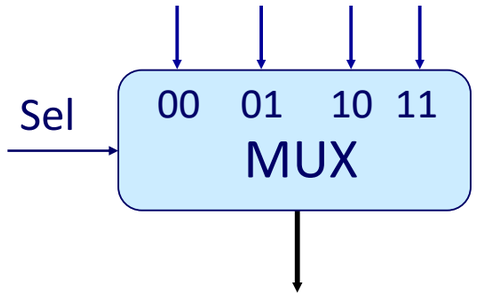
Combinational Logic

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



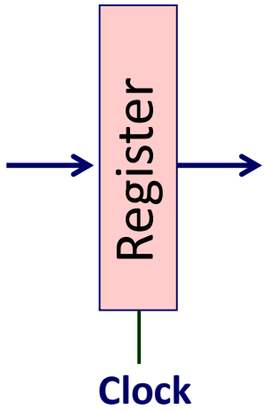


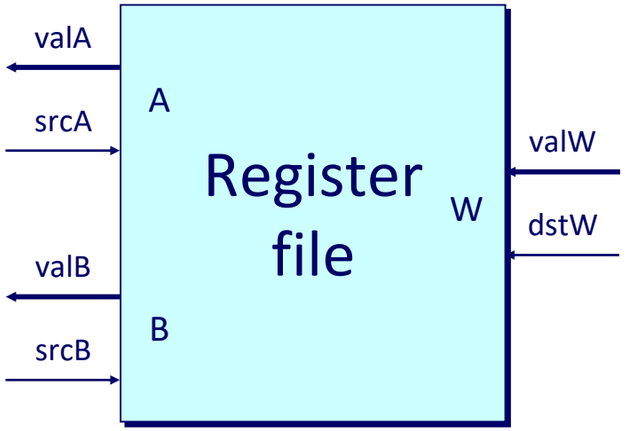




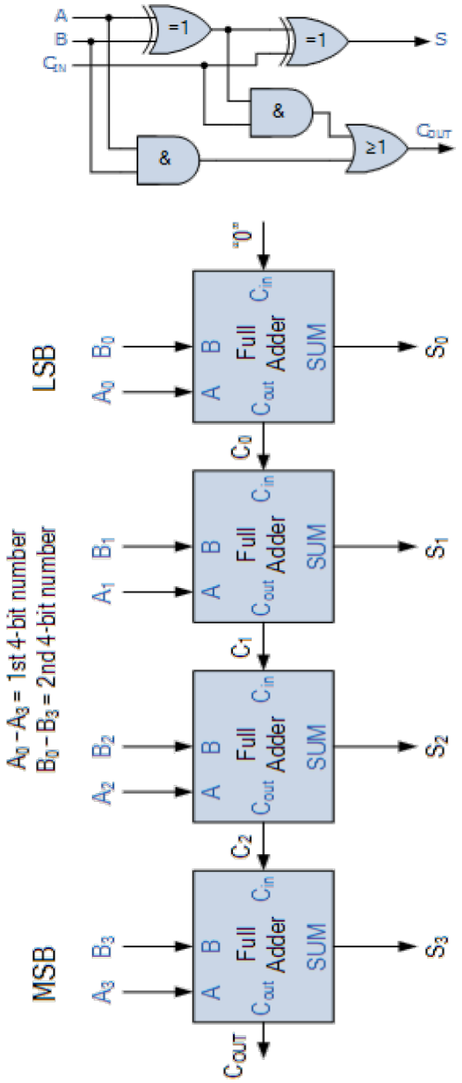
Storage (Sequential) Elements

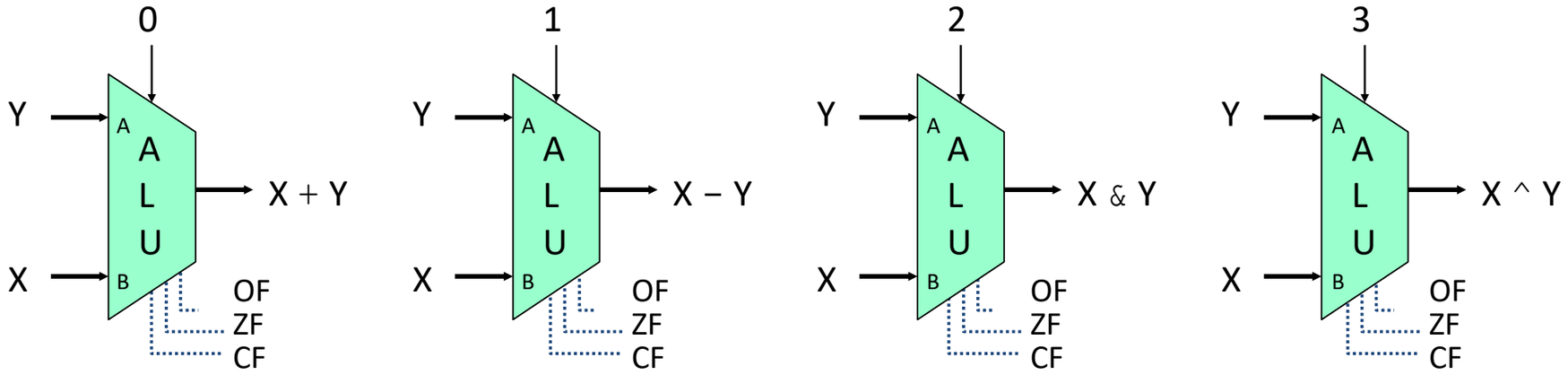
- Store bits
- Addressable memories
- Non-addressable registers
- Loaded only as clock rises





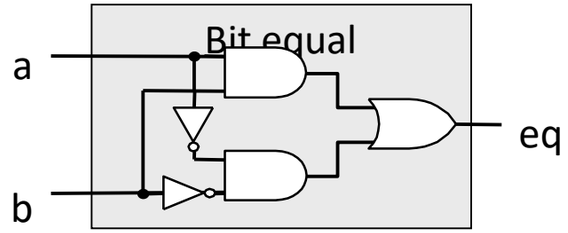
Arithmetic Logic Unit

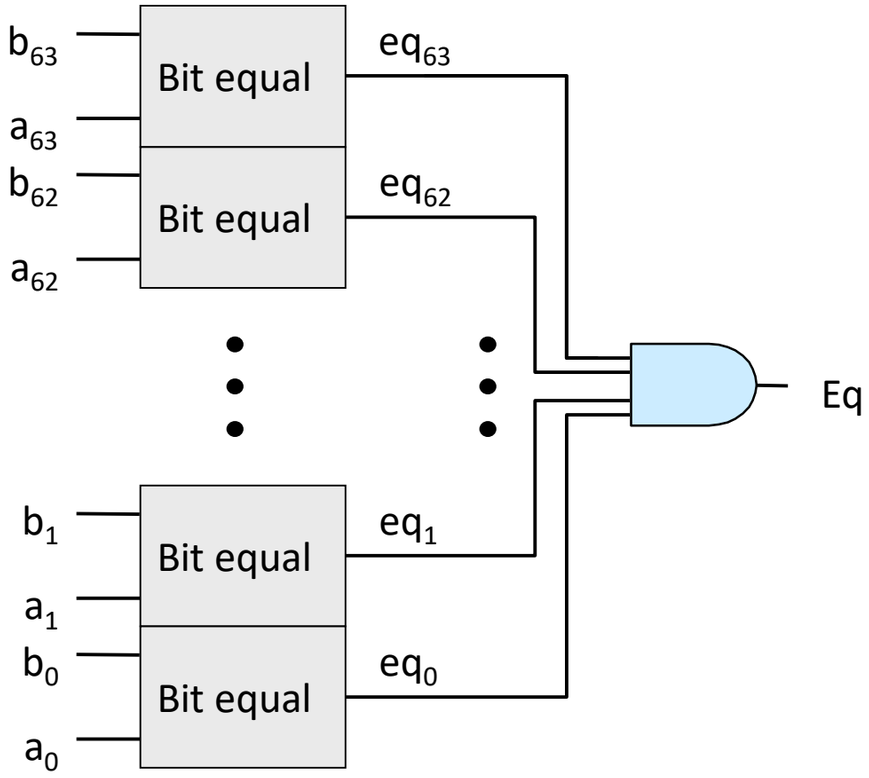


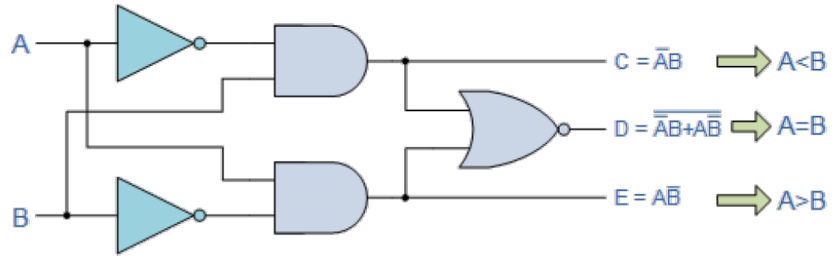


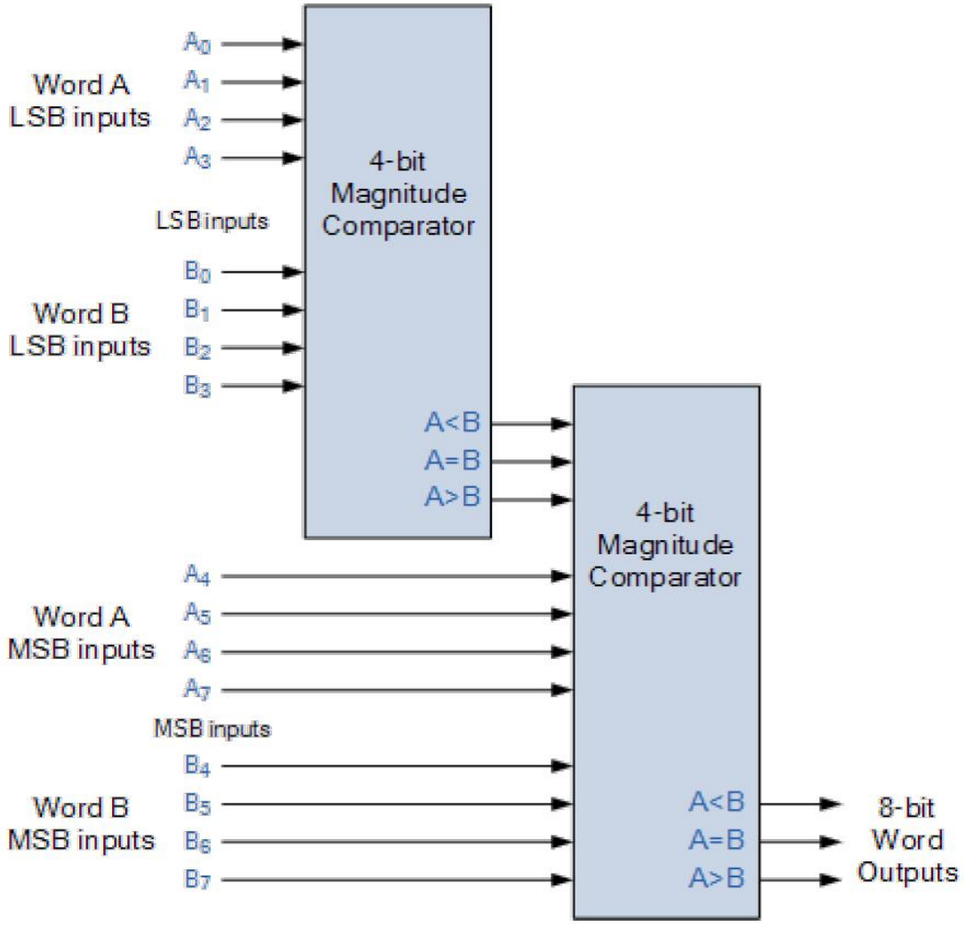
- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86-64
- Also computes values for condition codes

