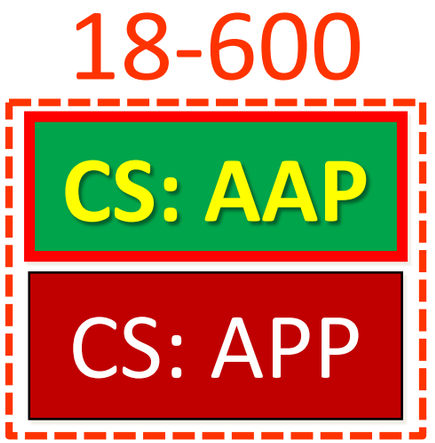
18-600 Foundations of Computer Systems

Lecture 8: "Processor Architecture and Design"

John P. Shen & Zhiyi Yu
September 26, 2016



- 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

18-600 Foundations of Computer Systems

Lecture 8: "Processor Architecture and Design"

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

Electrical & Computer ENGINEERING

Computer Instruction Set Architecture

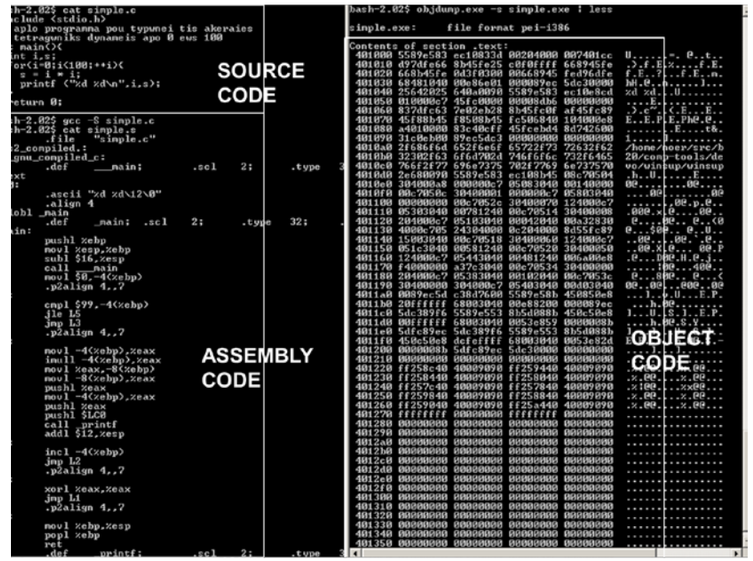
Software Engineering

CS: APP

Program Development

SPECIFICATION

Application Software
- Program development
- Program compilation



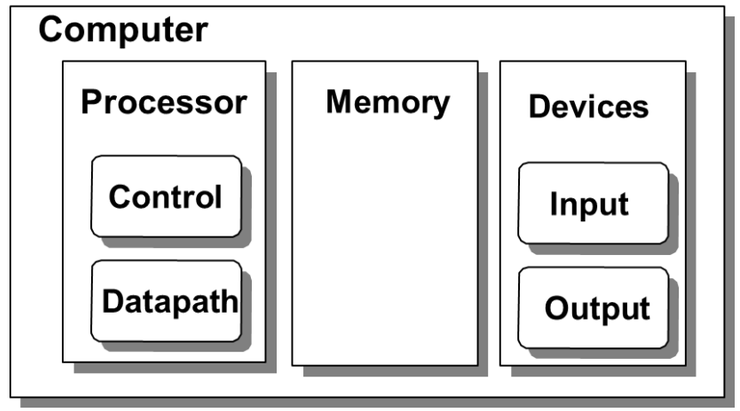
Instruction Set Architecture (ISA)

CS: AAP

Computer Engineering

Processor Design

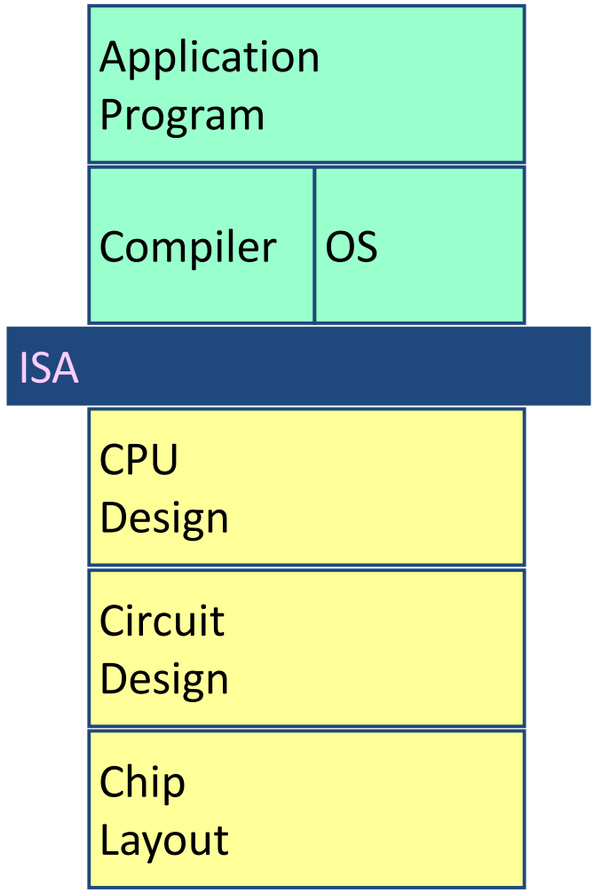
Hardware Technology
- Program execution
- Computer performance



IMPLEMENTATION

Instruction Set Architecture

- 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



CISC Instruction Sets

- Complex Instruction Set Computer
- IA32 is example

Stack-oriented instruction set

- Use stack to pass arguments, save program counter
- Explicit push and pop instructions

Arithmetic instructions can access memory

- `addq %rax, 12(%rbx,%rcx,8)`
  - requires memory read and write
  - Complex address calculation

Condition codes

- Set as side effect of arithmetic and logical instructions

Philosophy

- Add instructions to perform "typical" programming tasks

RISC Instruction Sets

- Reduced Instruction Set Computer
- Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)

Fewer, simpler instructions

- Might take more to get given task done
- Can execute them with small and fast hardware

Register-oriented instruction set

- Many more (typically 32) registers
- Use for arguments, return pointer, temporaries

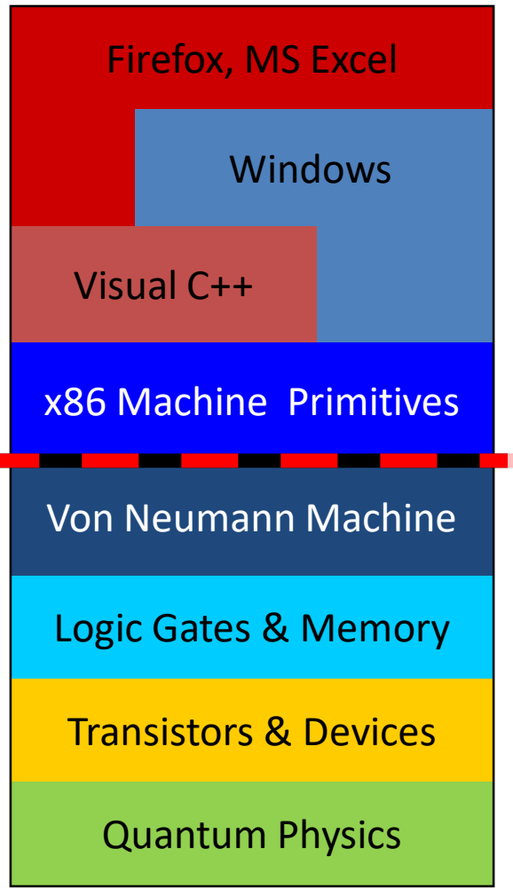
Only load and store instructions can access memory

- Similar to Y86-64 `mrmovq` and `rmmovq`

No Condition codes

- Test instructions return 0/1 in register

Computer Dynamic-Static Interface



PROGRAM

"static"

ARCHITECTURE

"dynamic"

MACHINE

Architectural state requirements:
- Support sequential instruction execution semantics.
- Support precise servicing of exceptions & interrupts.

Architectural State — Exposed to SW

Dynamic/Static Interface (DSI)=(ISA)

Microarchitecture State — Hidden in HW

Buffering needed between arch and uarch states:
- Allow uarch state to deviate from arch state.
- Able to undo speculative uarch state if needed.

DSI = ISA = a contract between the program and the machine.

CISC vs. RISC

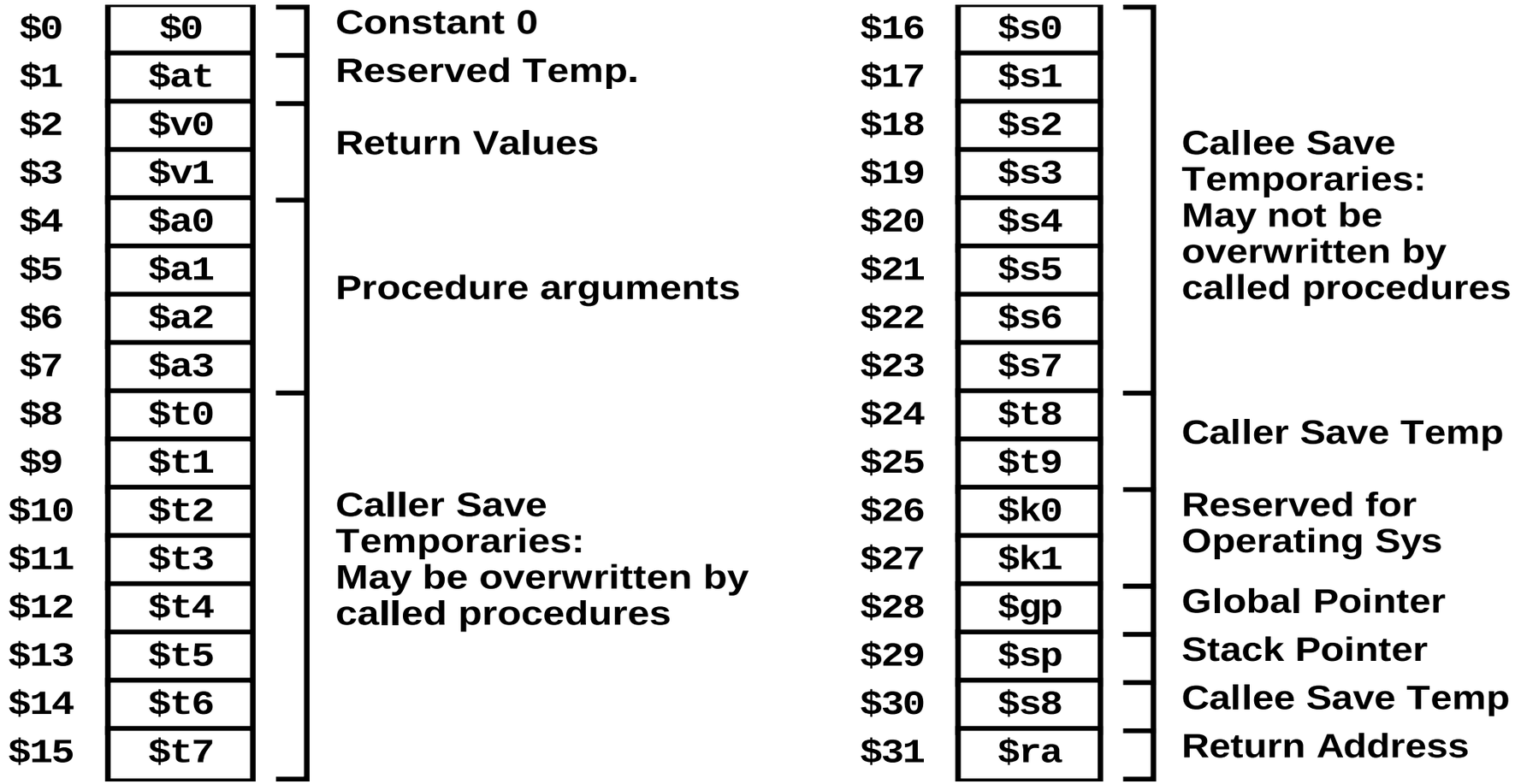
Original Debate

- Strong opinions!
- CISC proponents---easy for compiler, fewer code bytes
- RISC proponents---better for optimizing compilers, can make machine run fast with simple chip design

Current Status

- For desktop processors, choice of ISA not a technical issue
  - With enough hardware, can make anything run fast
  - Code compatibility more important
- x86-64 adopted many RISC features
  - More registers; use them for argument passing
- For embedded processors, RISC makes sense
  - Smaller, cheaper, less power
  - Most mobile phones use ARM processors

MIPS Registers



MIPS Instruction Examples

R-R


```
addu $3,$2,$1        # Register add: $3 = $2+$1
```

R-I


```
addu $3,$2, 3145     # Immediate add: $3 = $2+3145
sll $3,$2,2          # Shift left: $3 = $2 << 2
```