Equality Comparator

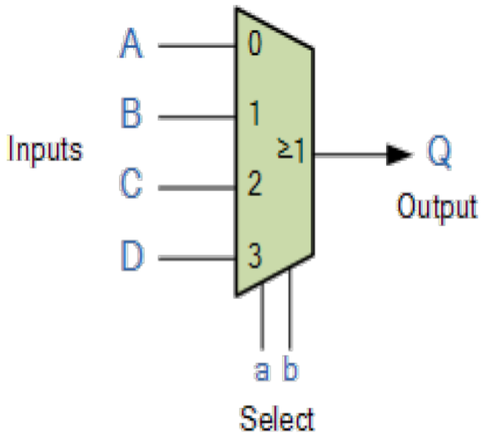


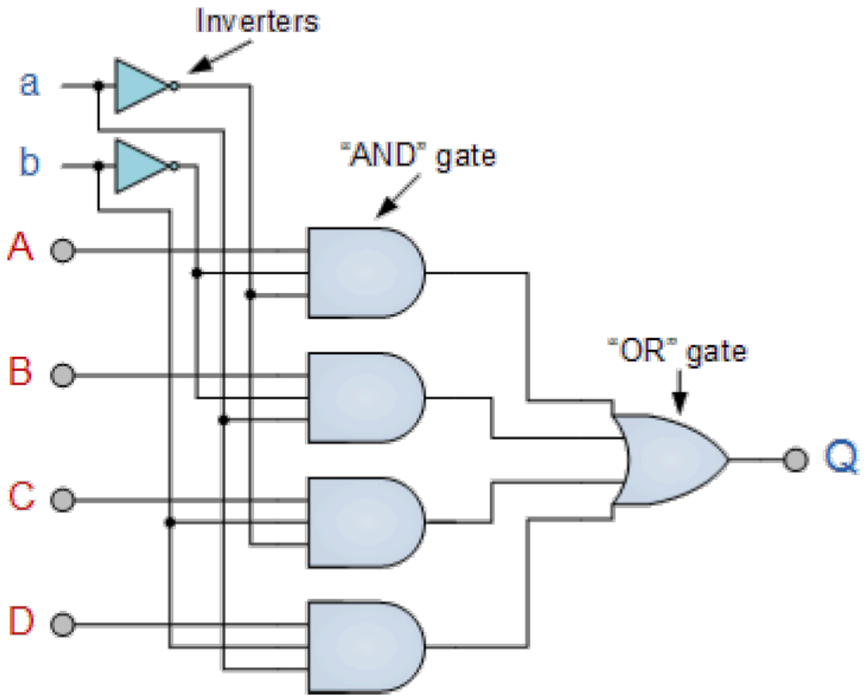


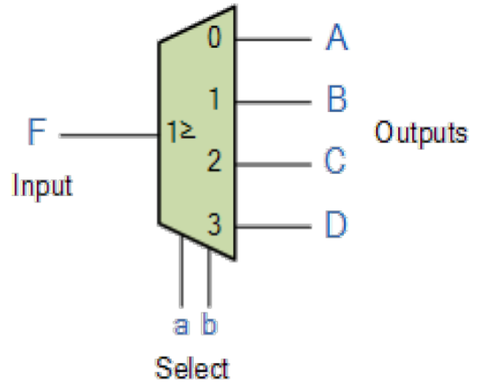


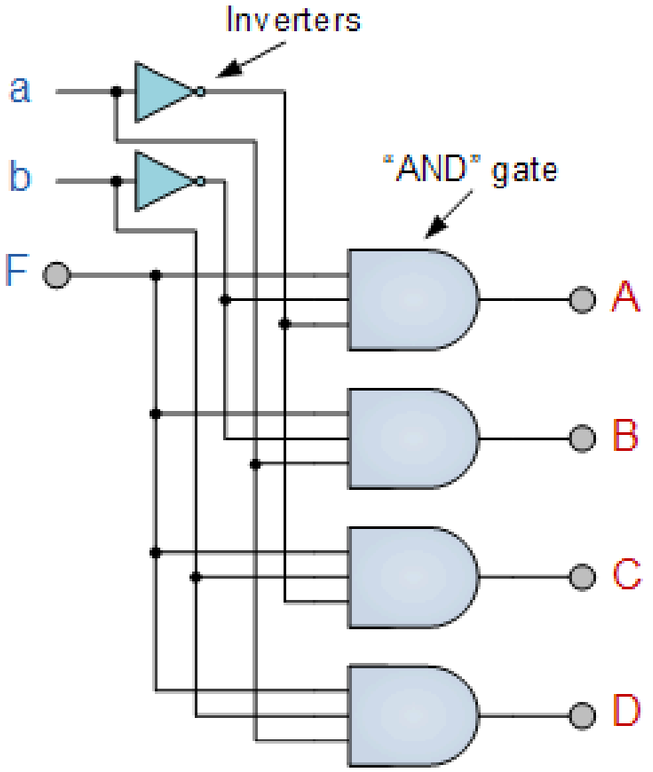


Multiplexor/Demultiplexor

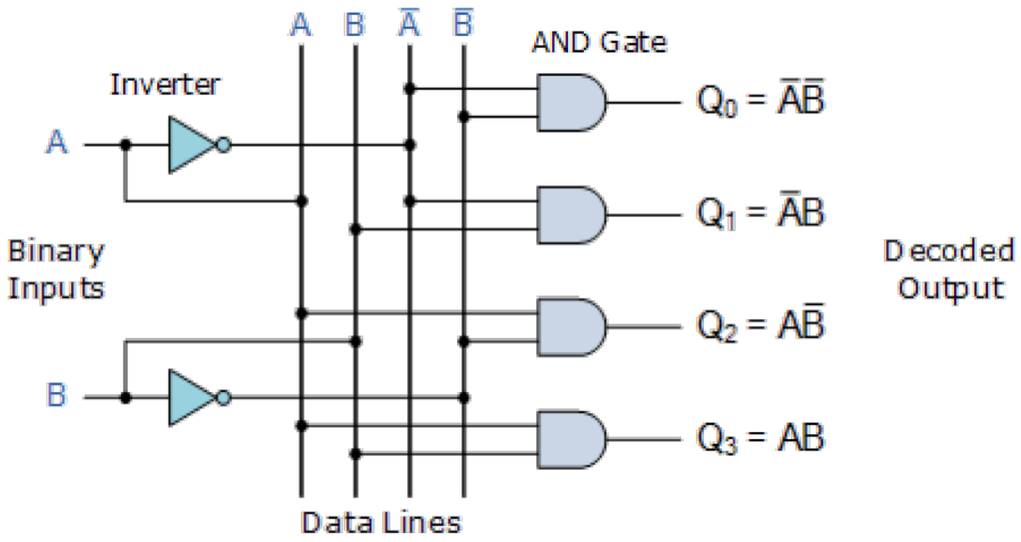




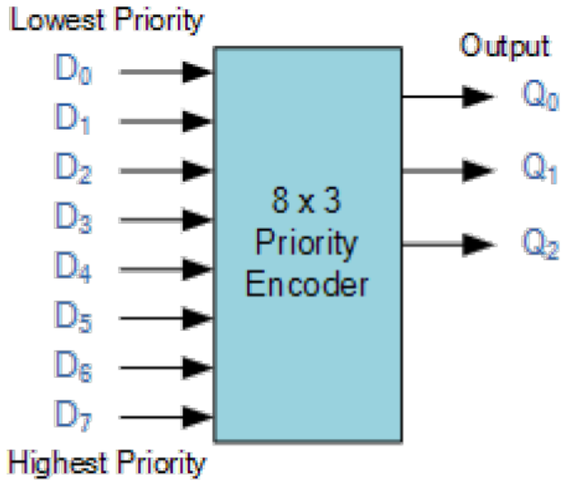




Decoder/Encoder

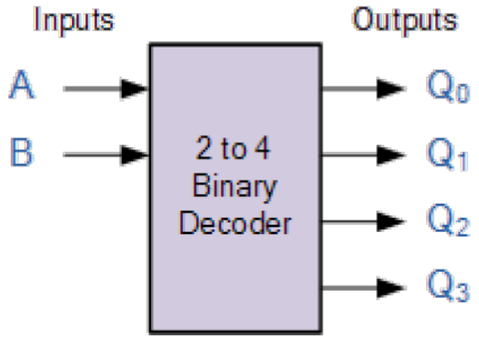


Decoded Output



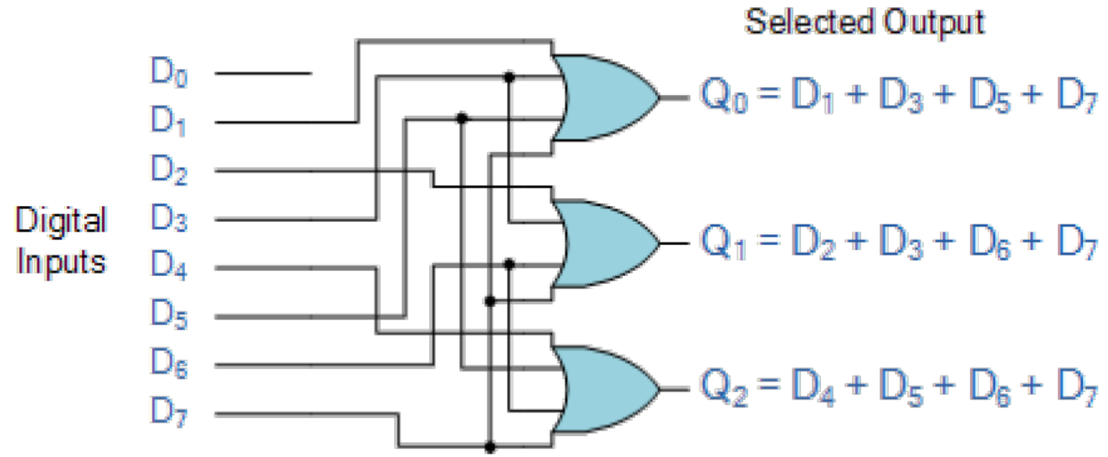
Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	x	x	x	x	x	x	1	1	0
1	x	x	x	x	x	x	x	1	1	1

X = don't care

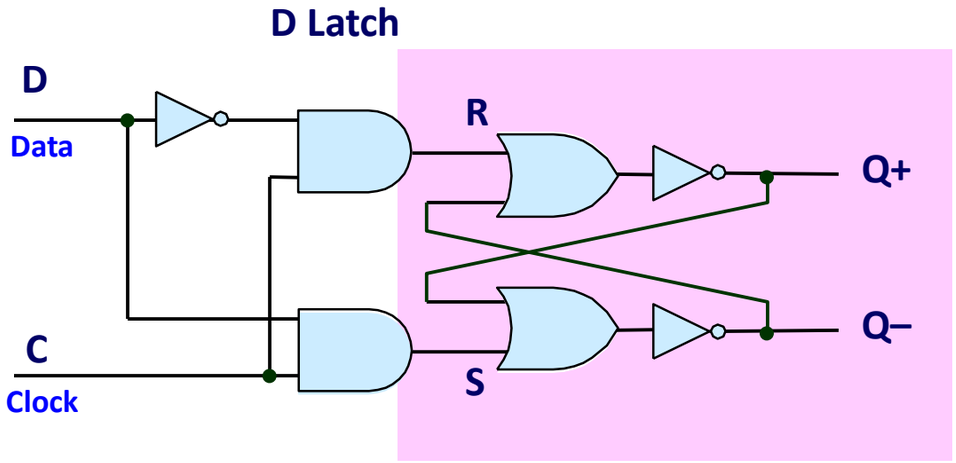


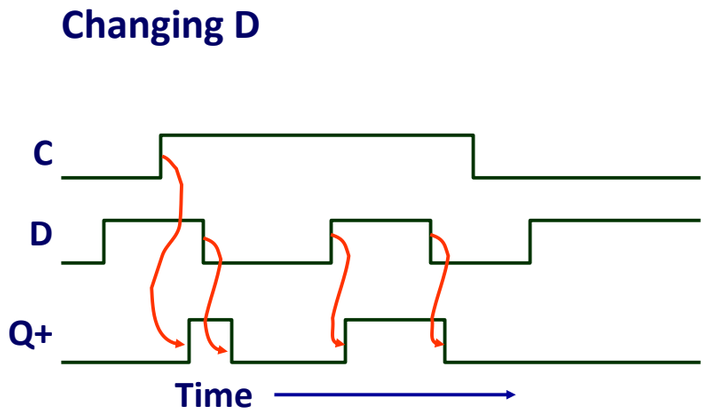
Truth Table

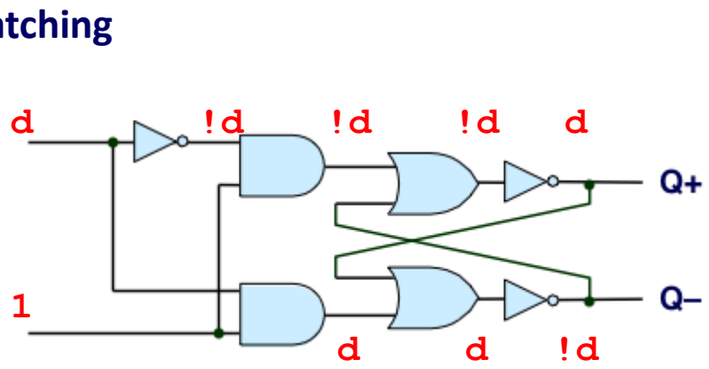
A	B	Q_0	Q_1	Q_2	Q_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

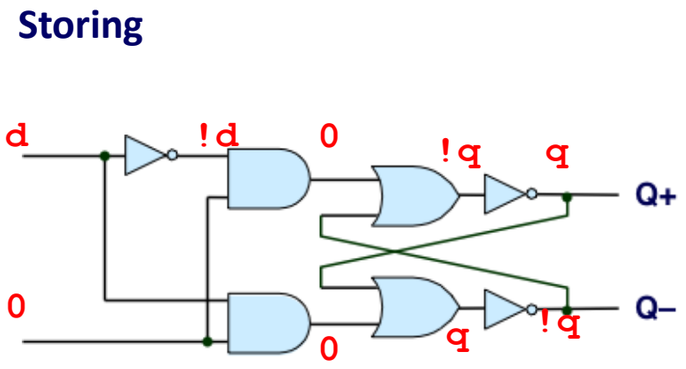


Transparent Latch



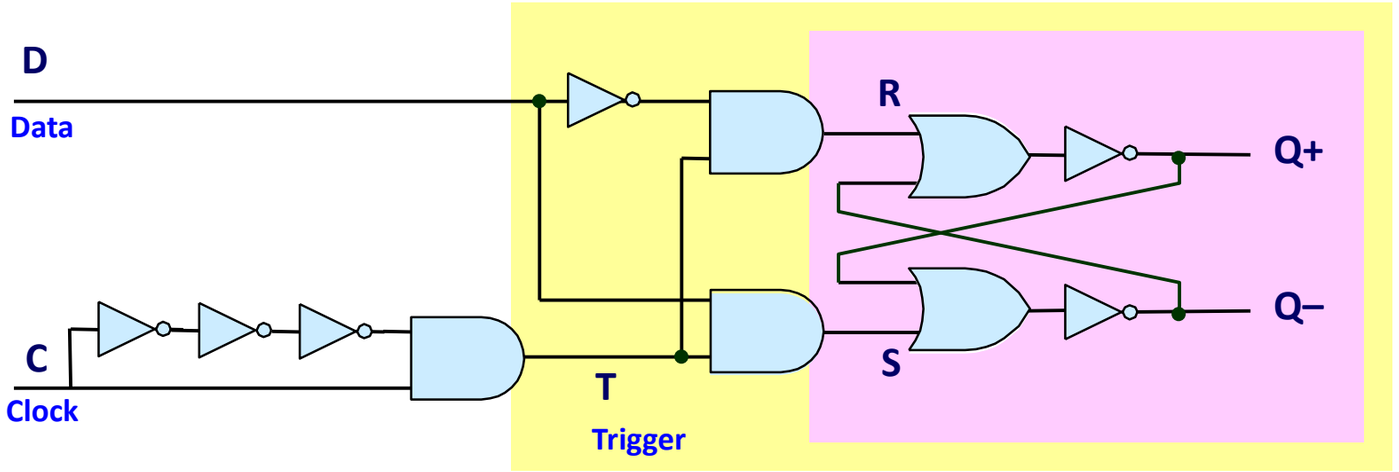


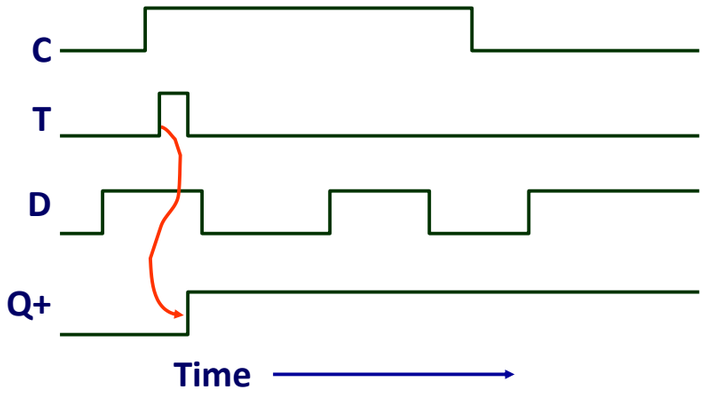




- When in latching mode, combinational propagation from D to Q+ and Q–
- Value latched depends on value of D as C falls

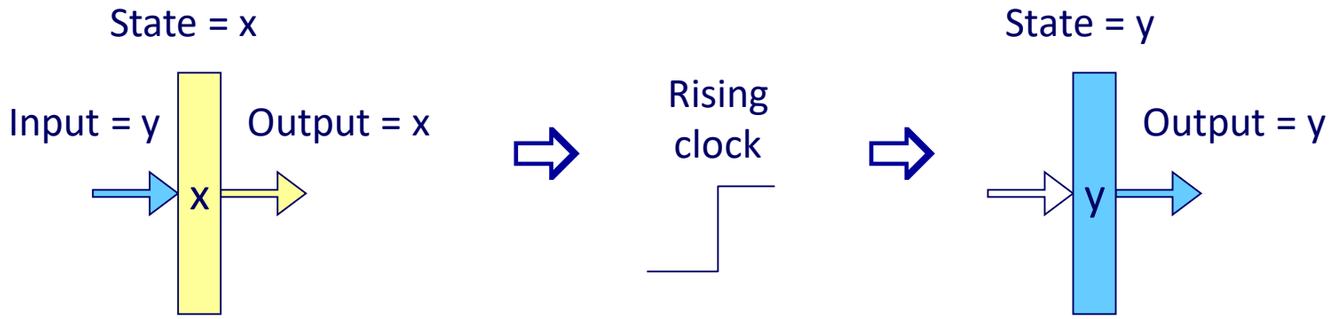
Edge-Triggered Latch

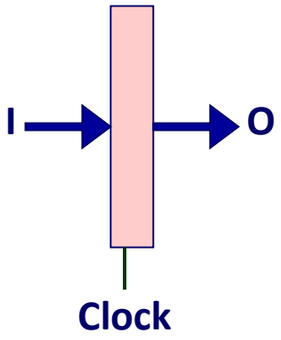




- Only in latching mode for brief period
  - Rising clock edge
- Value latched depends on data as clock rises
- Output remains stable at all other times

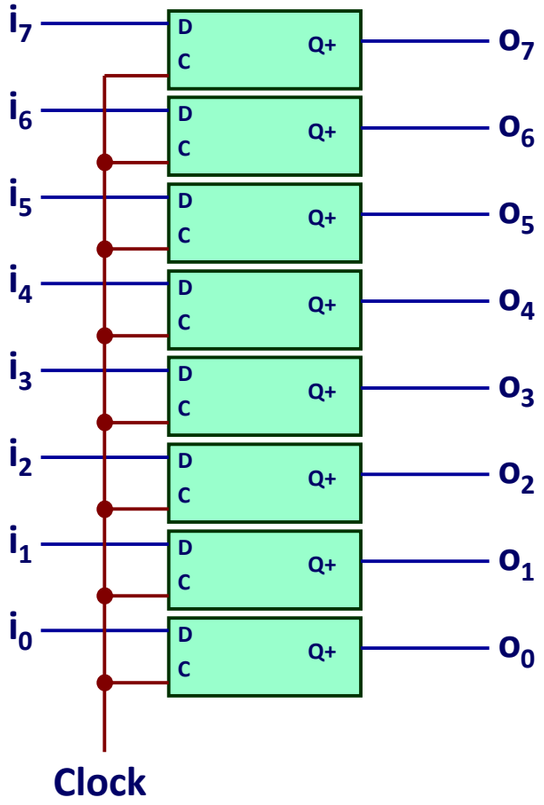
Registers





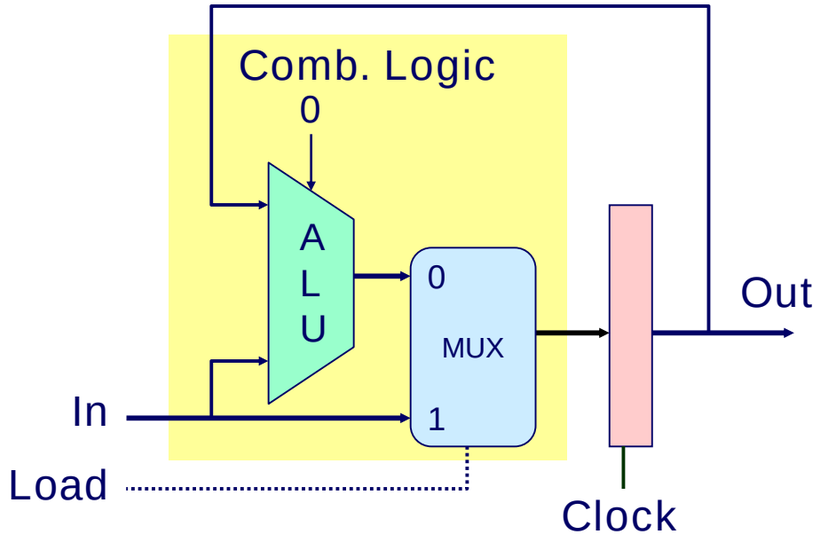
- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

Structure

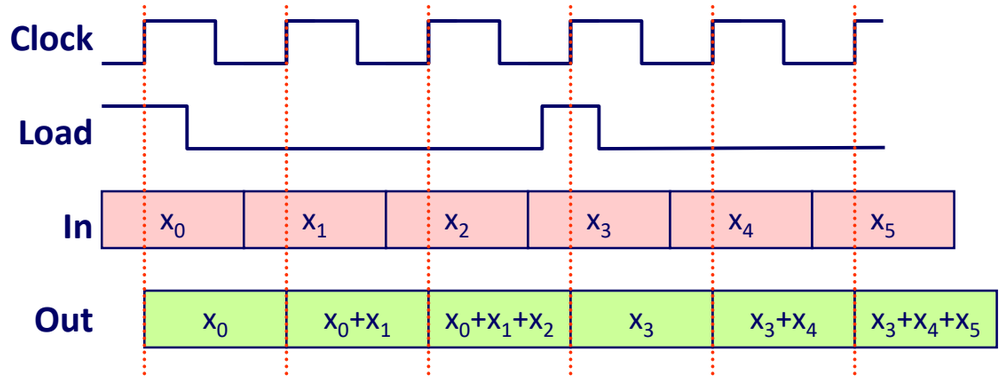


- Stores word of data
  - Different from *program registers* seen in assembly code
- Collection of edge-triggered latches
- Loads input on rising edge of clock

State Machine Example

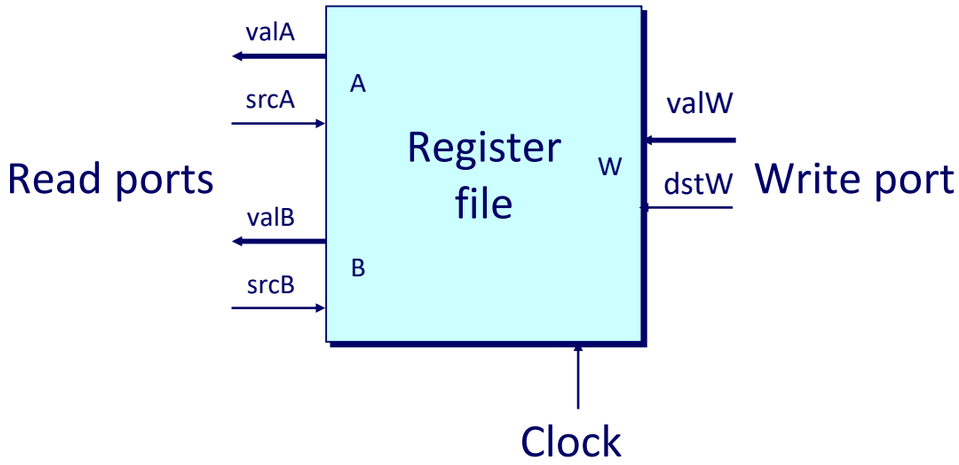


- **Accumulator circuit**
- **Load or accumulate on each cycle**

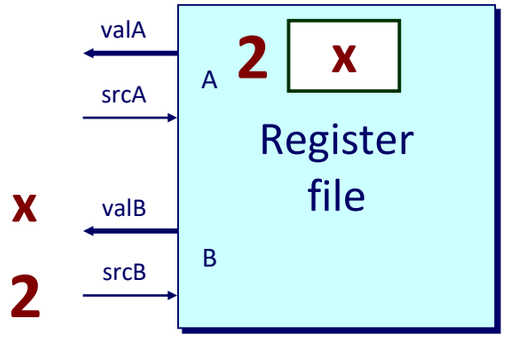


Random-Access Memory

- **Stores multiple words of memory**
  - **Address input specifies which word to read or write**
- **Register file**
  - **Holds values of program registers**
  - **%rax, %rsp, etc.**
  - **Register identifier serves as address**
    - » ID 15 (0xF) implies no read or write performed
- **Multiple Ports**
  - **Can read and/or write multiple words in one cycle**
    - » Each has separate address and data input/output



Register File Timing

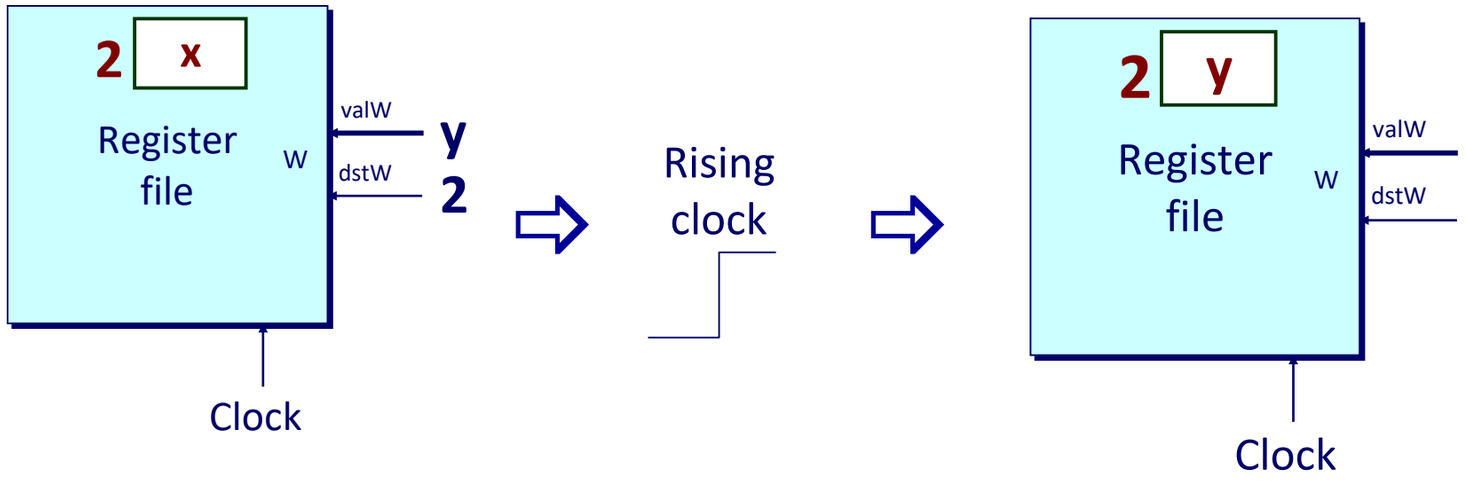


Reading

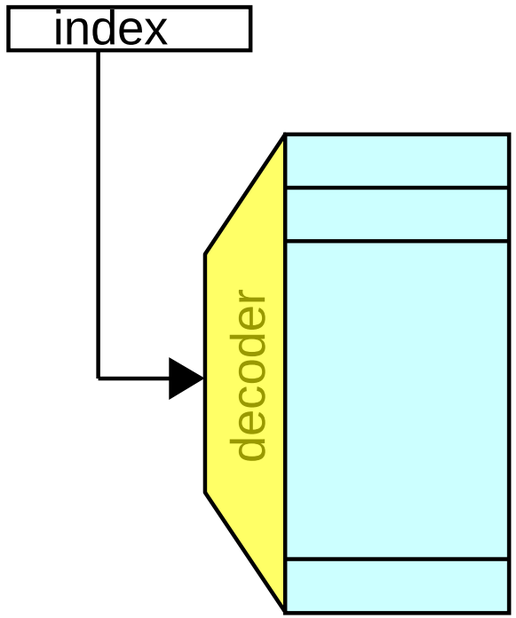
- Like combinational logic
- Output data generated based on input address
  - After some delay