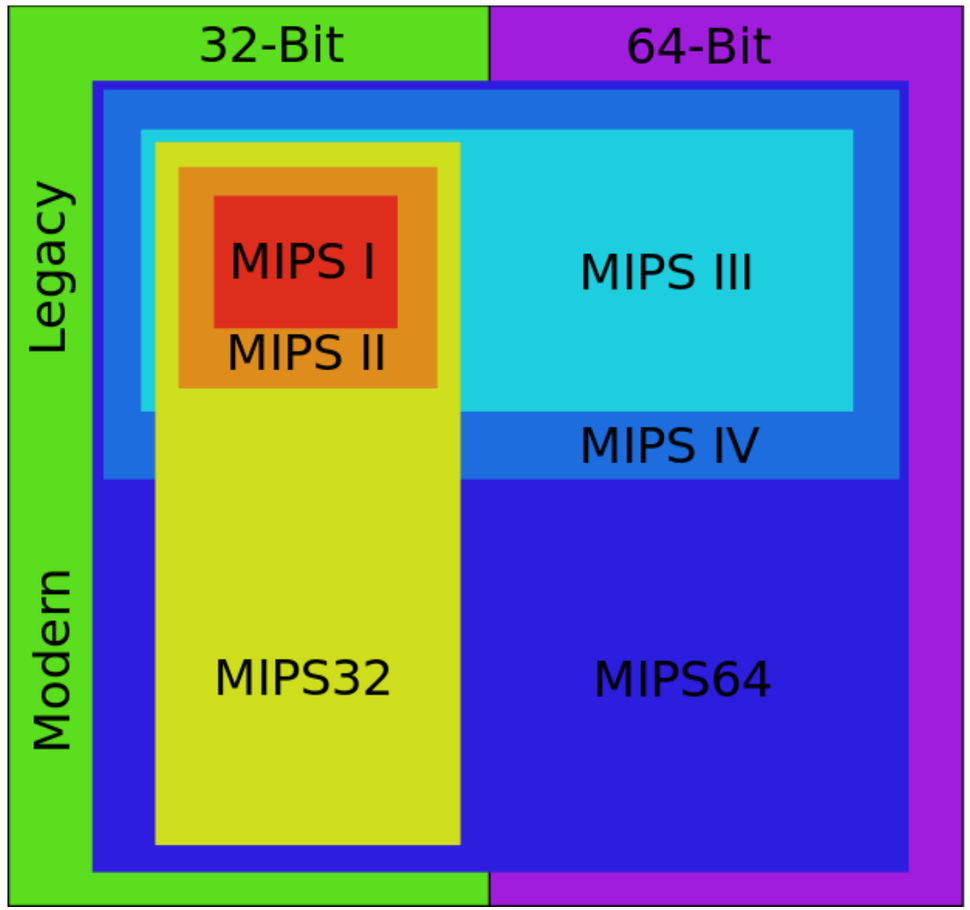
Branch

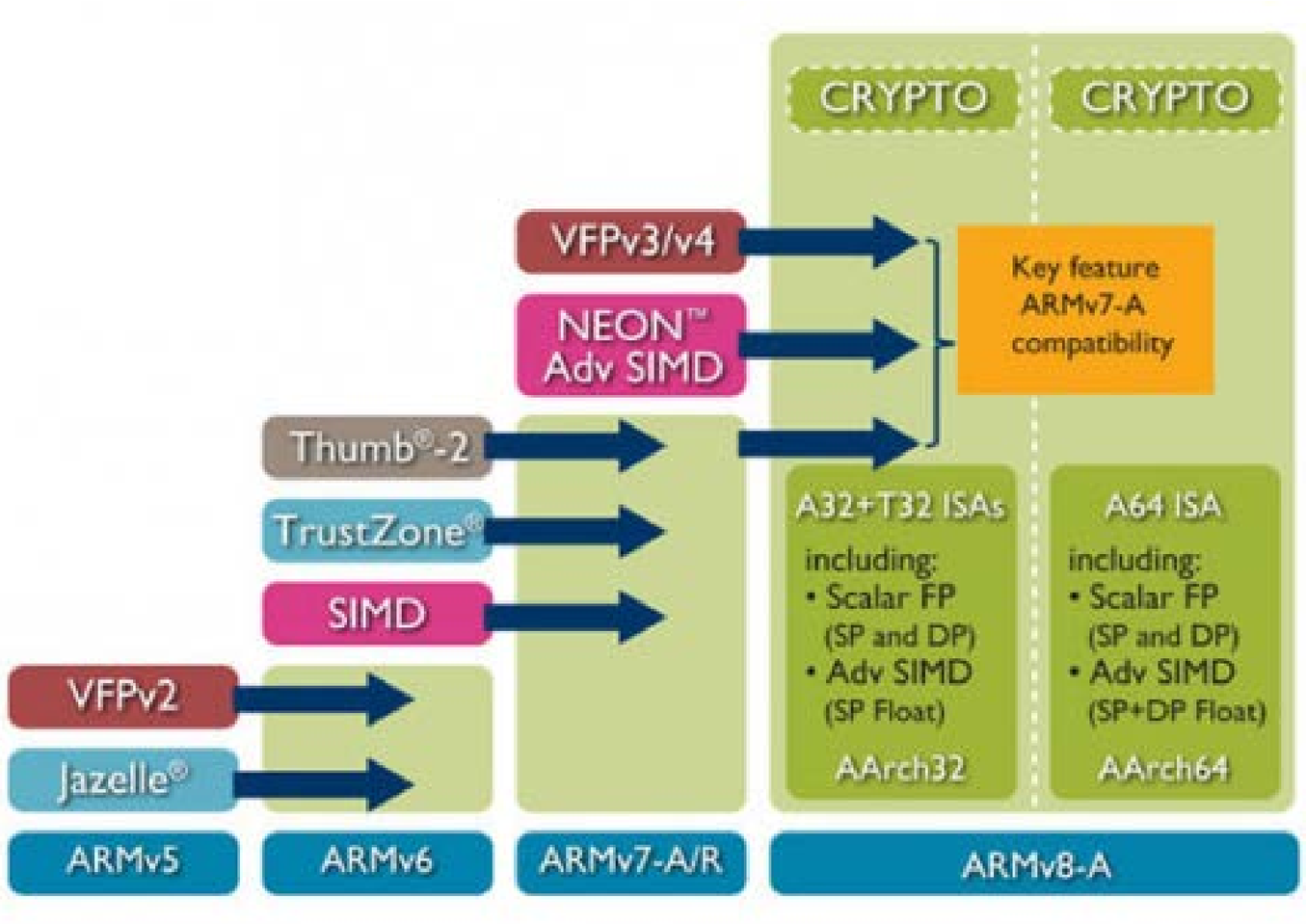

```
beq $3,$2,dest       # Branch when $3 = $2
```

Load/Store


```
lw $3,16($2)         # Load Word: $3 = M[$2+16]
sw $3,16($2)         # Store Word: M[$2+16] = $3
```

Evolutions of MIPS and ARM ISAs





ISA Design Summary

How Important is ISA Design?

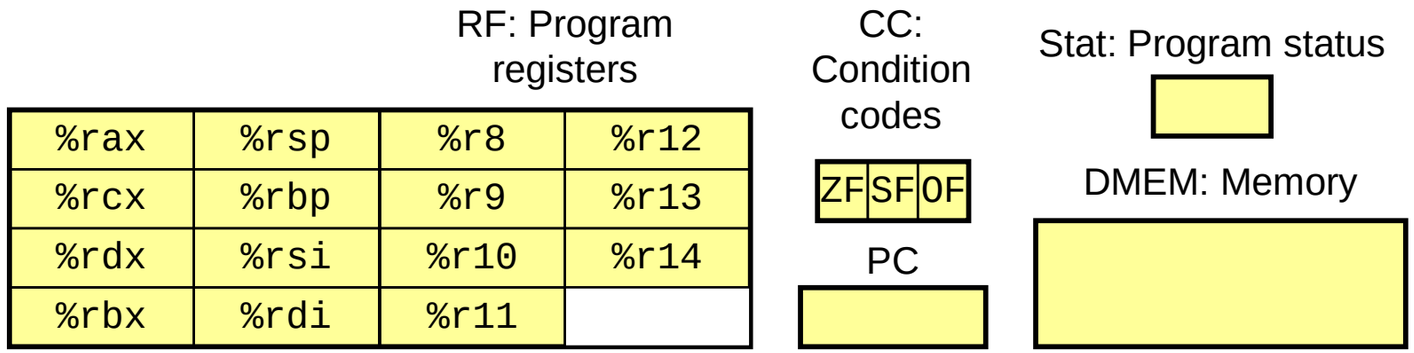
- Less now than before
  - With enough hardware, can make almost anything go fast
  - Installed based of legacy code and SW development tools dominate

Y86-64 Instruction Set Architecture

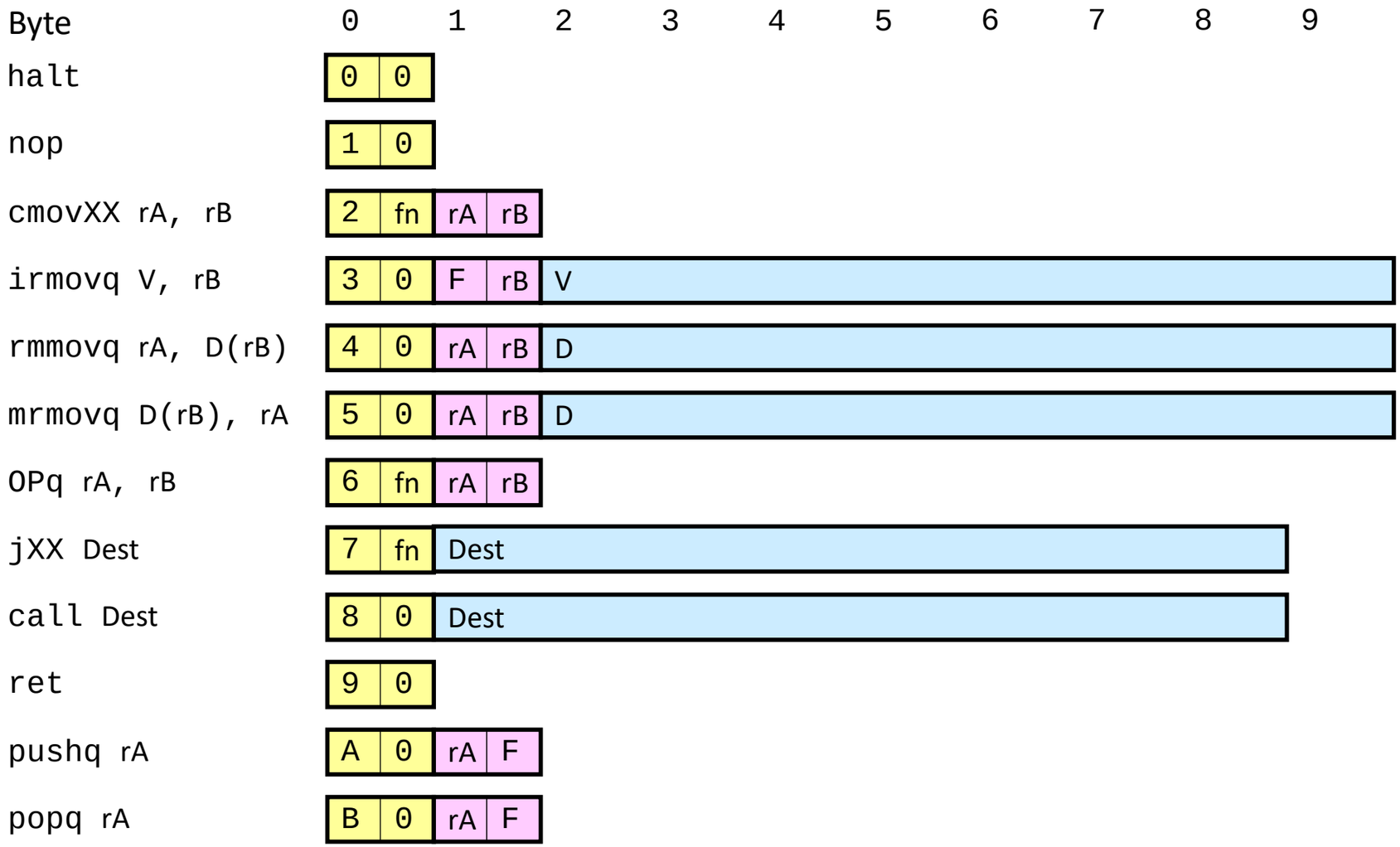
- Similar state and instructions as x86-64
- Simpler encodings
- Somewhere between CISC and RISC