Writing

- Like register
- Update only as clock rises

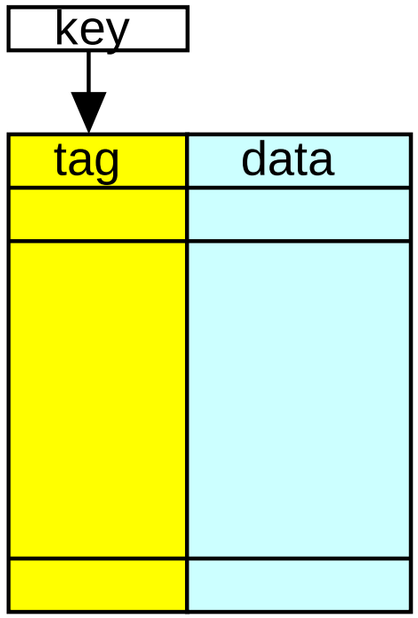


(Cache) Memory Implementation Options



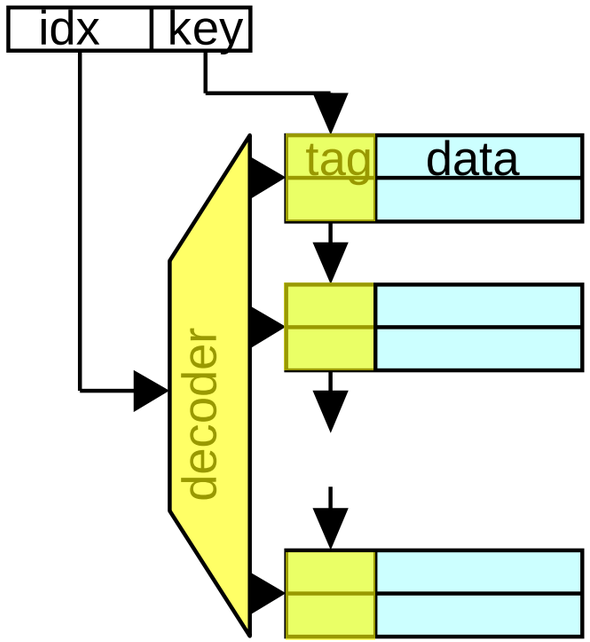
Indexed Memory

k-bit index
2^k blocks



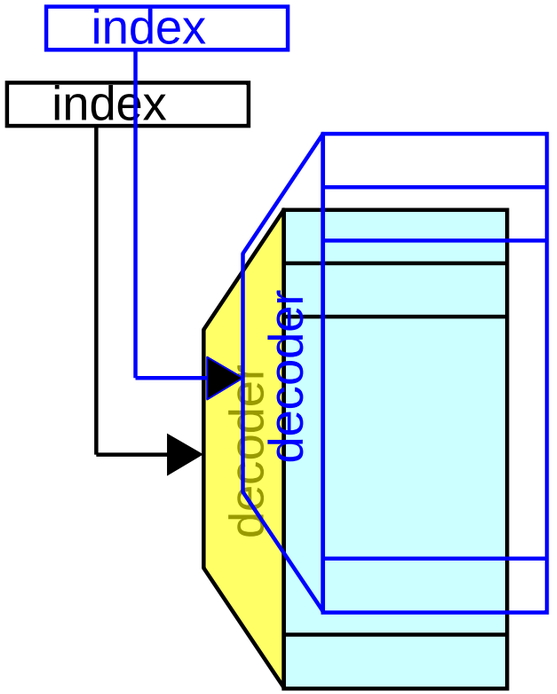
Associative Memory (CAM)

no index
unlimited blocks



N-Way Set-Associative Memory

k-bit index
$2^k \bullet N$ blocks



Indexed Memory (Multi-Ported)

(2x) k-bit index
(2x) 2^k blocks

18-600 Foundations of Computer Systems

Lecture 7: "Instruction-Set Processor Design"

1. **Processor Architecture**
   a. Instruction Set Architecture (ISA)
   b. Y86-64 Instruction Set Architecture
   c. Logic Design Revisited/Simplified
2. **Processor Implementation**
   a. Instruction Set Processor (CPU)
   b. Processor Organization Design
   c. Y86-64 Sequential Processor Design
3. **Motivation for Pipelining**

Electrical & Computer ENGINEERING

Instruction Processing Steps

Processor State

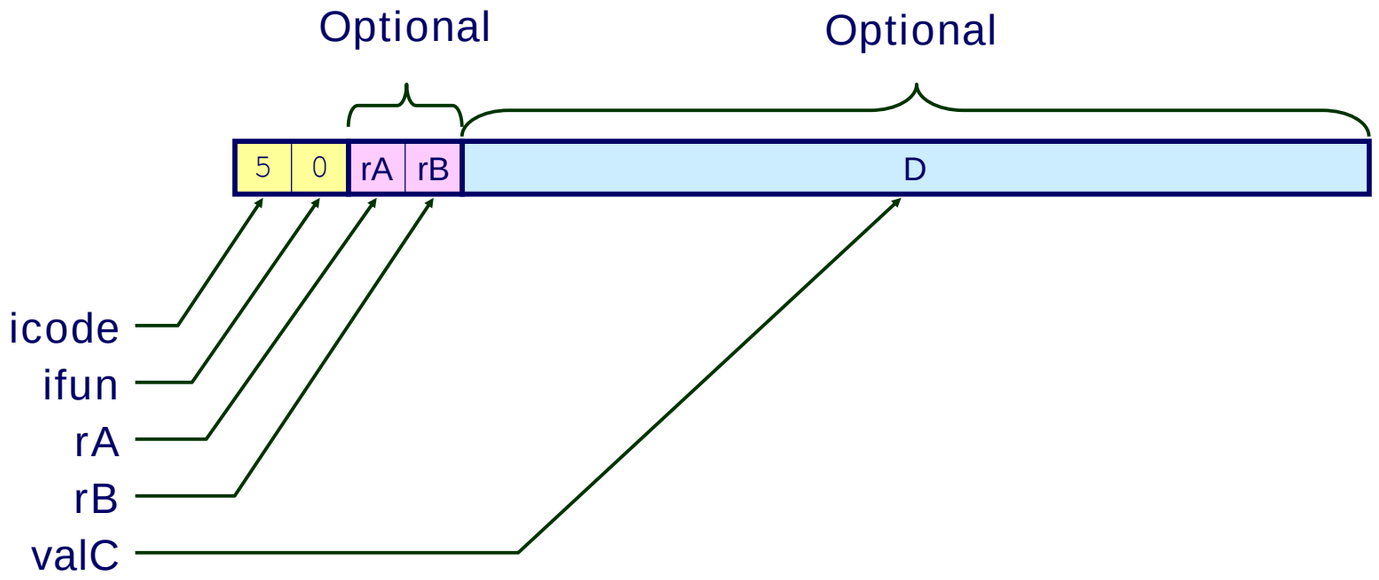
- Program counter register (PC)
- Condition code register (CC)
- Register File
- Memories
  - Access same memory space
  - Data: for reading/writing program data
  - Instruction: for reading instructions

Instruction Processing Flow

- Read instruction at address specified by PC
- Process through (four) typical steps
- Update program counter
- (Repeat)

1. Fetch
   - Read instruction from instruction memory
2. Decode
   - Determine Instruction type; Read program registers
3. Execute
   - Compute value or address
4. Memory
   - Read or write data in memory
5. Write Back
   - Write program registers
6. PC Update
   - Update program counter

Instruction Decoding



Instruction Format

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

Executing Arithmetic/Logical Operation

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/Log. Ops

	OPq 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 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

Processing Steps: `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

Execute

- Increment stack pointer by 8

Memory

- Read from old stack pointer

Write back

- Update stack pointer
- Write result to register

PC Update

- Increment PC by 2

Processing Steps: 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)

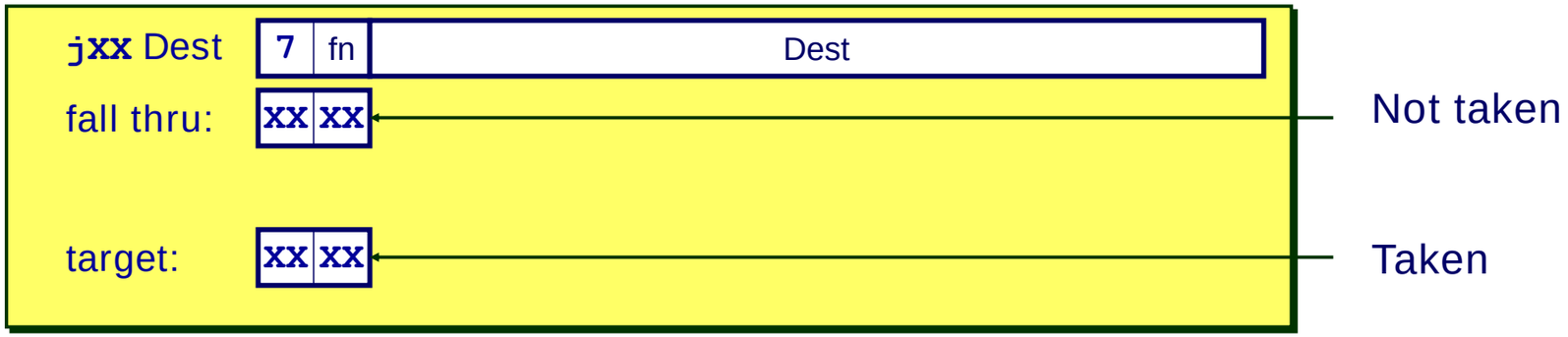
PC Update
- Increment PC by 2

Processing Steps: 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

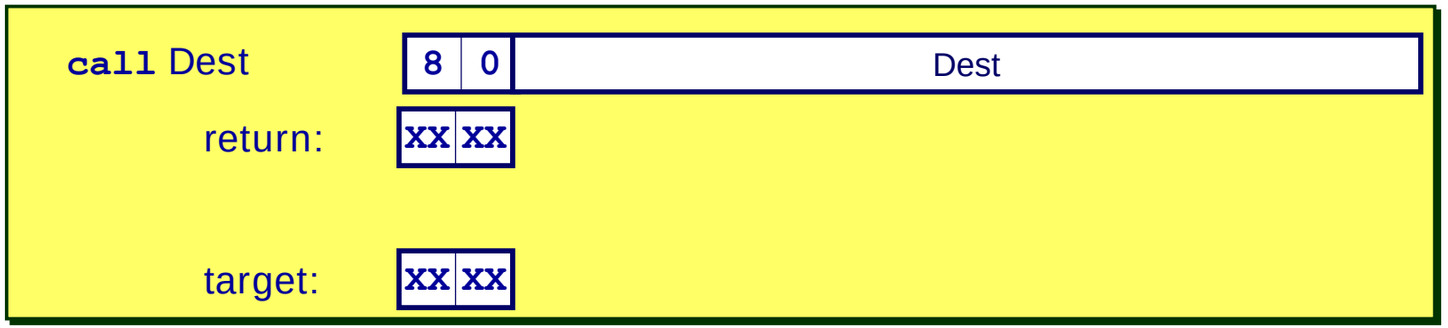
- Set PC to Dest if branch taken or to incremented PC if not branch

Processing Steps: 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

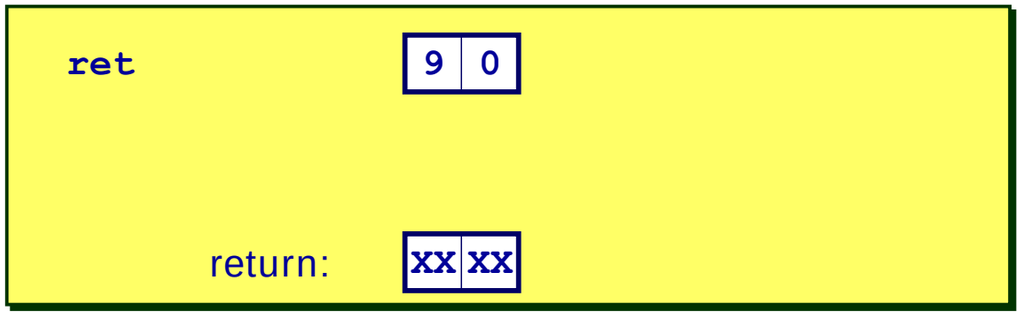
- Set PC to Dest

Processing Steps: `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

Processing Steps: `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

Instruction Processing 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 in each step

Instruction Processing 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 in 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

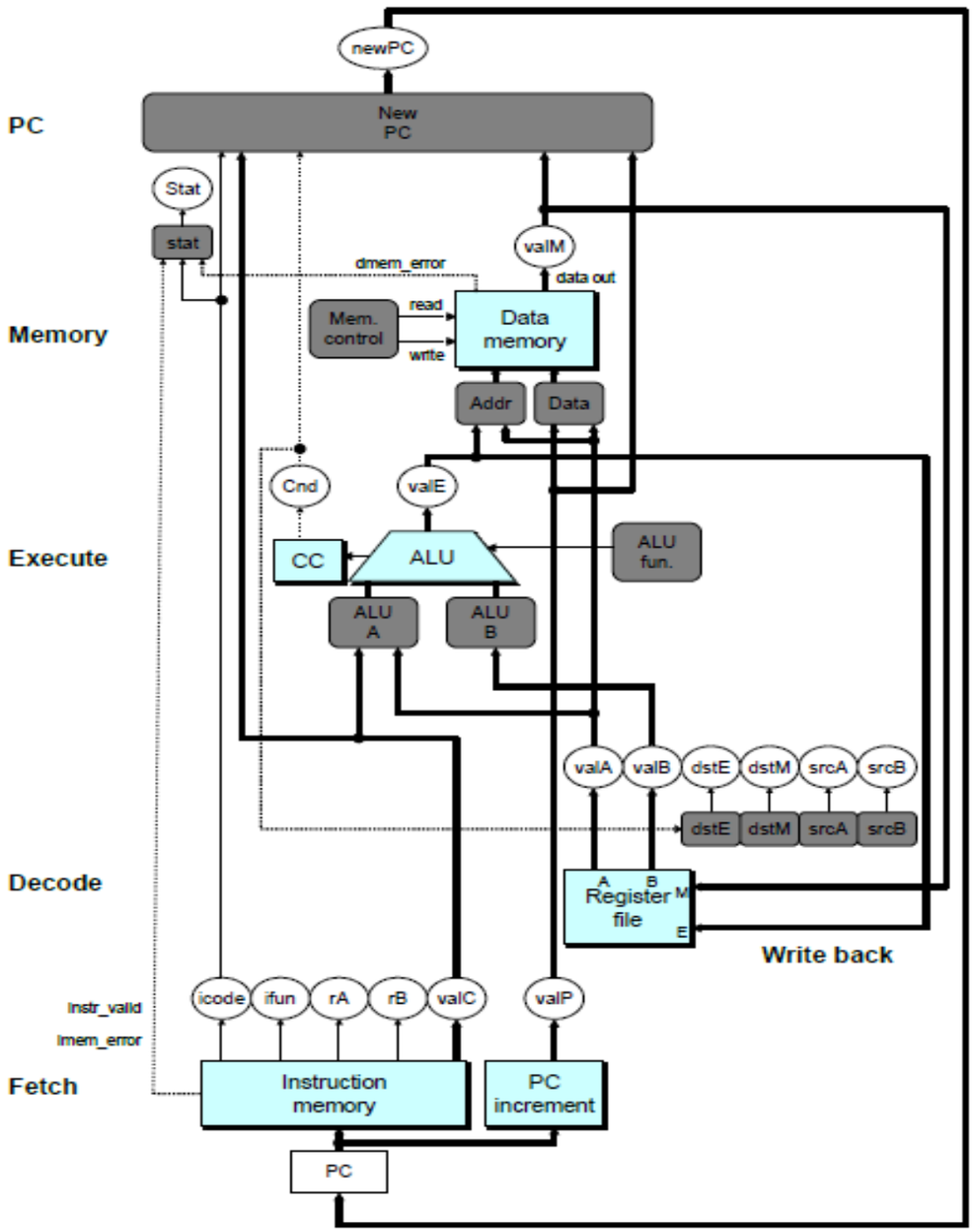
Memory

- valM Value from memory

SEQ Hardware

Key

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



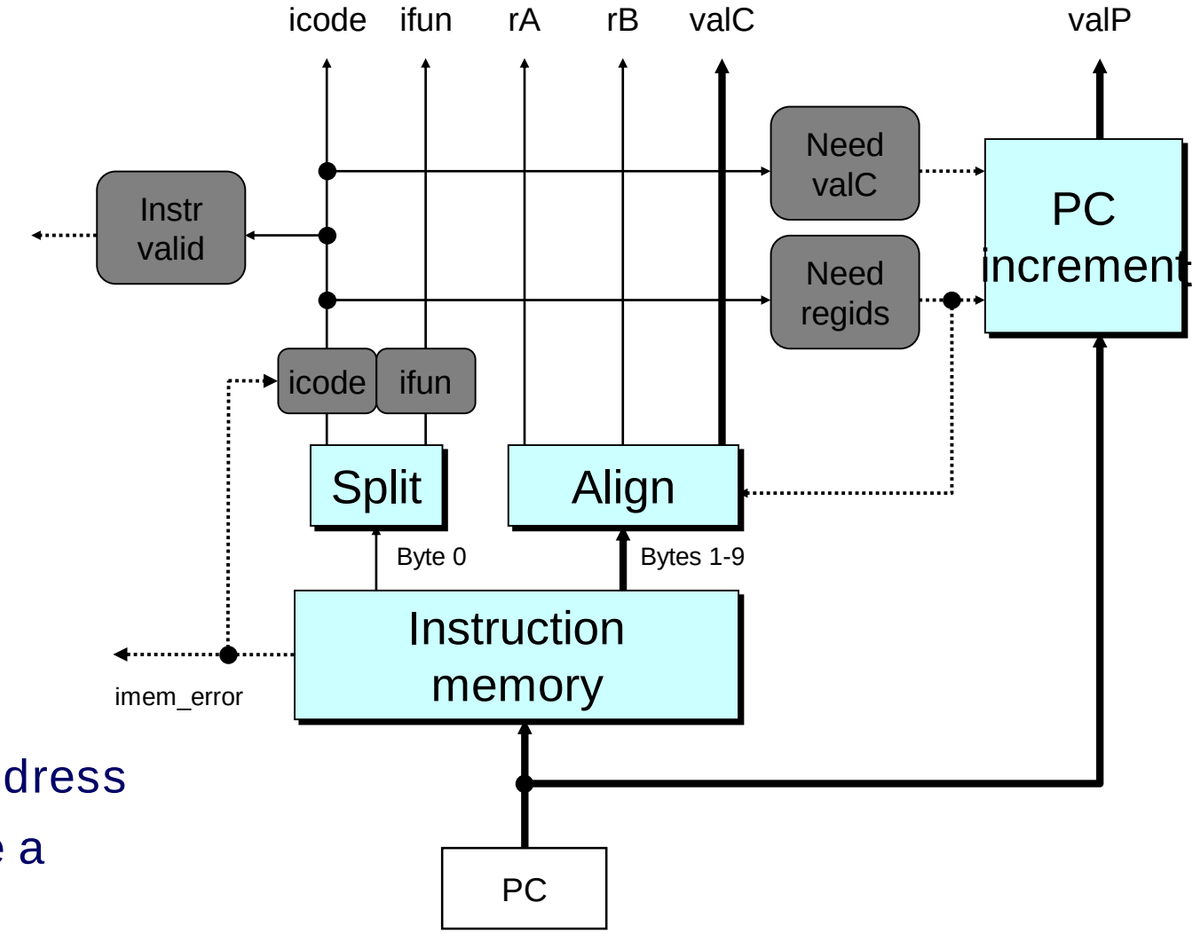
Fetch Logic

Predefined Blocks

- PC: Register containing PC
- Instruction memory: Read 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

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?



Decode Logic

Register File

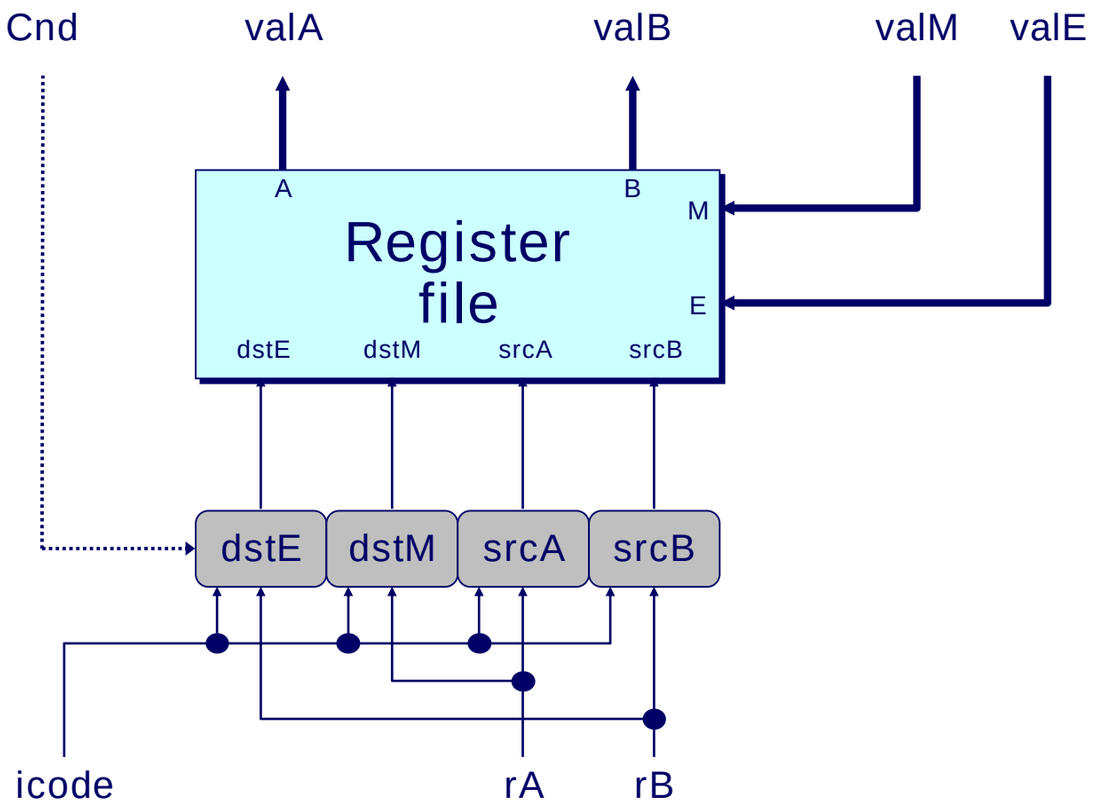
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

Control Logic

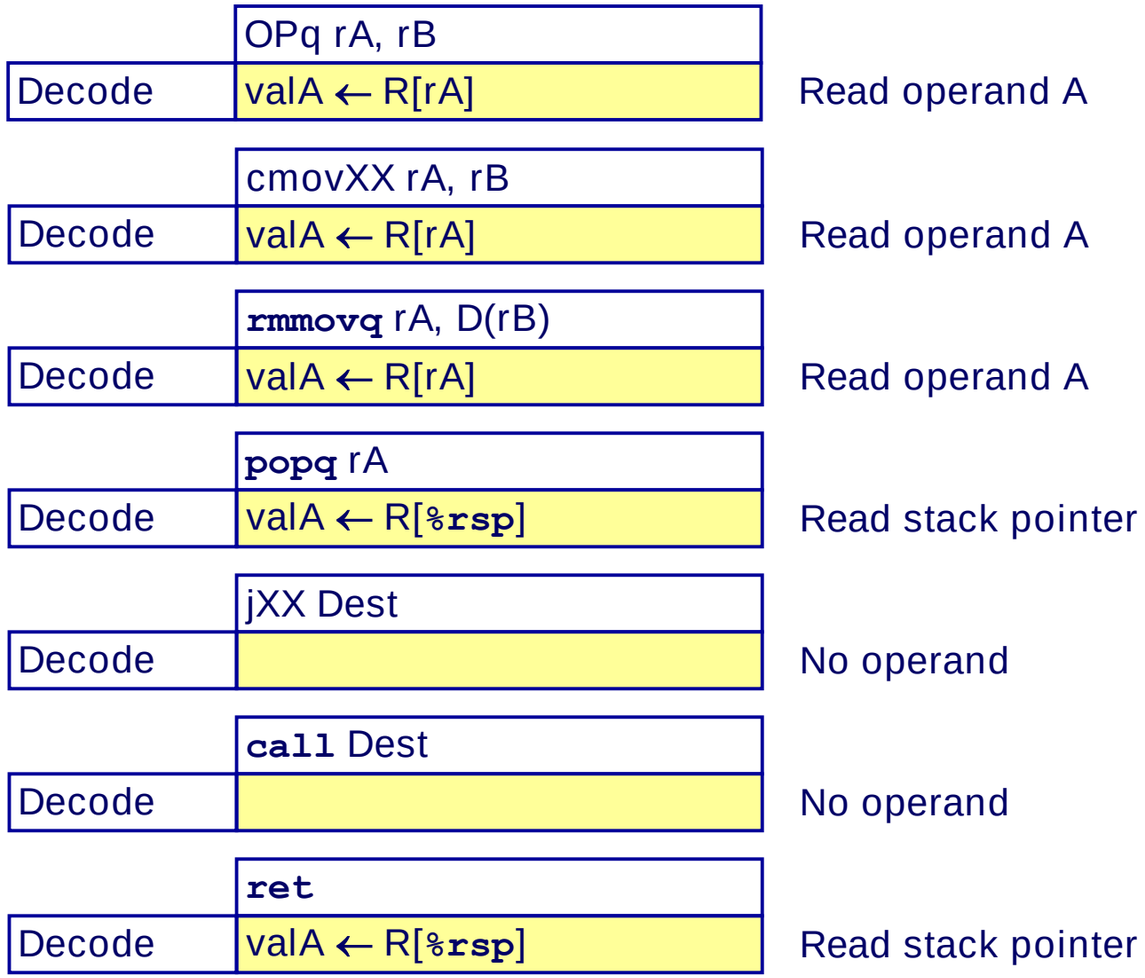
- srcA, srcB: read port addresses
- dstE, dstM: write port addresses

Signals

- Cnd: Indicate whether or not to perform conditional move
  - Computed in Execute stage

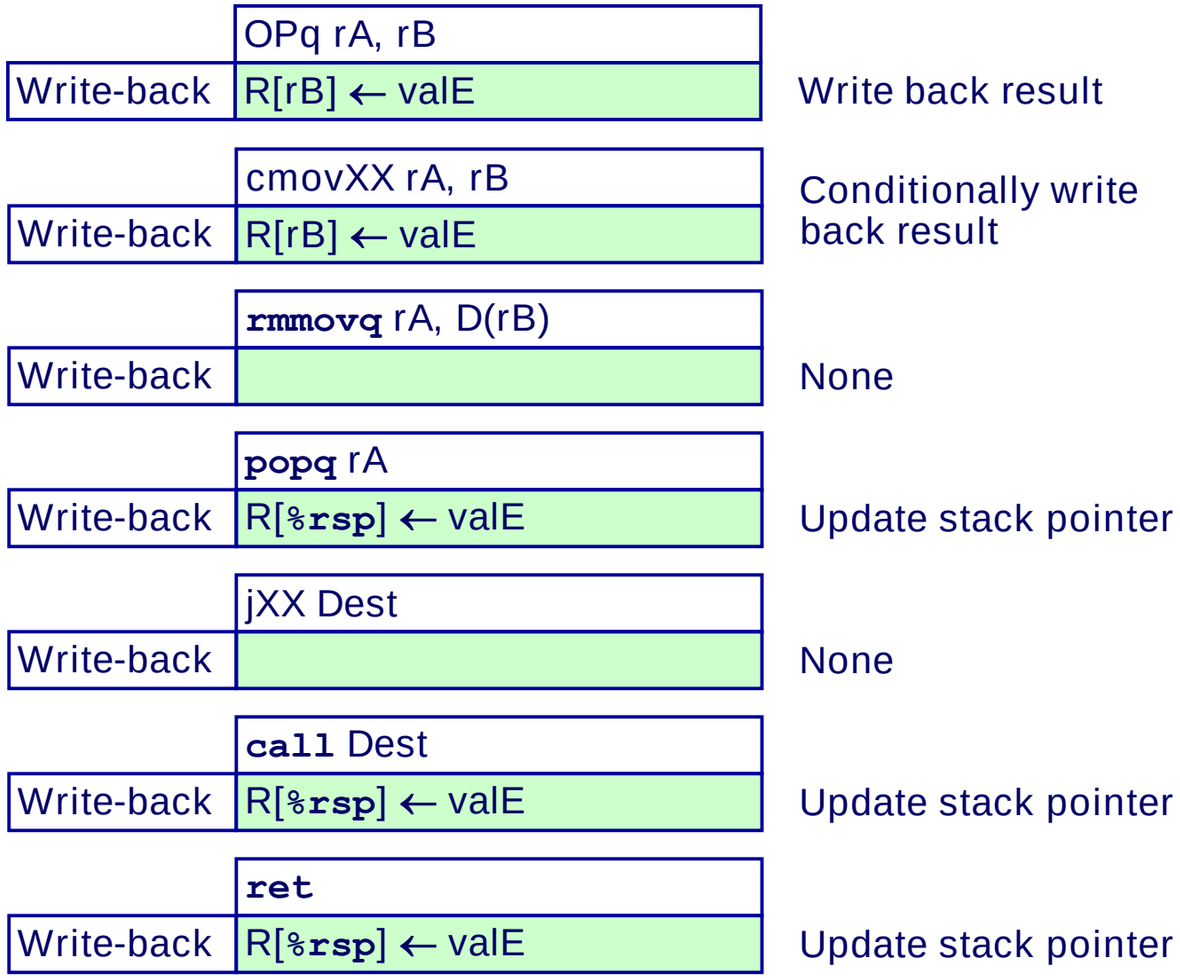


A Source




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

E Destination




```
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
];
```