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



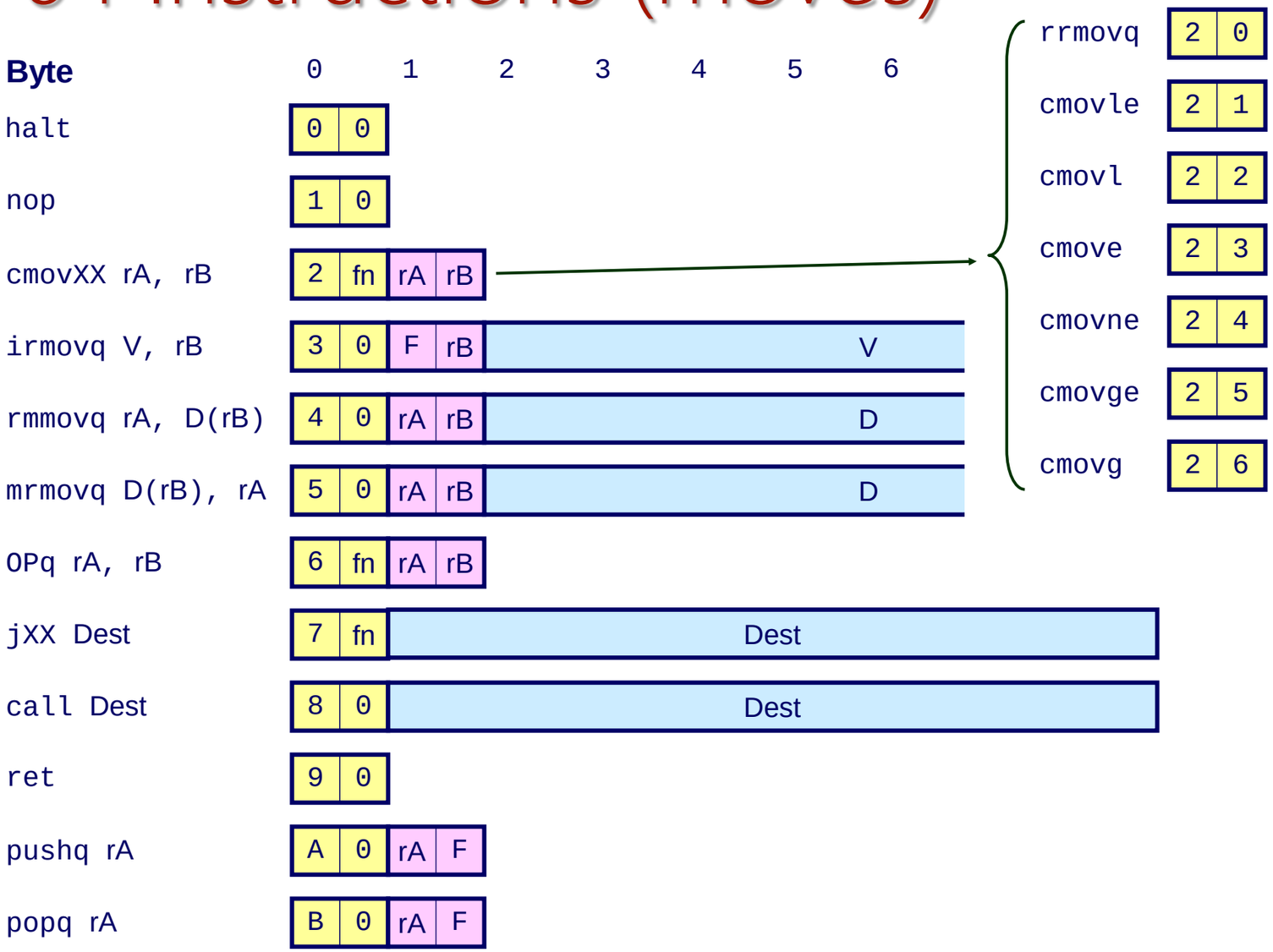
Y86-64 Instruction Set



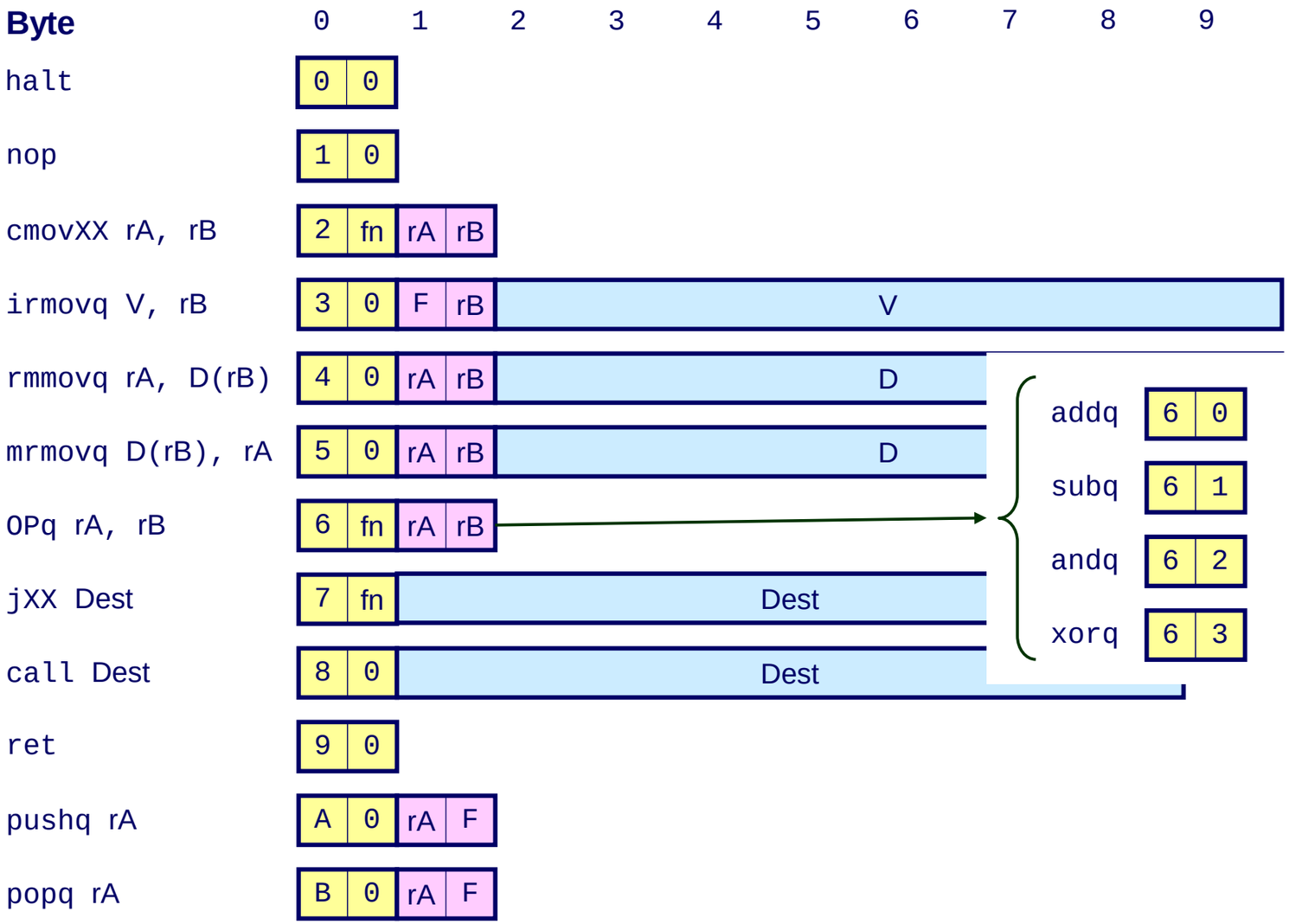
Y86-64 Instruction Formats

- 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

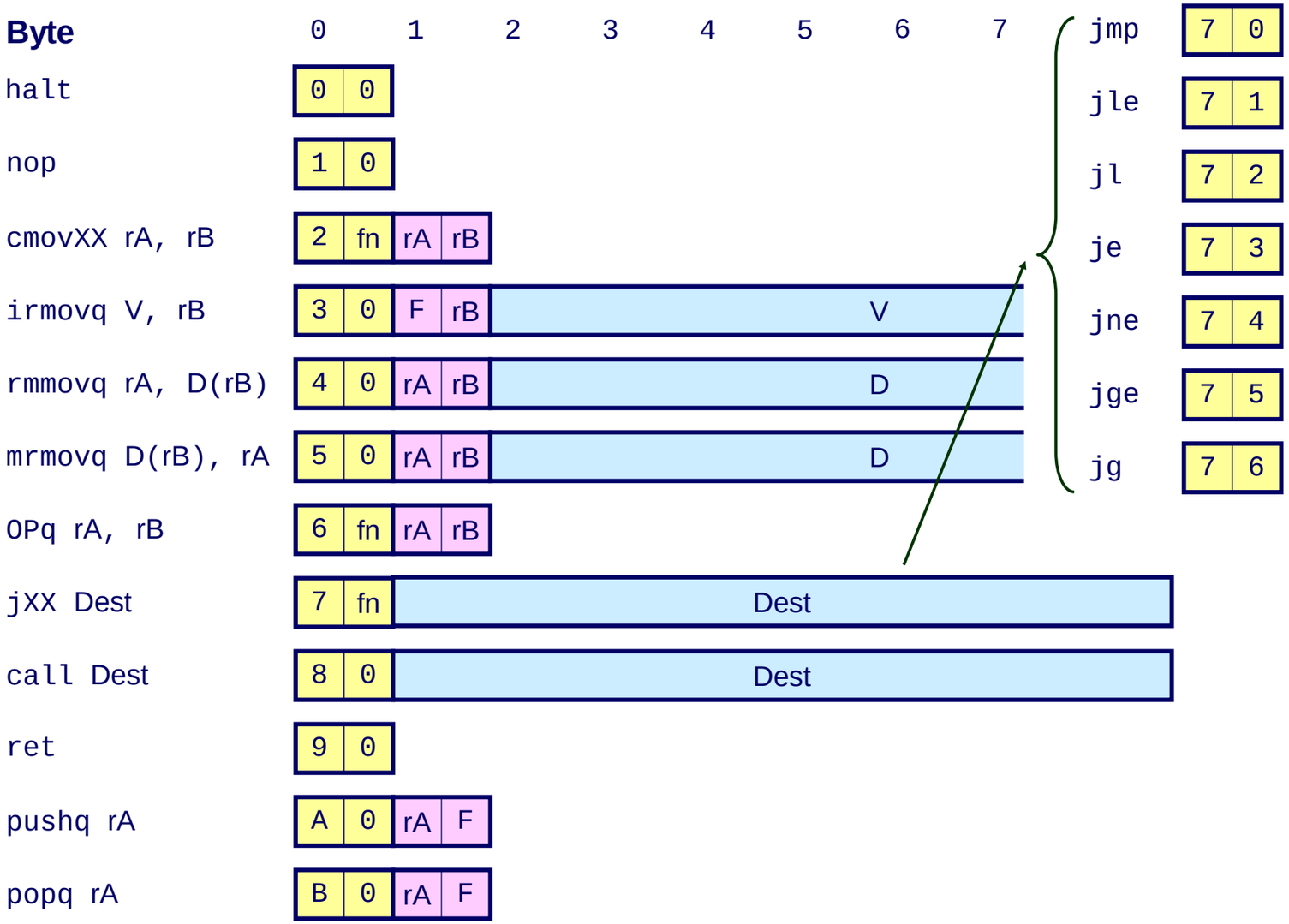
Y86-64 Instructions (moves)



Y86-64 Instructions (ALUs)



Y86-64 Instructions (branches)



Encoding Register Specifiers

- Each register has 4-bit ID

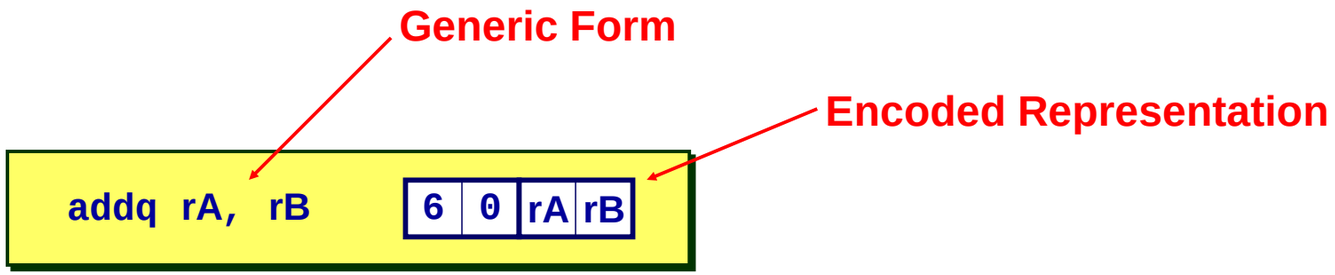
%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7