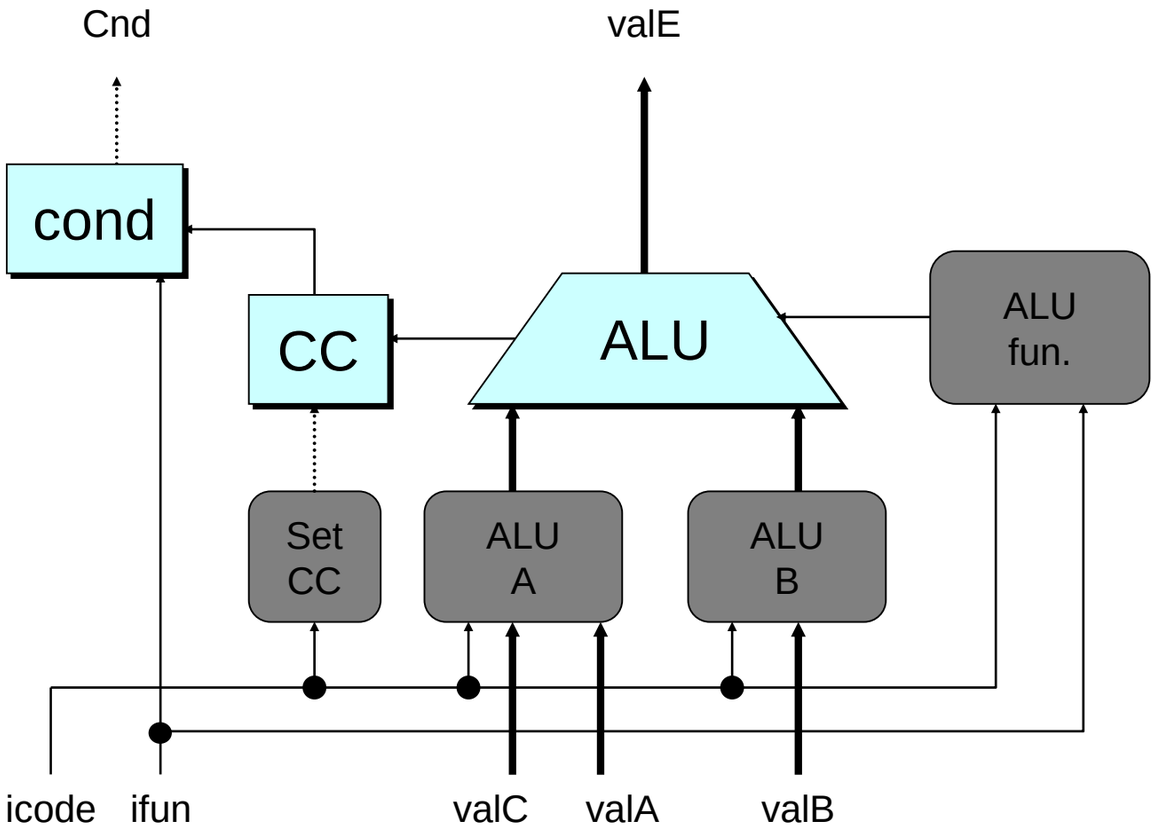
Execute Logic

Units

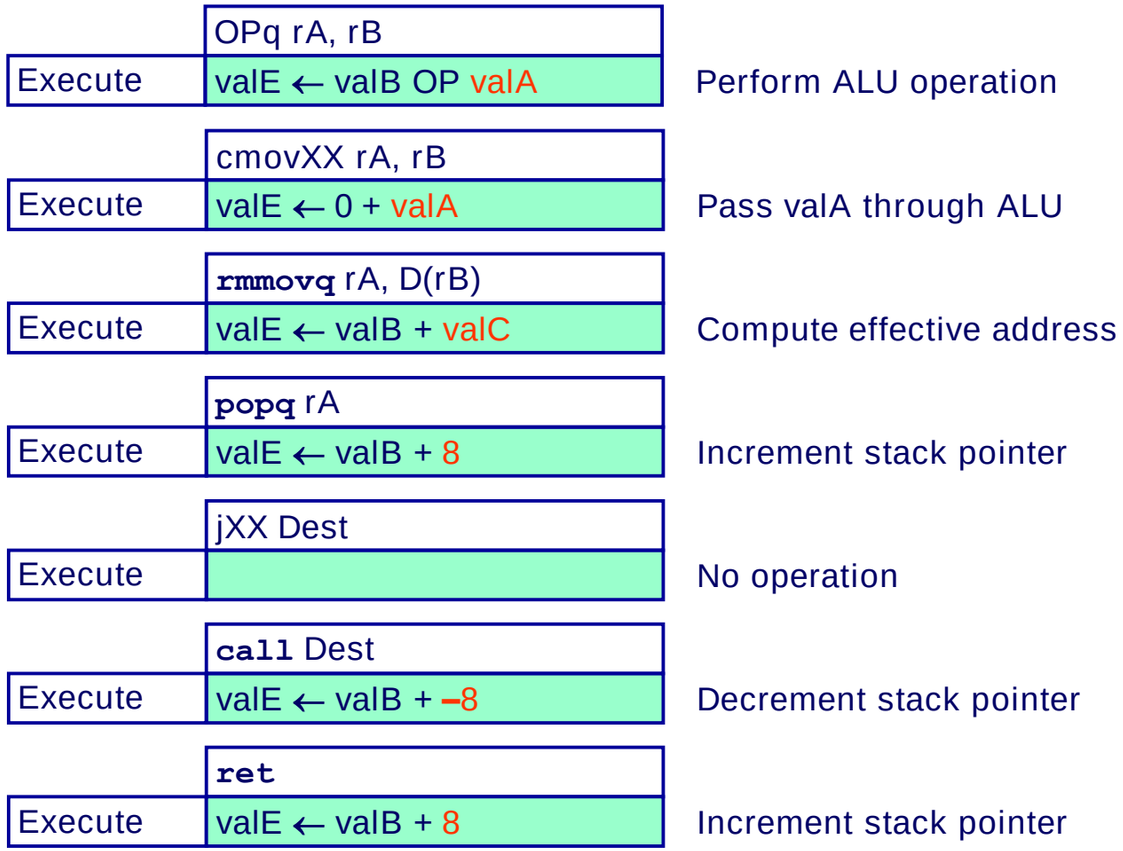
- ALU
  - Implements 4 required functions
  - Generates condition code values
- CC
  - Register with 3 condition code bits
- cond
  - Computes conditional jump/move flag

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 { 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 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

	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;
];
```

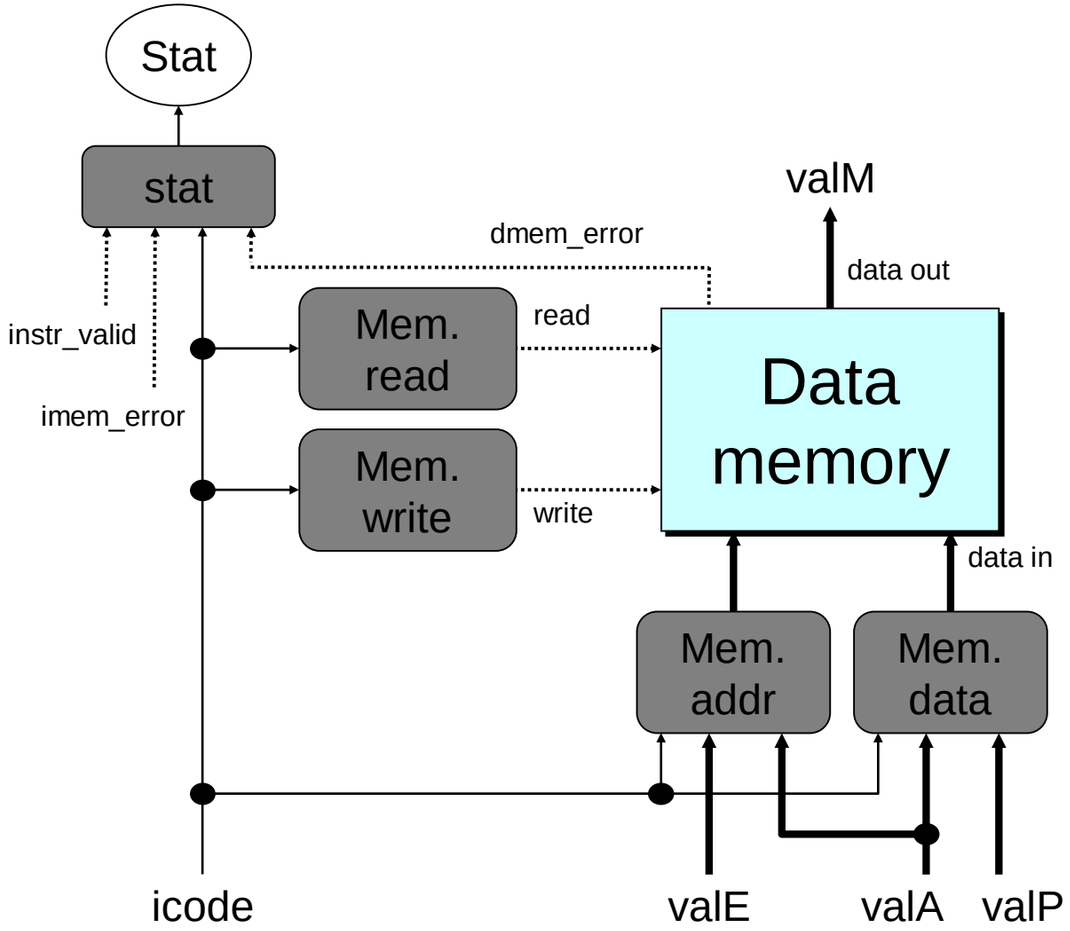
Memory Logic

Memory

- Reads or writes memory word

Control Logic

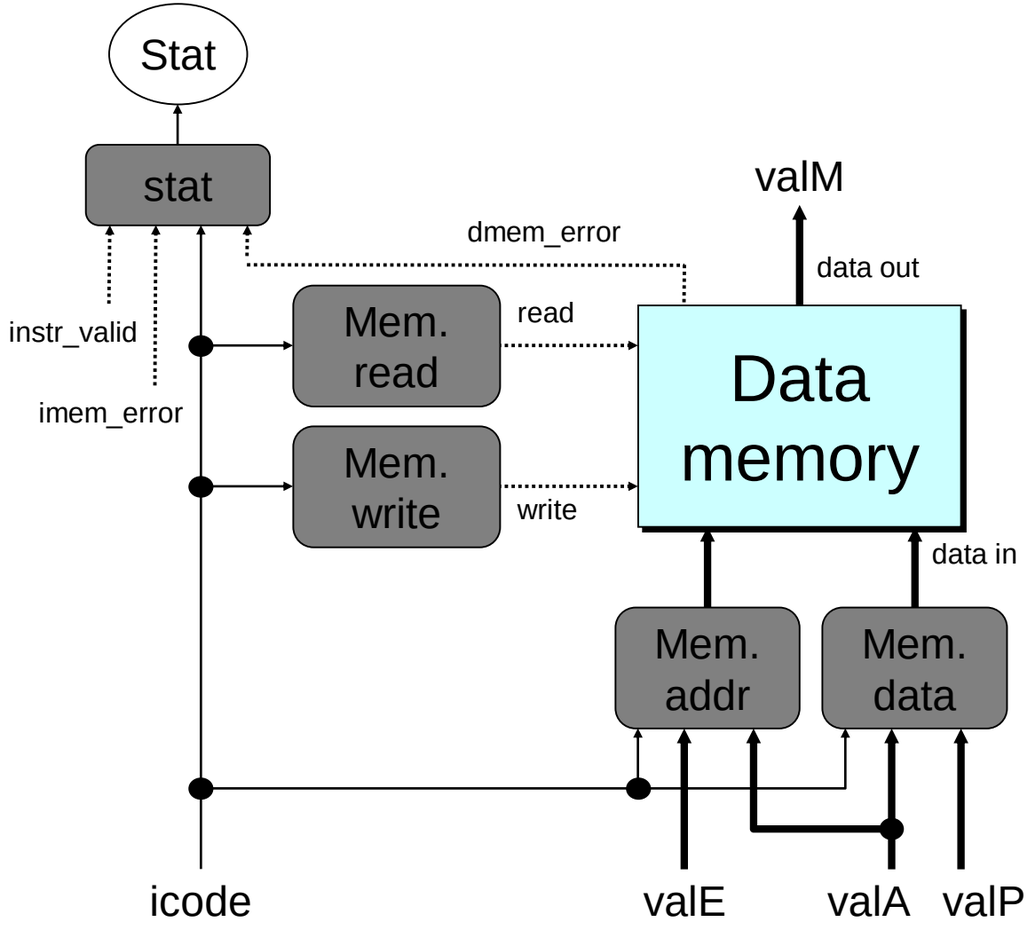
- stat: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



Instruction Status

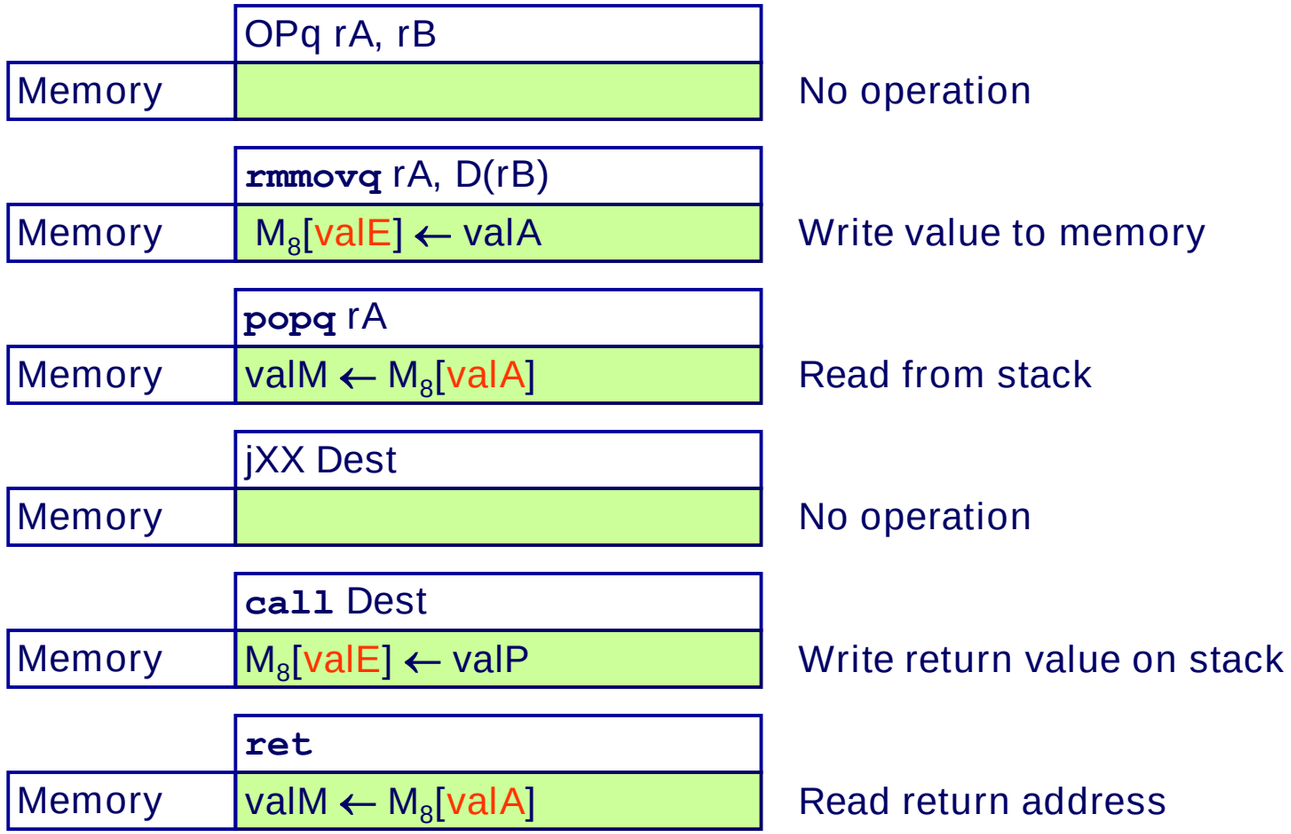
Control Logic

- stat: What is instruction status?



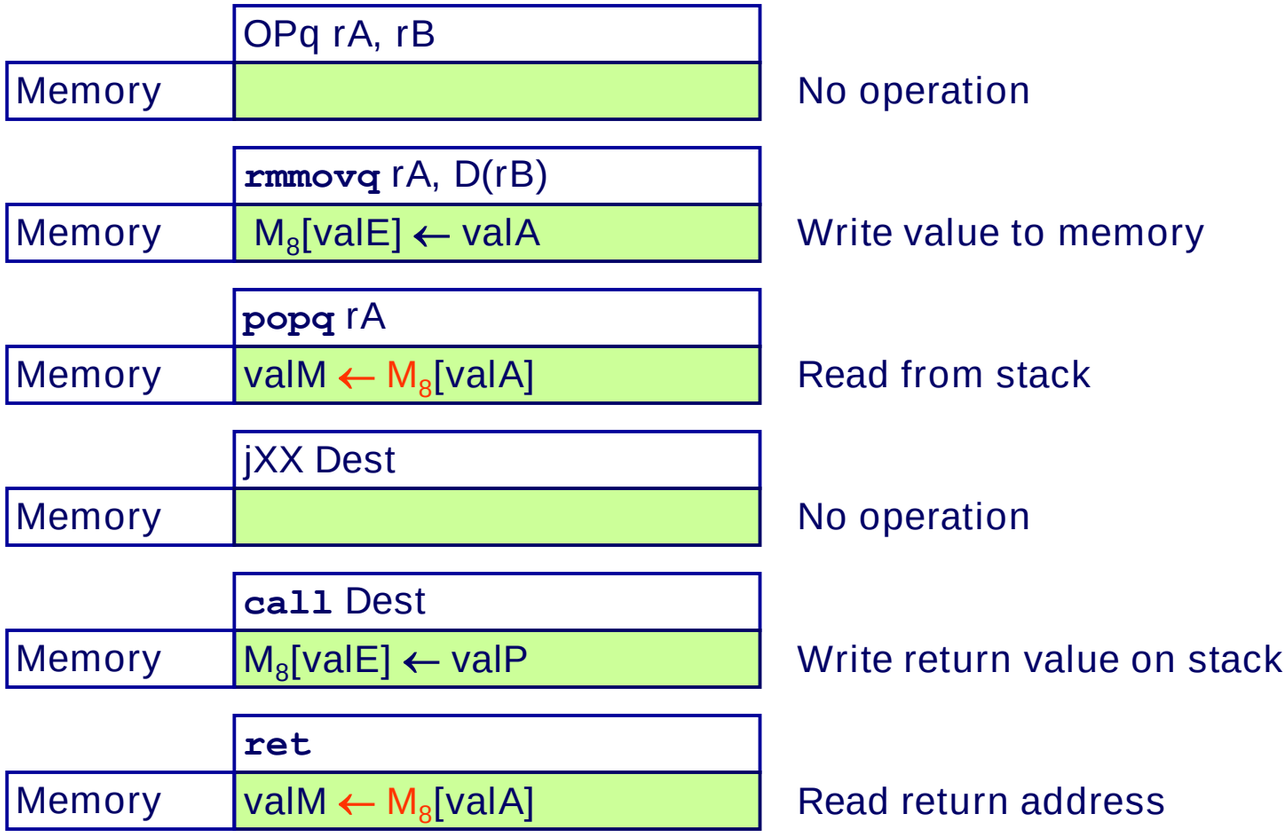

```
## Determine instruction status
int Stat = [
        imem_error || dmem_error : SADR;
        !instr_valid: SINS;
        icode == IHALT : SHLT;
        1 : SAOK;
];
```

Memory Address




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

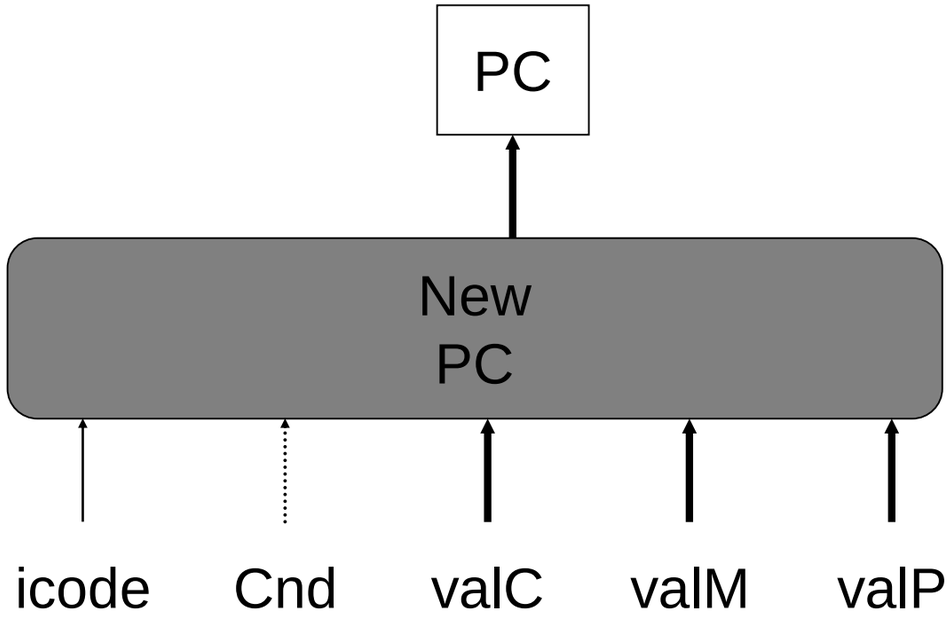
Memory Read




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

PC Update

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

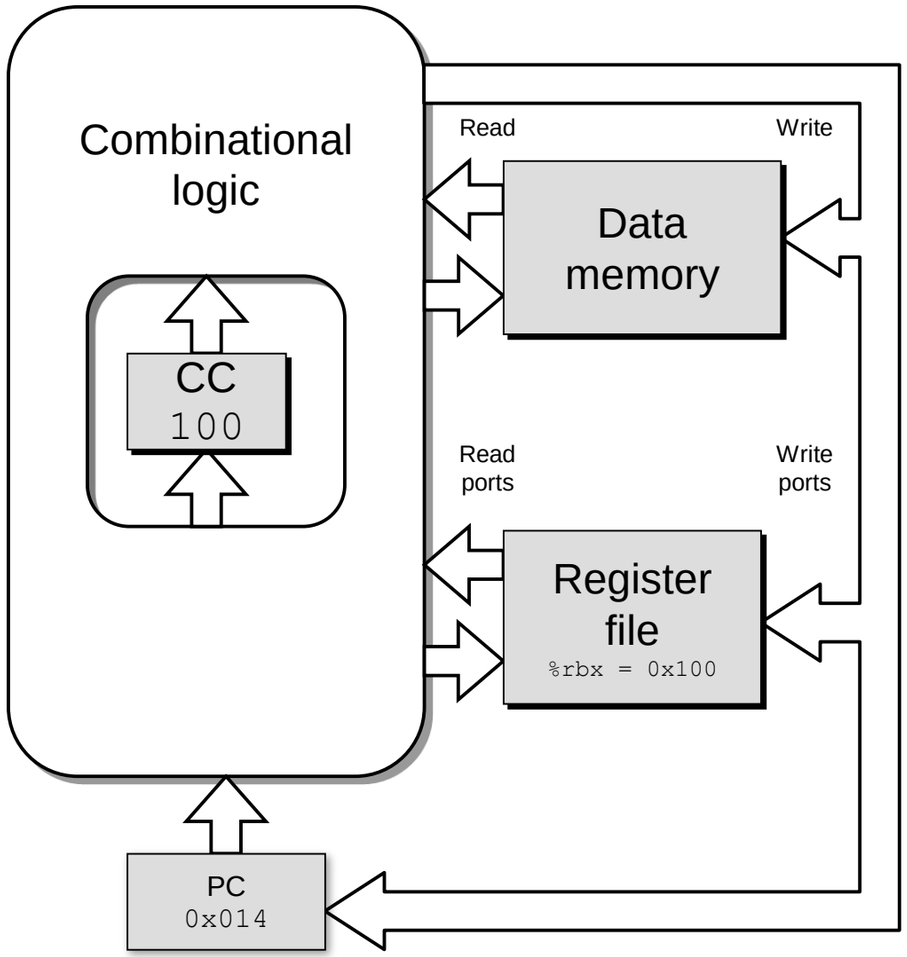


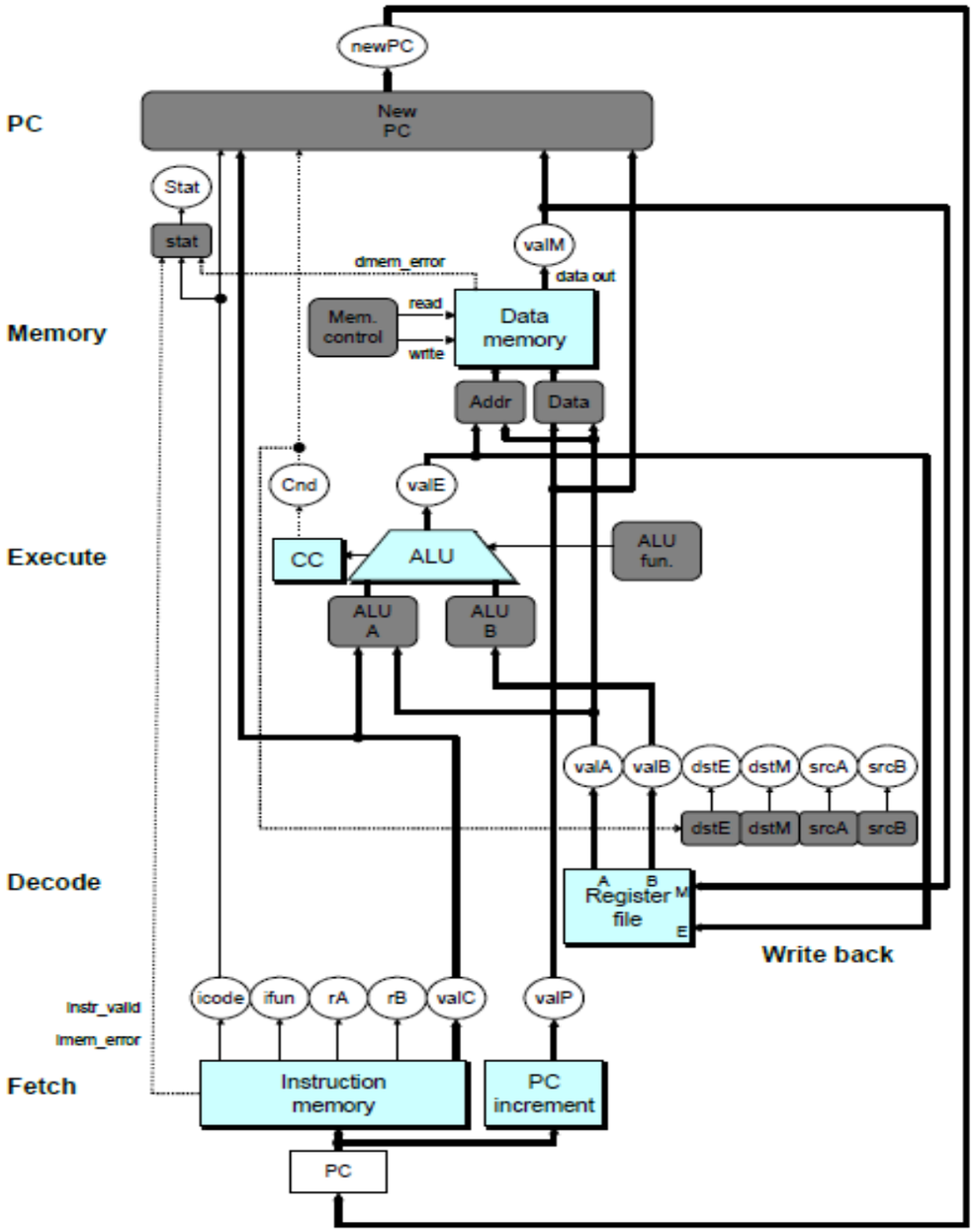
	OPq rA, rB	
PC update	PC ← valP	Update PC
	rmmovq rA, D(rB)	
PC update	PC ← valP	Update PC
	popq rA	
PC update	PC ← valP	Update PC
	jXX Dest	
PC update	PC ← Cnd ? valC : valP	Update PC
	call Dest	
PC update	PC ← valC	Set PC to destination
	ret	
PC update	PC ← valM	Set PC to return address

New PC

- **Select next value of PC**

SEQ Y86-64 Processor





18-600 Foundations of Computer Systems

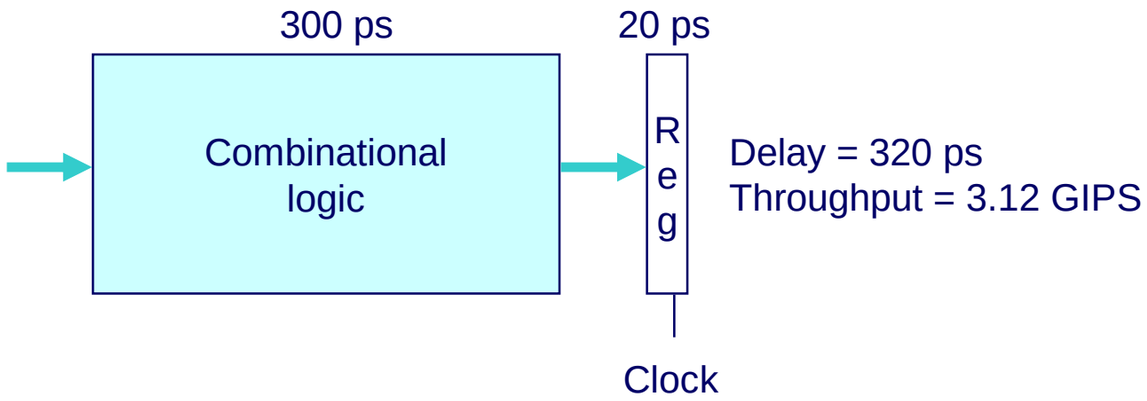
Lecture 7: "Instruction-Set Processor Design"

1. **Processor Architecture**
   a. Instruction Set Architecture (ISA)
   b. Y86-64 Instruction Set Architecture
   c. Logic Design Revisited/Simplified
2. **Processor Implementation**
   a. Instruction Set Processor (CPU)
   b. Processor Organization Design
   c. Y86-64 Sequential Processor Design
3. **Motivation for Pipelining**

Electrical & Computer ENGINEERING

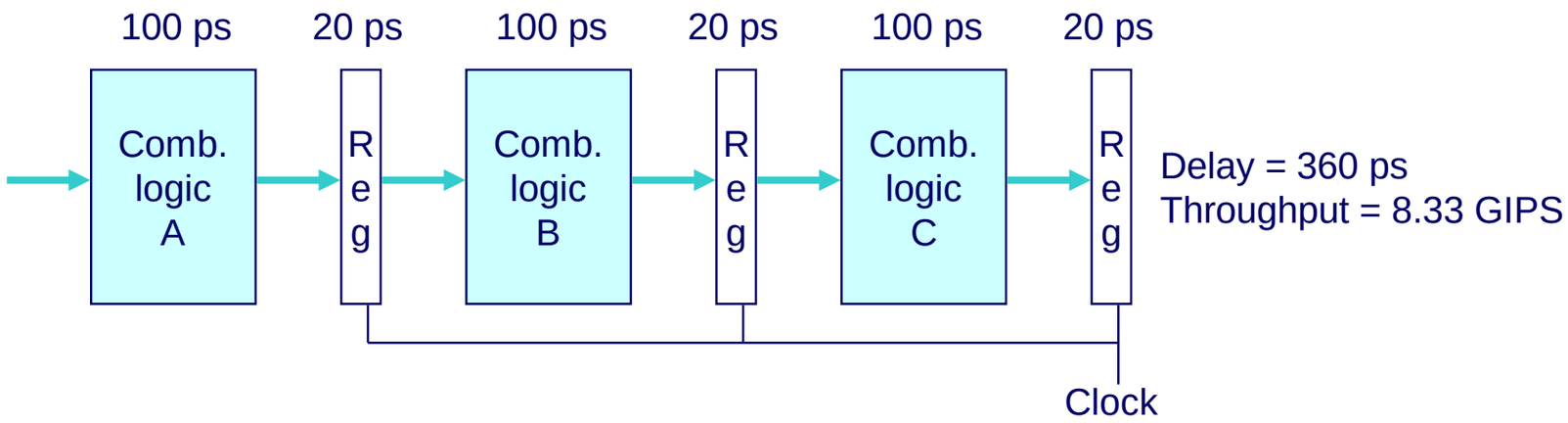
Carnegie Mellon University

Computational Example



- System
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Must have clock cycle of at least 320 ps

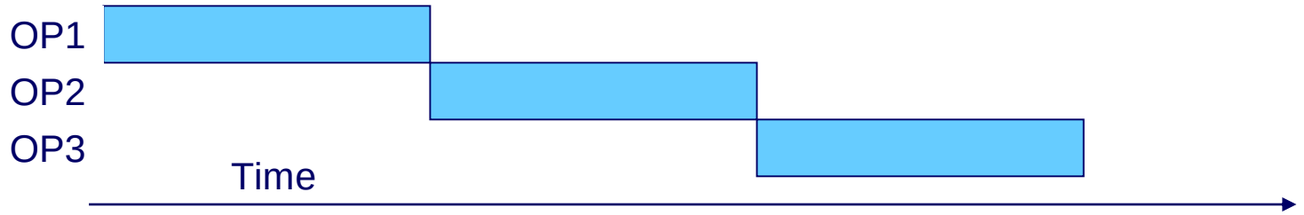