%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

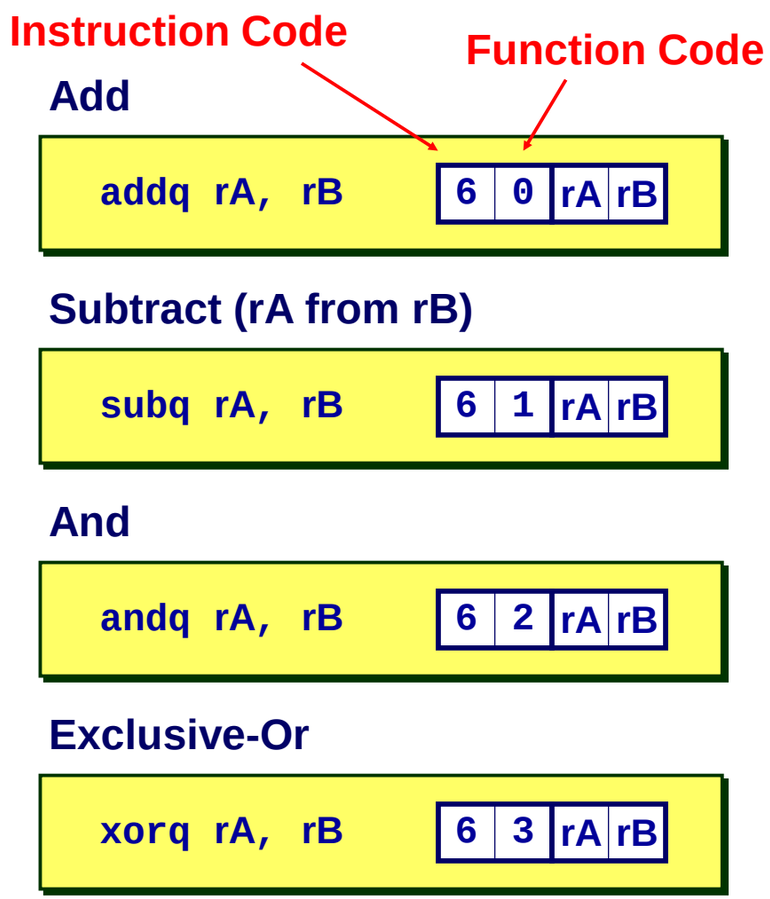
Instruction Format Example

Addition Instruction



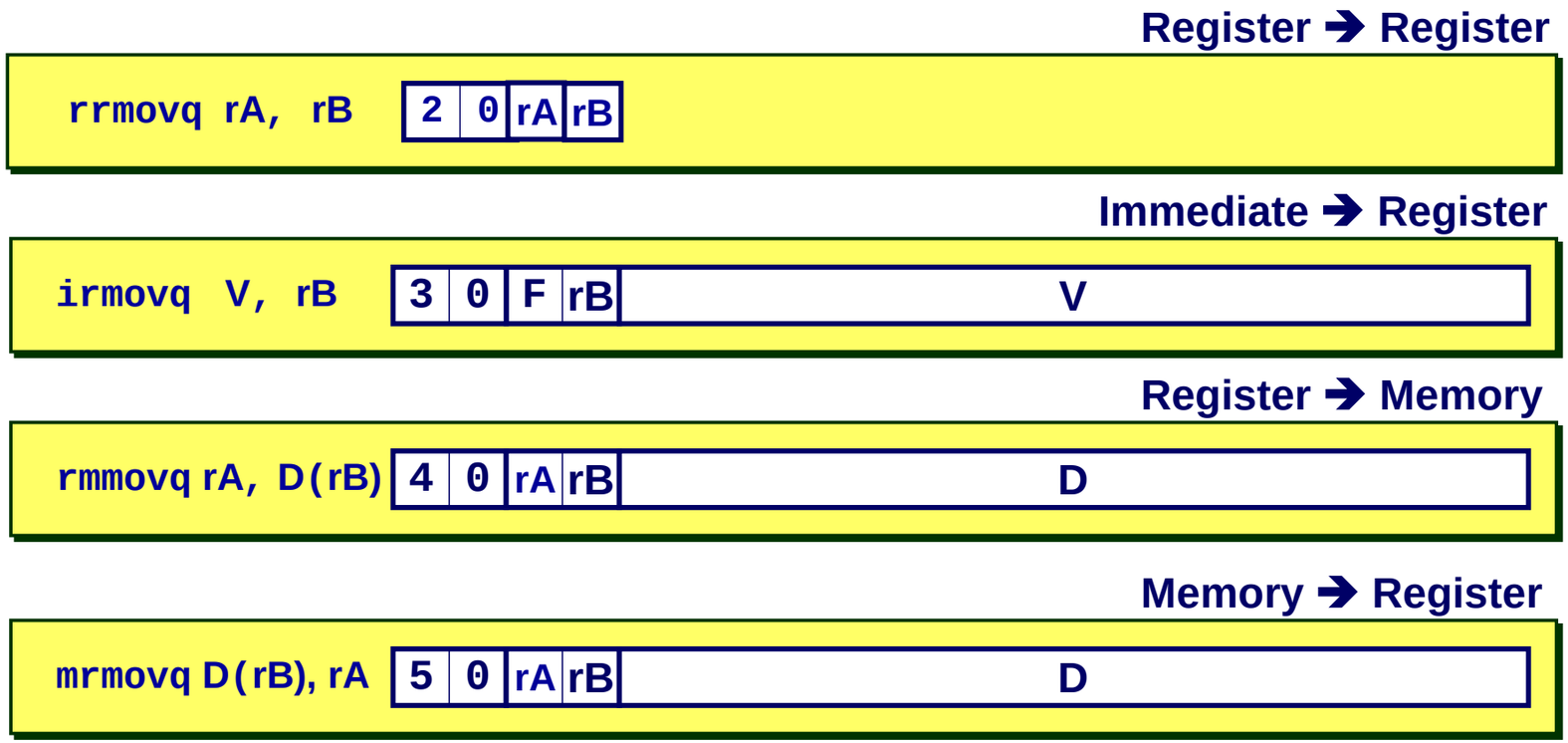
- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- e.g., `addq %rax,%rsi` Encoding: `60 06`
- Two-byte encoding
  - First indicates instruction type
  - Second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

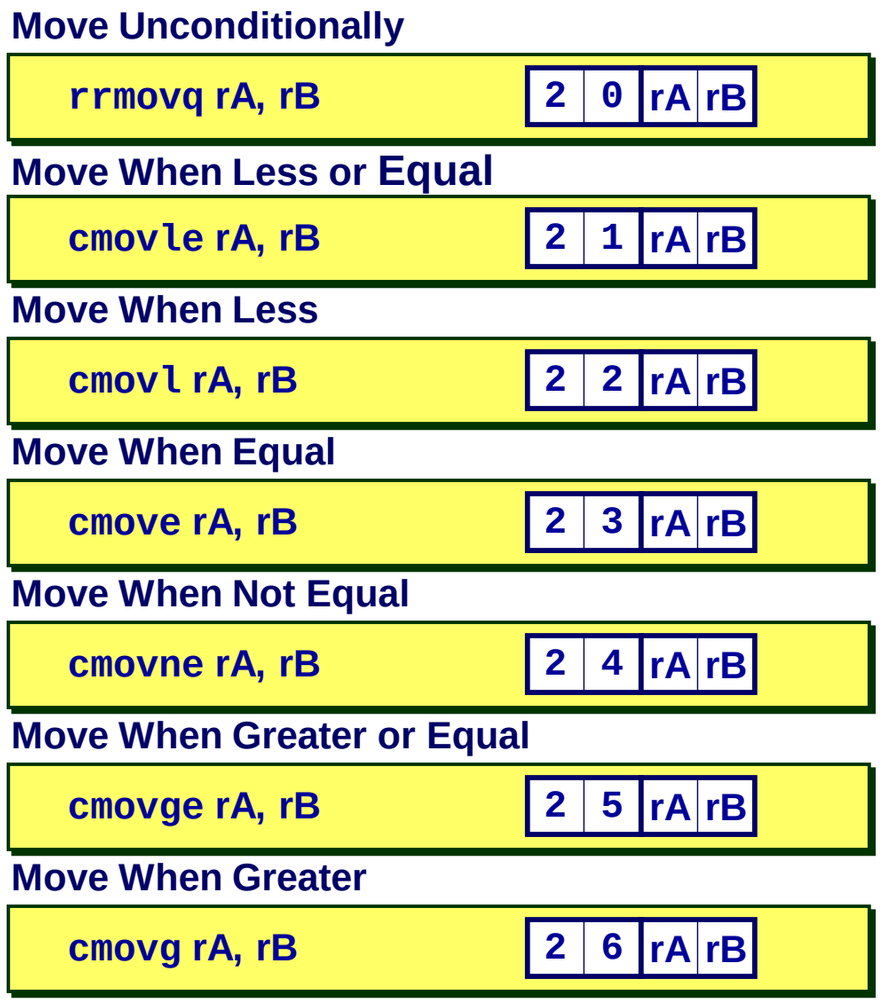


- Like the x86-64 `movq` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

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

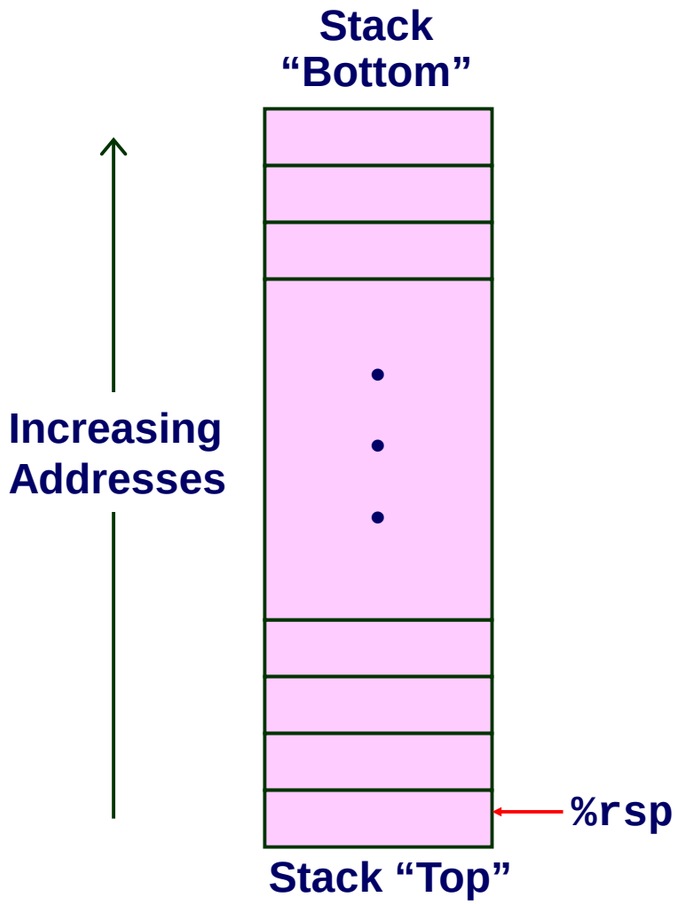
Jump When Equal

Jump When Not Equal

Jump When Greater or Equal

Jump When Greater

Y86-64 Program Stack



- 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

Stack Operations

pushq rA

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

popq rA

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

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

Miscellaneous Instructions

- Don't do anything

- Stop executing instructions
- x86-64 has comparable instruction, but can't execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

Status Conditions

Mnemonic	Code
AOK	1

- **Normal operation**

Mnemonic	Code
HLT	2

- **Halt instruction encountered**

Mnemonic	Code
ADR	3

- **Bad address (either instruction or data) encountered**

Mnemonic	Code
INS	4

- **Invalid instruction encountered**

Desired Behavior

- **If AOK, keep going**
- **Otherwise, stop program execution**

Writing Y86-64 Code

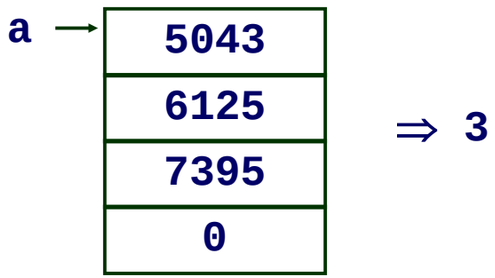
Try to Use C Compiler as Much as Possible

- Write code in C
- Compile for x86-64 with `gcc –Og –S`
- Transliterate into Y86-64
- *Modern compilers make this more difficult*

Coding Example

- Find number of elements in null-terminated list

```
int len1(int a[]);
```



⇒ 3

Y86-64 Code Generation Example

First Try

- Write typical array code

```
/* Find number of elements in
   null-terminated list */
long len(long a[])
{
  long len;
  for (len = 0; a[len]; len++)
       ;
  return len;
}
```