3-Way Pipelined Version



- System
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous one passes through stage A.
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

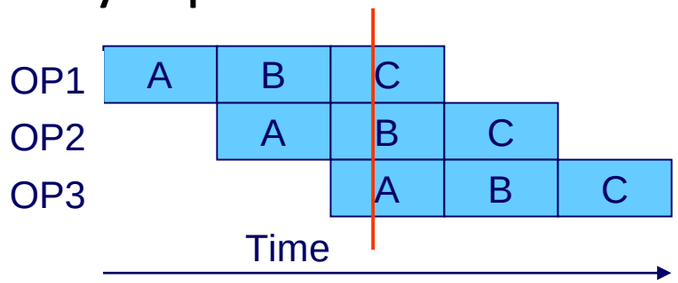
Pipeline Diagrams

- Unpipelined



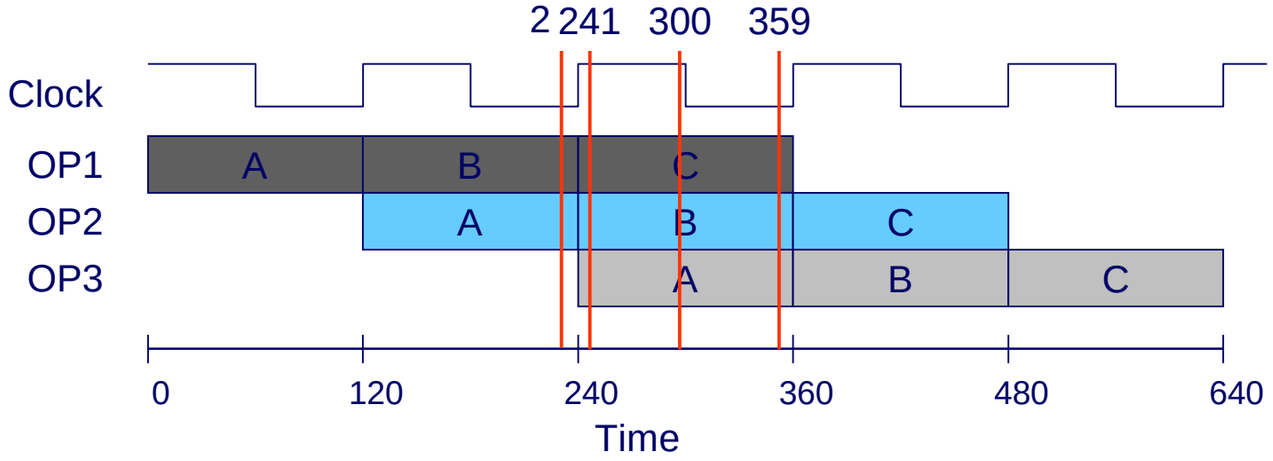
  - Cannot start new operation until previous one completes

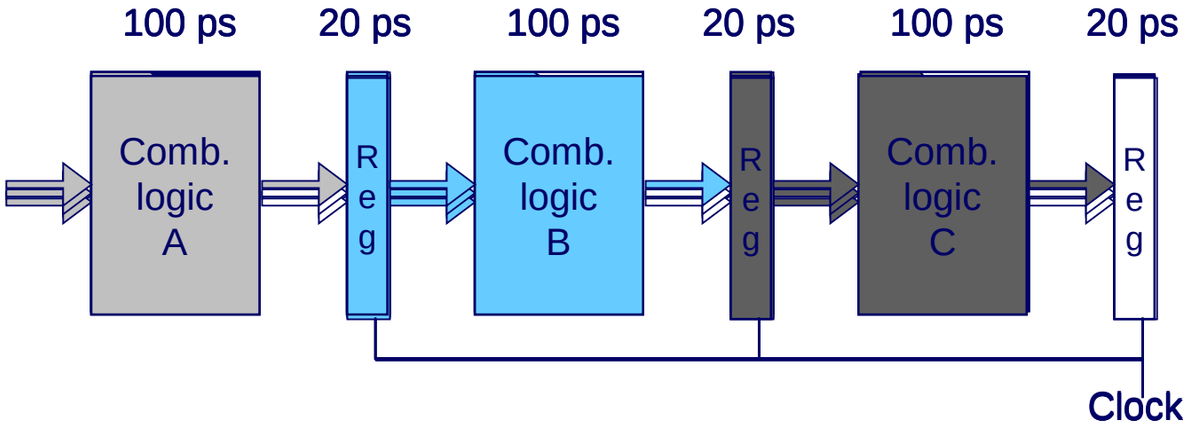
- 3-Way Pipelined



  - Up to 3 operations in process simultaneously

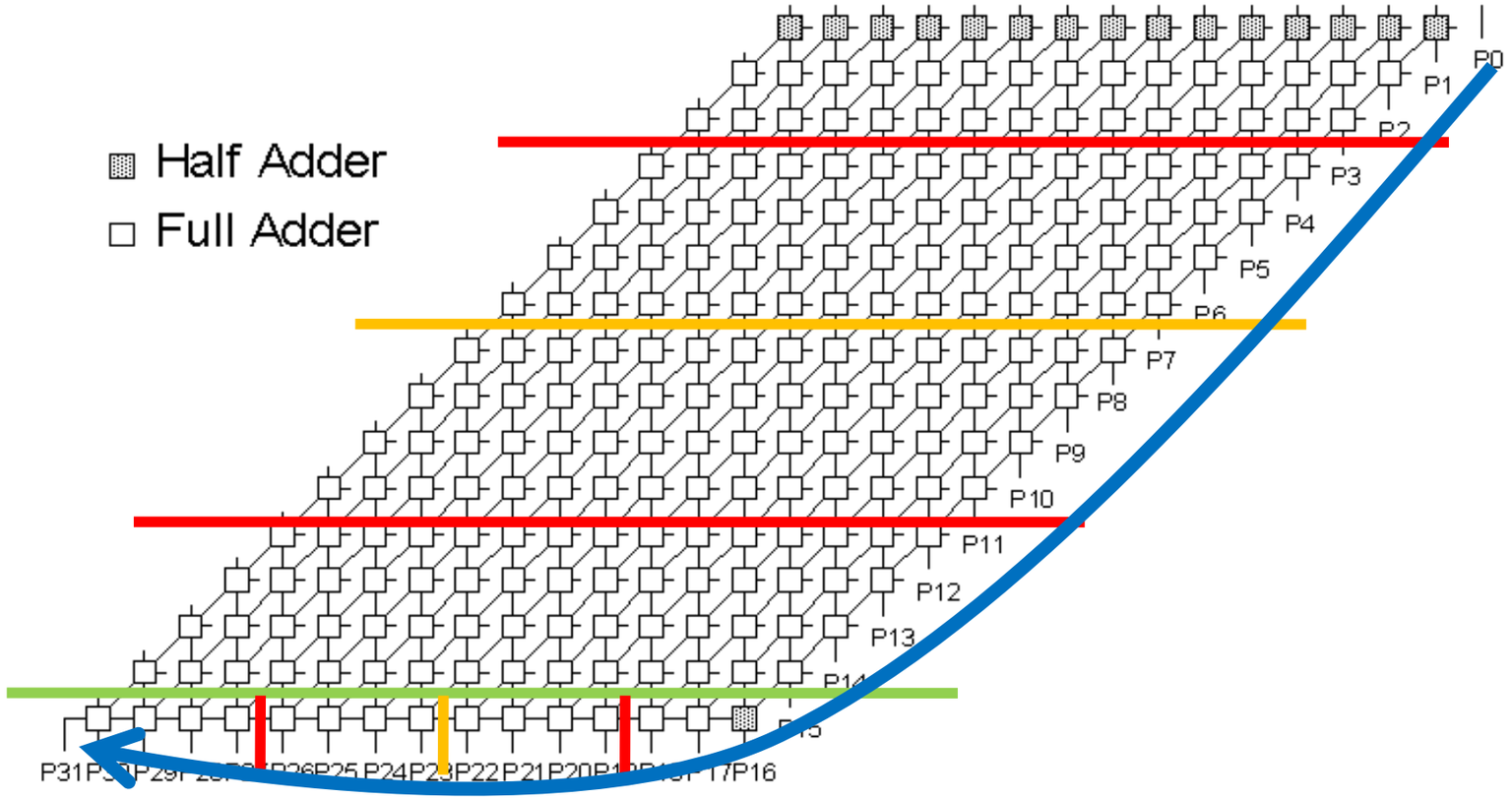
Operating a Pipeline





[Source: J. Hayes, Univ. of Michigan]

Example: Integer Multiplier



- 16x16 combinational multiplier
  - ISCAS-85 C6288 standard benchmark
- Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

Example: Integer Multiplier

Configuration	Delay	MPS	Area (FF/wiring)	Area Increase
Combinational	3.52ns	284	7535 (--/1759)	
2 Stages	1.87ns	534 (1.9x)	8725 (1078/1870)	16%
4 Stages	1.17ns	855 (3.0x)	11276 (3388/2112)	50%
8 Stages	0.80ns	1250 (4.4x)	17127 (8938/2612)	127%

- Pipeline efficiency
  - 2-stage: nearly double throughput; marginal area cost
  - 4-stage: 75% efficiency; area still reasonable
  - 8-stage: 55% efficiency; area more than doubles
- Tools: Synopsys DC/LSI Logic 110nm gflxp ASIC

18-600 Foundations of Computer Systems

Lecture 8: "Pipelined Processor Design"

John P. Shen & Gregory Kesden

September 25, 2017

Next Time ...

- Required Reading Assignment:
  - **Chapter 4 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**
- Recommended Reference:
  - **Chapters 1 and 2 of Shen and Lipasti (SnL).**

Electrical & Computer ENGINEERING