- Compile with `gcc -Og -S`

Problem

- Hard to do array indexing on Y86-64
  - Since don't have scaled addressing modes

```
L3:
   addq $1,%rax
   cmpq  $0, (%rdi,%rax,8)
   jne L3
```

Y86-64 Code Generation Example #2

Second Try

- Write C code that mimics expected Y86-64 code

```
long len2(long *a)
{
    long ip = (long) a;
    long val = *(long *) ip;
    long len = 0;
    while (val) {
        ip += sizeof(long);
        len++;
        val = *(long *) ip;
    }
    return len;
}
```

Result

- Compiler generates exact same code as before!
- Compiler converts both versions into same intermediate form

Y86-64 Code Generation Example #3

```
len:
    irmovq $1, %r8          # Constant 1
    irmovq $8, %r9          # Constant 8
    irmovq $0, %rax         # len = 0
    mrmovq (%rdi), %rdx     # val = *a
    andq %rdx, %rdx         # Test val
    je Done                 # If zero, goto Done
Loop:
    addq %r8, %rax          # len++
    addq %r9, %rdi          # a++
    mrmovq (%rdi), %rdx     # val = *a
    andq %rdx, %rdx         # Test val
    jne Loop                # If !0, goto Loop
Done:
    ret
```

Register	Use
%rdi	a
%rax	len
%rdx	val
%r8	1
%r9	8

Y86-64 Sample Program Structure #1

```
init:                   # Initialization
   . . .
   call Main
   halt

   .align 8             # Program data
array:
   . . .

Main:                   # Main function
   . . .
   call len   . . .

len:                    # Length function
   . . .

   .pos 0x100           # Placement of stack
Stack:
```

- Program starts at address 0
- Must set up stack
  - Where located
  - Pointer values
  - Make sure don't overwrite code!
- Must initialize data

Y86-64 Program Structure #2

```
init:
      # Set up stack pointer
      irmovq Stack, %rsp
      # Execute main program
      call Main
      # Terminate
      halt

# Array of 4 elements + terminating 0
      .align 8
Array:
      .quad 0x000d000d000d000d
      .quad 0x00c000c000c000c0
      .quad 0x0b000b000b000b00
      .quad 0xa000a000a000a000
      .quad 0
```

- Program starts at address 0
- Must set up stack
- Must initialize data
- Can use symbolic names

Y86-64 Program Structure #3

```
Main:
      irmovq array,%rdi
      # call len(array)
      call len
      ret
```

Set up call to len

- **Follow x86-64 procedure conventions**
- **Push array address as argument**

Assembling Y86-64 Program

```
unix> yas len.ys
```

- **Generates "object code" file `len.yo`**
  - **Actually looks like disassembler output**

```
0x054:                      | len:
0x054: 30f80100000000000000 |   irmovq $1, %r8          # Constant 1
0x05e: 30f90800000000000000 |   irmovq $8, %r9          # Constant 8
0x068: 30f00000000000000000 |   irmovq $0, %rax         # len = 0
0x072: 50270000000000000000 |   mrmovq (%rdi), %rdx     # val = *a
0x07c: 6222                 |   andq %rdx, %rdx         # Test val
0x07e: 73a000000000000000   |   je Done                 # If zero, goto Done
0x087:                      | Loop:
0x087: 6080                 |   addq %r8, %rax          # len++
0x089: 6097                 |   addq %r9, %rdi          # a++
0x08b: 50270000000000000000 |   mrmovq (%rdi), %rdx     # val = *a
0x095: 6222                 |   andq %rdx, %rdx         # Test val
0x097: 748700000000000000   |   jne Loop                # If !0, goto Loop
0x0a0:                      | Done:
0x0a0: 90                   |   ret
```

Simulating Y86-64 Program

```
unix> yis len.yo
```

- **Instruction set simulator**
  - **Computes effect of each instruction on processor state**
  - **Prints changes in state from original**

```
Stopped in 33 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000004
%rsp:   0x0000000000000000      0x0000000000000100
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000001
%r9:    0x0000000000000000      0x0000000000000008

Changes to memory:
0x00f0: 0x0000000000000000      0x0000000000000053
0x00f8: 0x0000000000000000      0x0000000000000013
```

18-600 Foundations of Computer Systems

Lecture 8: "Processor Architecture and Design"

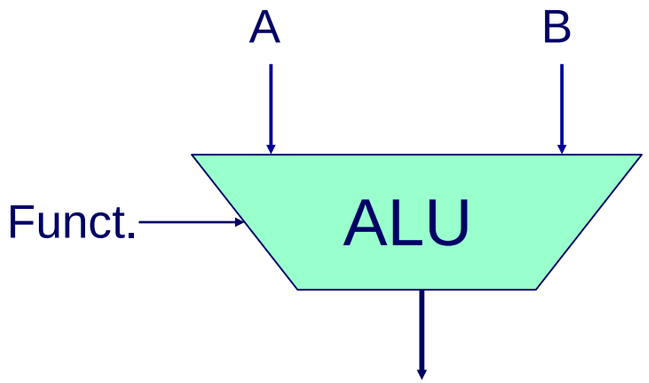
1. **Processor Architecture**
   a. Instruction Set Architecture (ISA)
   b. Instruction Set Designs
   c. Y86-64 Instruction Set
2. **Processor Implementation**
   a. Instruction Set Processor (CPU)
   b. Processor Organization Design
   c. Y86-64 Sequential Processor
3. **Motivation for Pipelining**

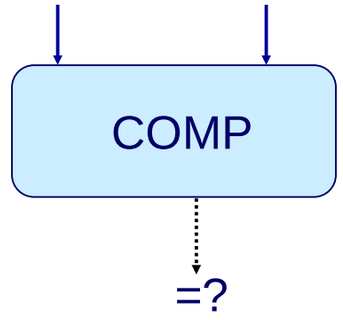
Electrical & Computer ENGINEERING

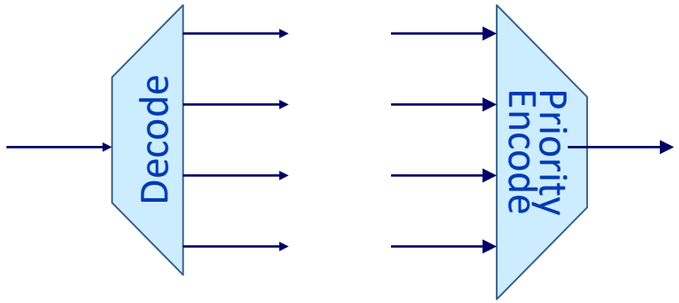
Building Blocks

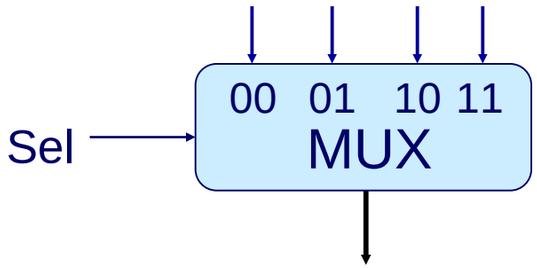
Combinational Logic

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



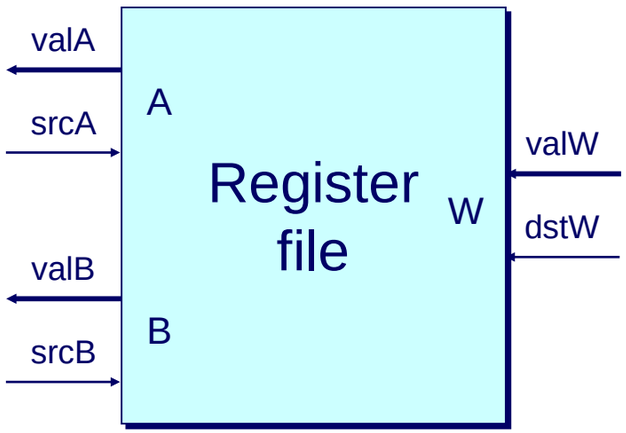


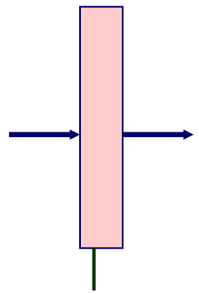




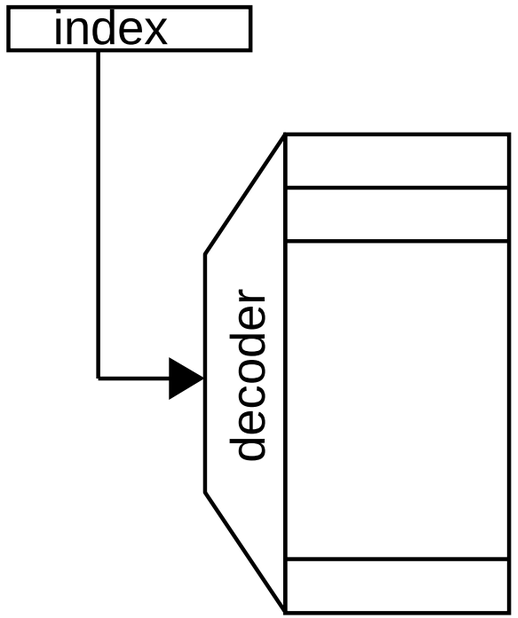
Storage Elements

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



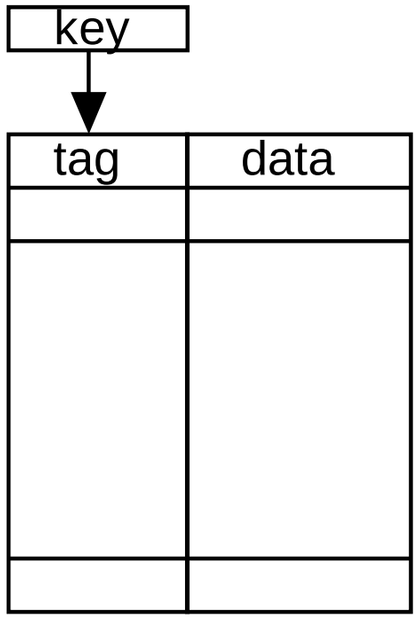


(Cache) Memory Implementation Options



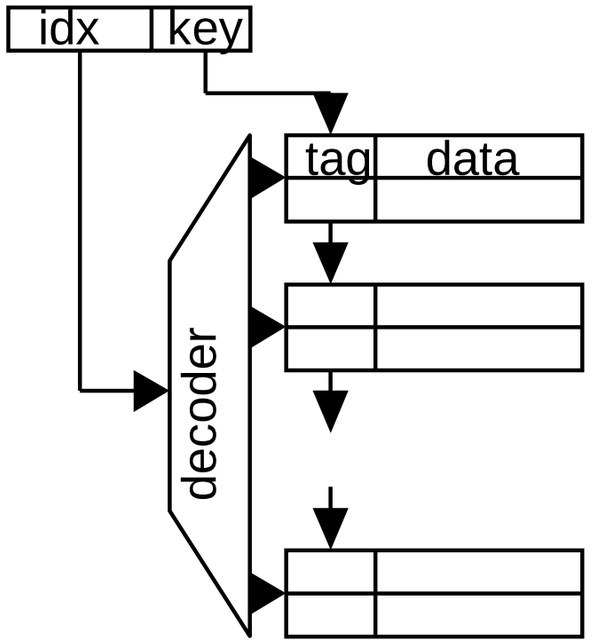
Indexed Memory

k-bit index
2^k blocks



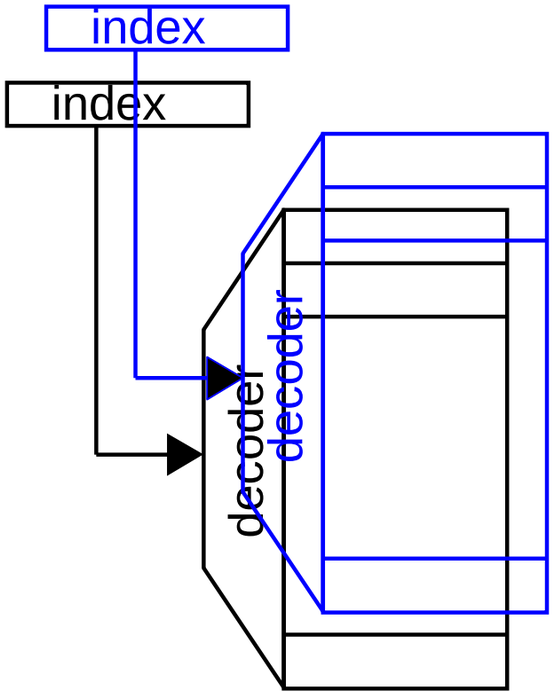
Associative Memory (CAM)

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

Hardware Control Language

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

Data Types

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

Statements

- `bool a =` ***`bool-expr`*** `;`
- `int A =` ***`int-expr`*** `;`

HCL Operations

- Classify by type of value returned

Boolean Expressions

- Logic Operations
  - `a && b, a || b, !a`
- Word Comparisons
  - `A == B, A != B, A < B, A <= B, A >= B, A > B`
- Set Membership
  - `A in { B, C, D }`
    - » Same as `A == B || A == C || A == D`

Word Expressions

- Case expressions
  - `[ a : A; b : B; c : C ]`
  - Evaluate test expressions a, b, c, … in sequence
  - Return word expression A, B, C, … for first successful test

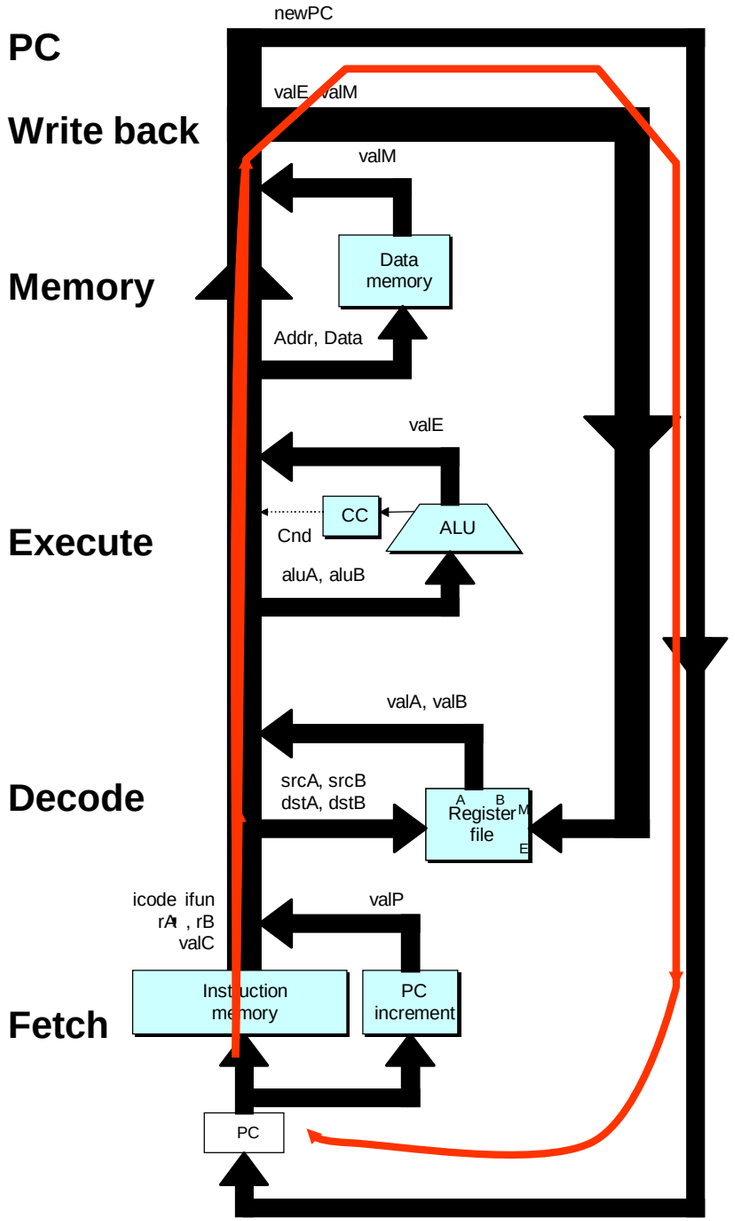
SEQ Hardware Structure

State

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

Instruction Flow

- Read instruction at address specified by PC
- Process through stages
- Update program counter



SEQ Stages

Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

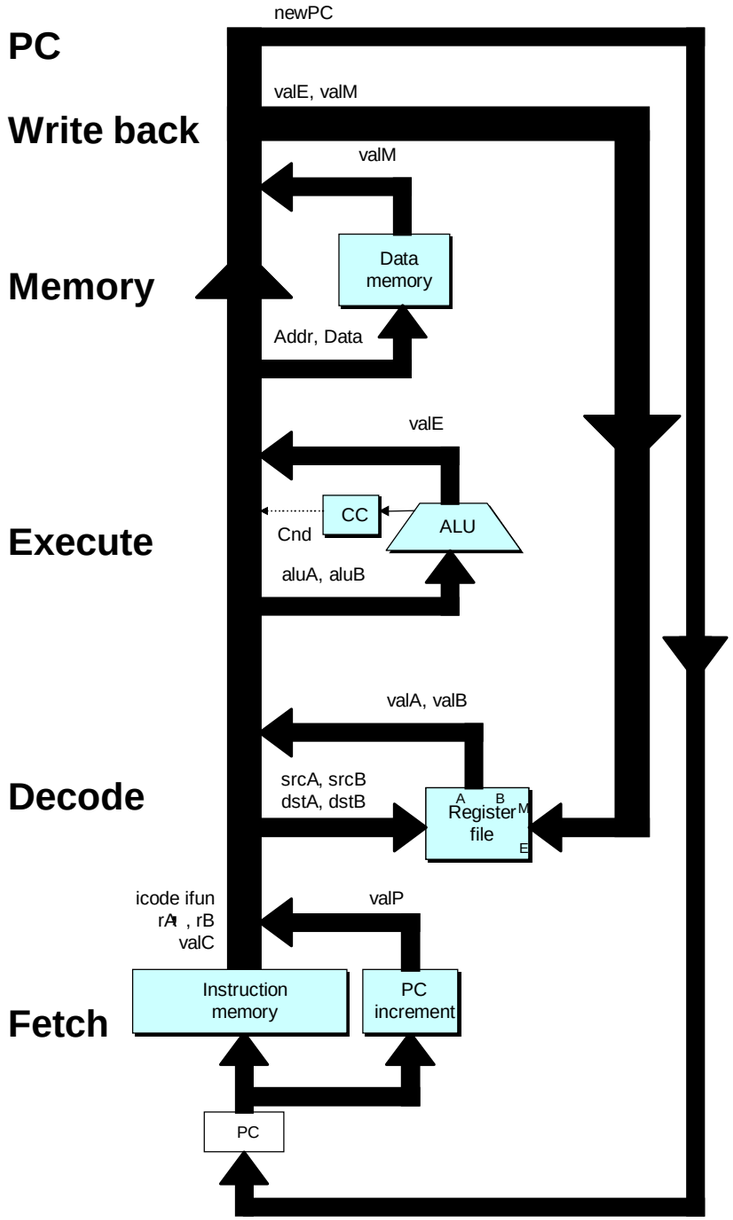
- Read or write data

Write Back

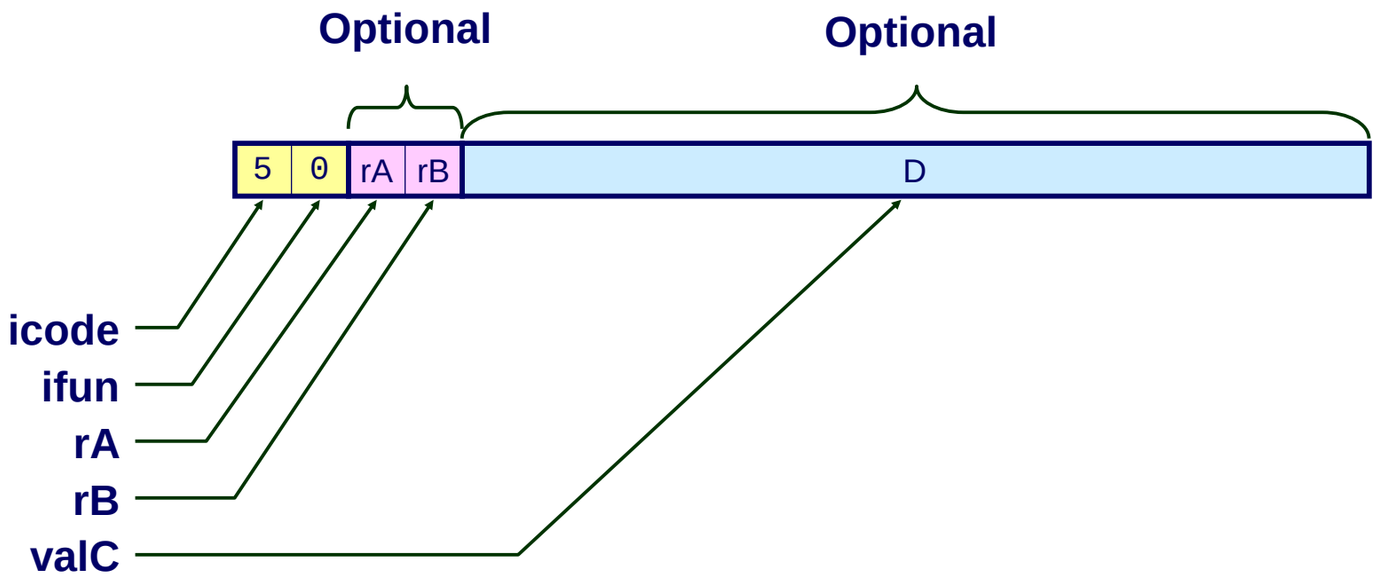
- Write program registers

PC

- Update program counter



Instruction Decoding



Instruction Format

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

Executing Arithmetic/Logical 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

Stage Computation: `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

Stage Computation: 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

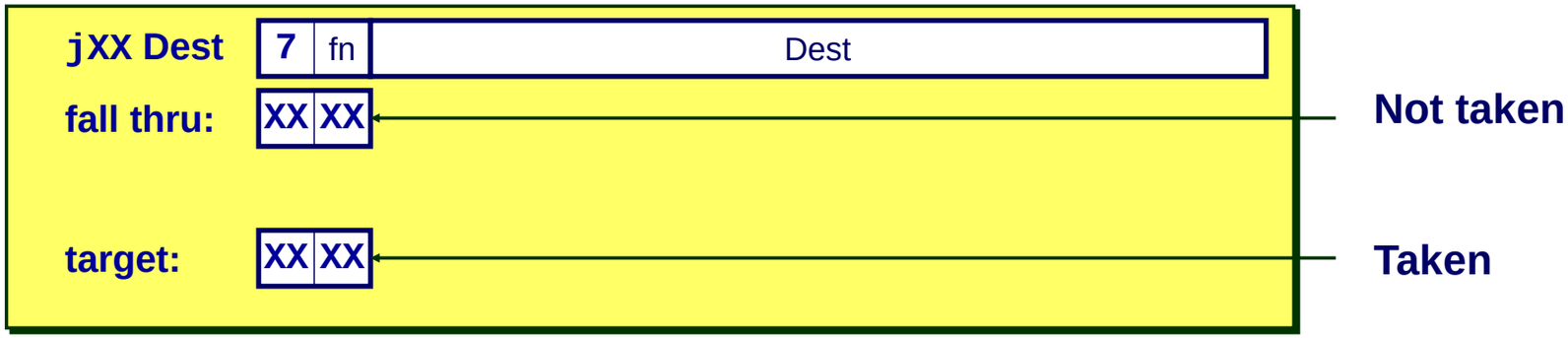
- Increment PC by 2

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)
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

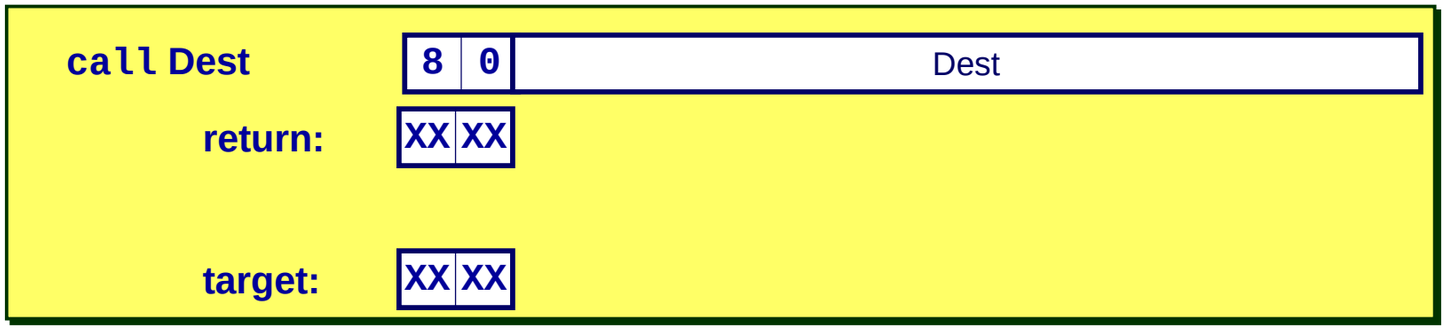
- Set PC to Dest if branch taken or to incremented PC if not branch

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

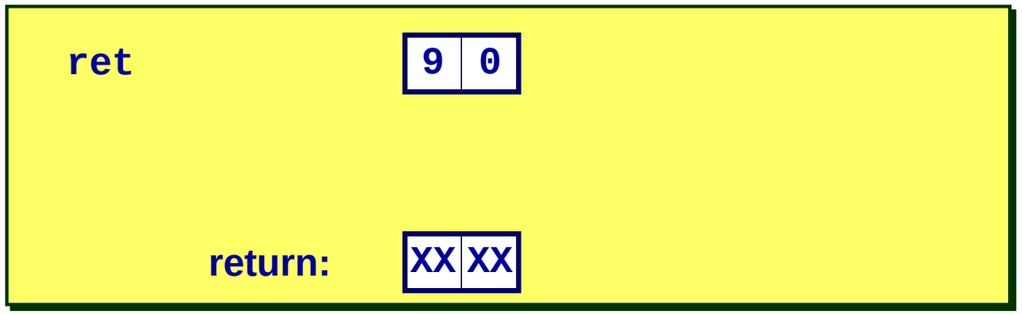
- Set PC to Dest

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

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

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

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

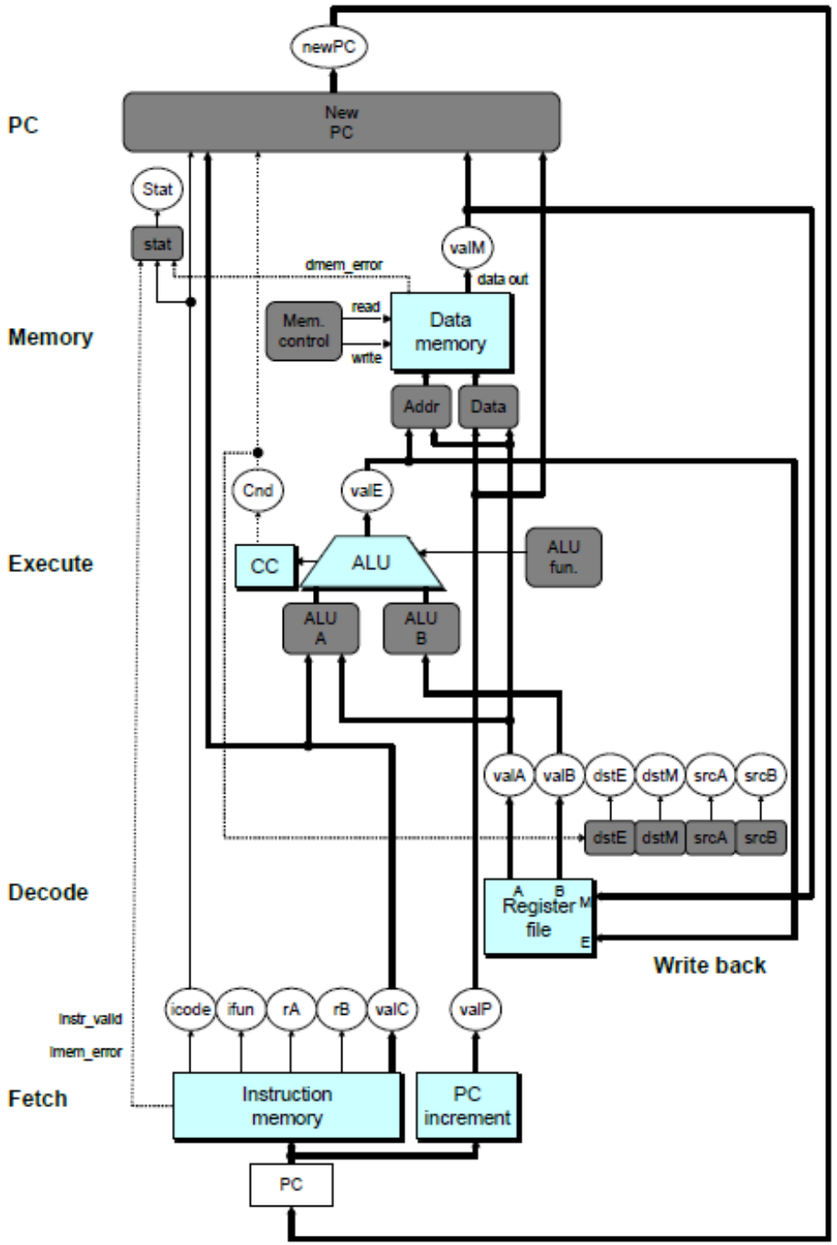
Memory

- **valM** Value from memory

SEQ Hardware

Key

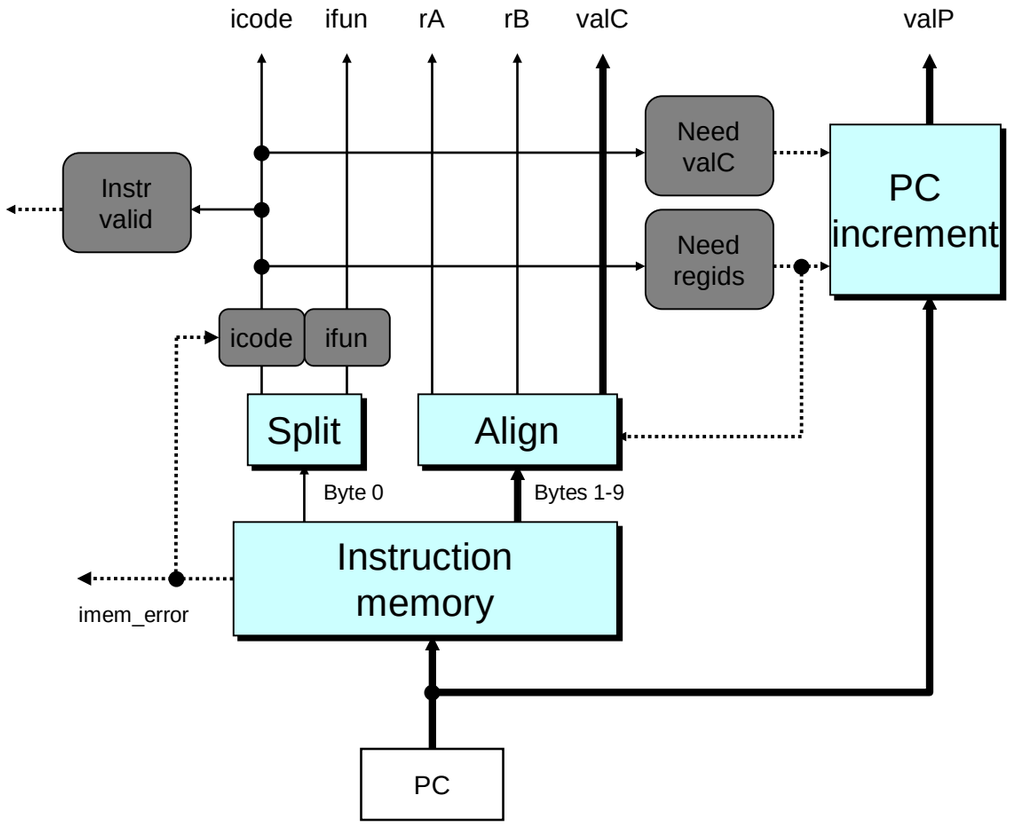
- Blue boxes: predesigned hardware blocks
  - E.g., memories, ALU
- Gray boxes: control logic
  - Describe in HCL
- White ovals: labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



Fetch Logic

Predefined Blocks

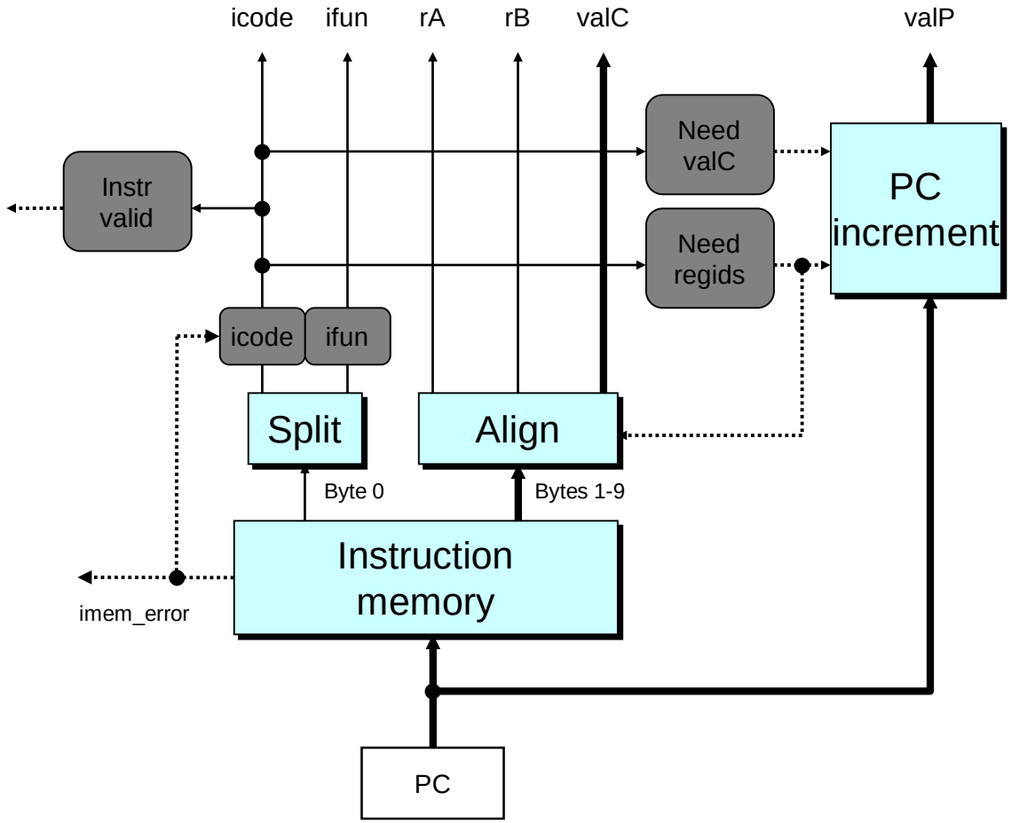
- PC: Register containing PC
- Instruction memory: Read 10 bytes (PC to PC+9)
  - Signal invalid address
- Split: Divide instruction byte into icode and ifun
- Align: Get fields for rA, rB, and valC



Fetch Logic

Control Logic

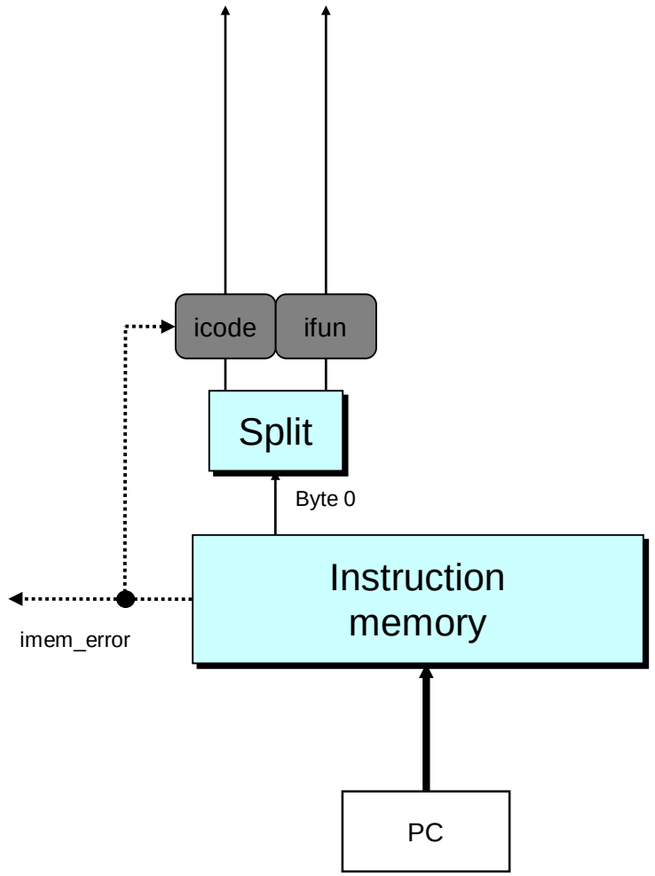
- Instr. Valid: Is this instruction valid?
- icode, ifun: Generate no-op if invalid address
- Need regids: Does this instruction have a register byte?
- Need valC: Does this instruction have a constant word?



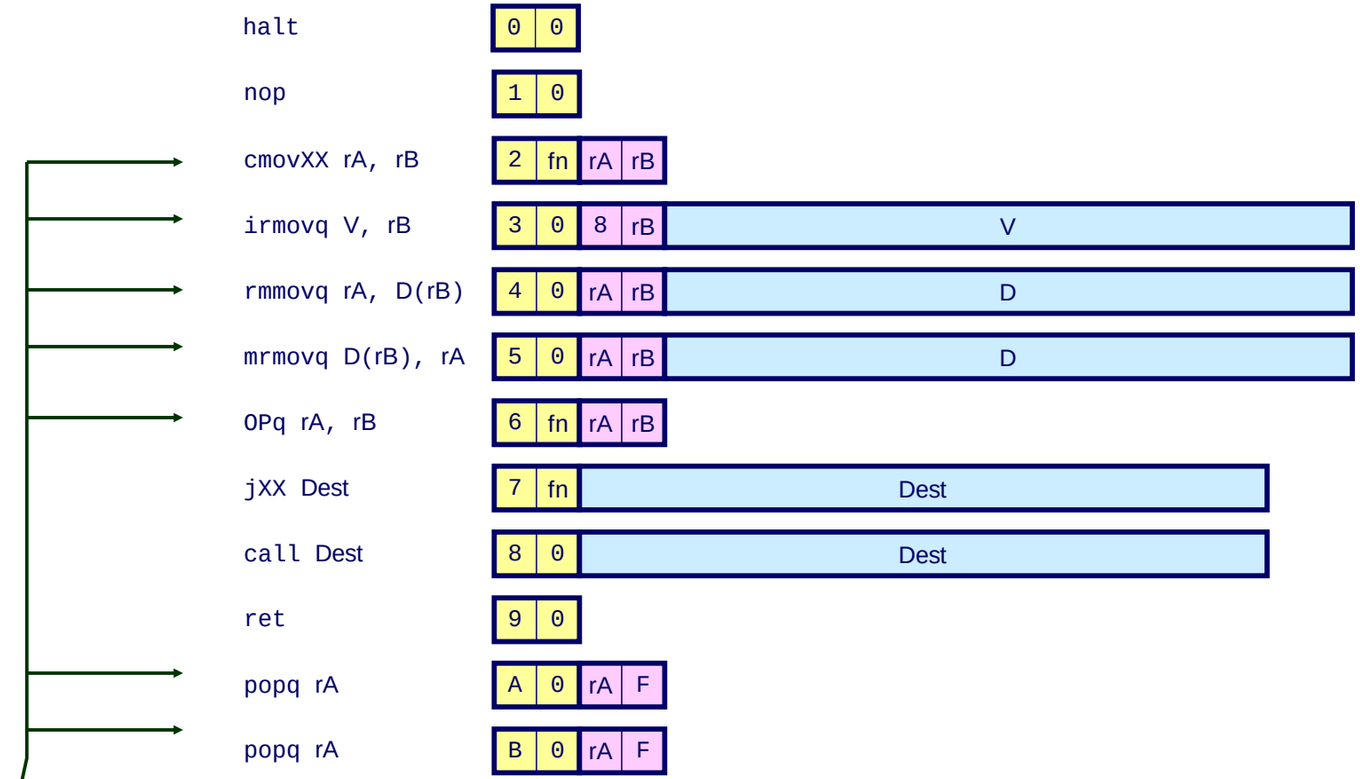
Fetch Control Logic in HCL

```
# 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




```
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 };
```

Decode Logic

Register File

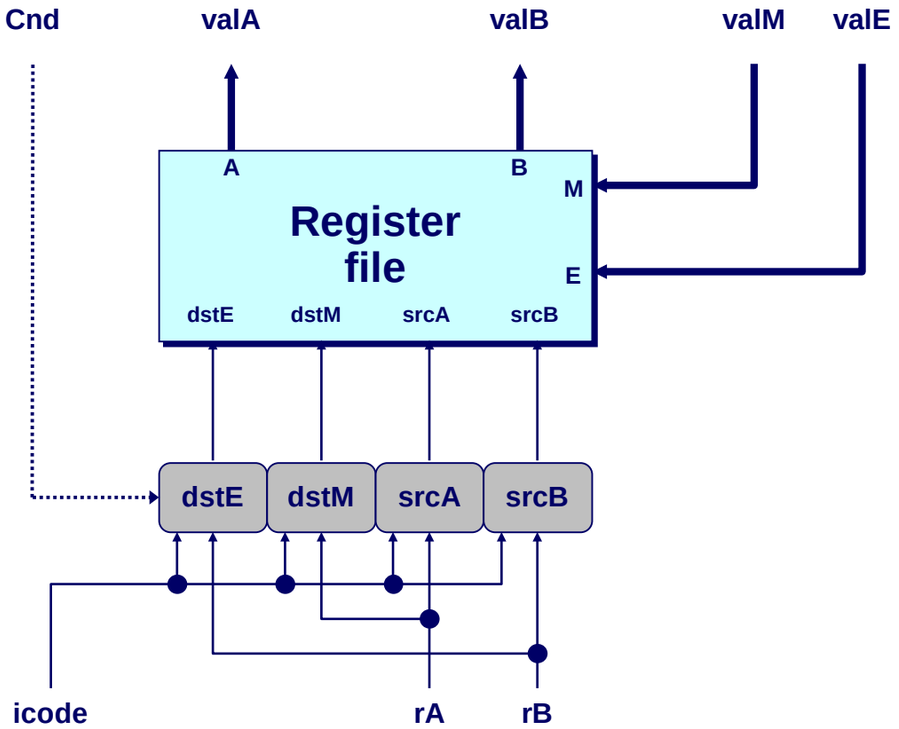
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

Control Logic

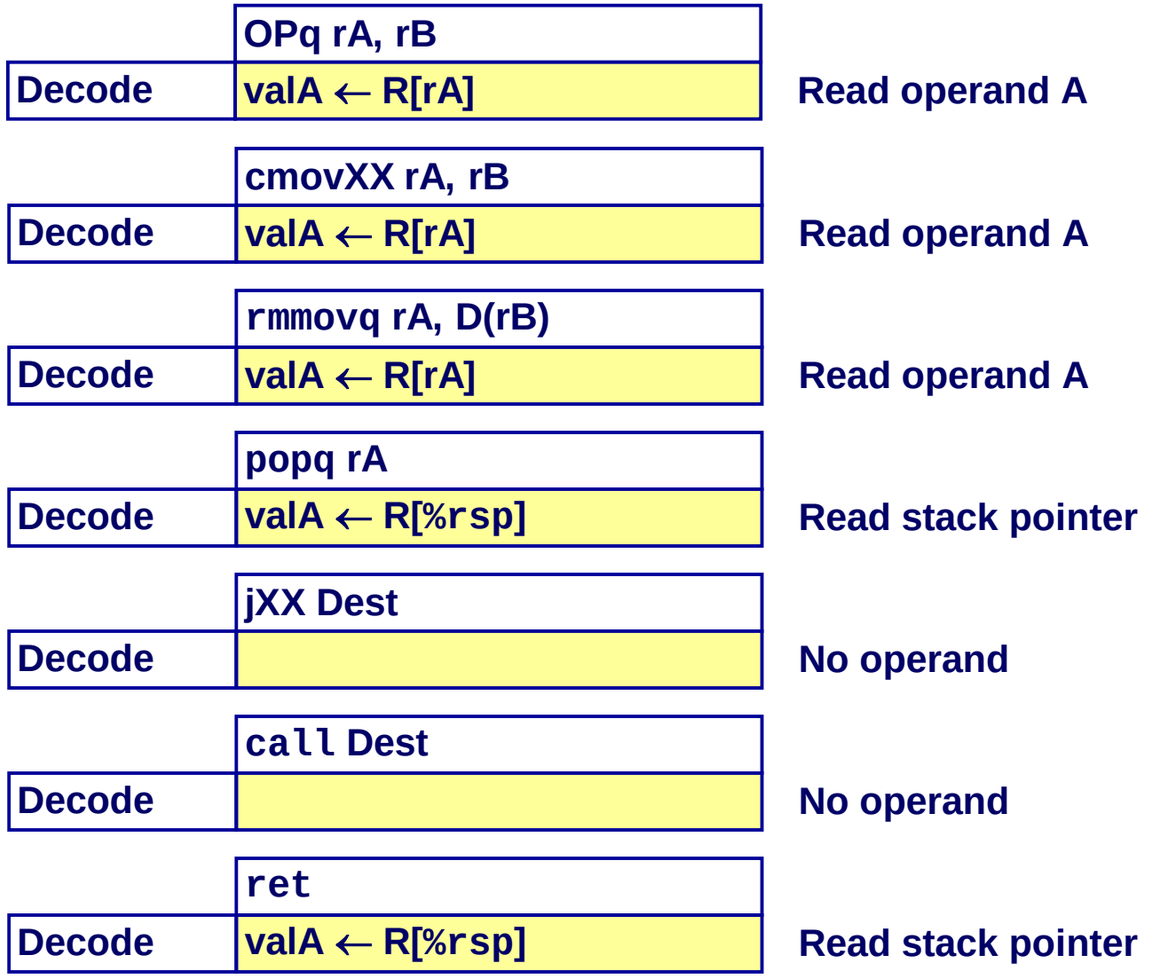
- srcA, srcB: read port addresses
- dstE, dstM: write port addresses

Signals

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

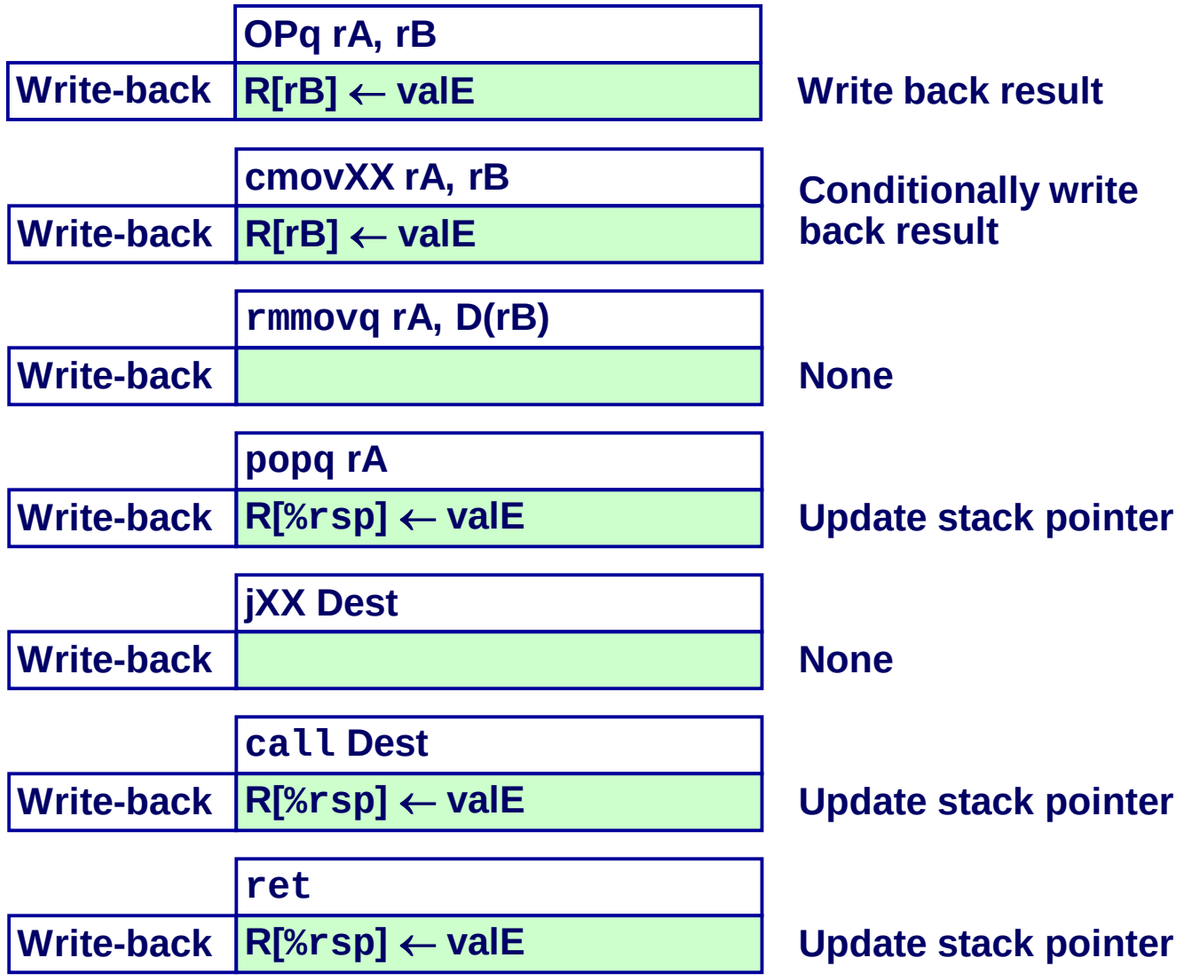


A Source




```
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
];
```

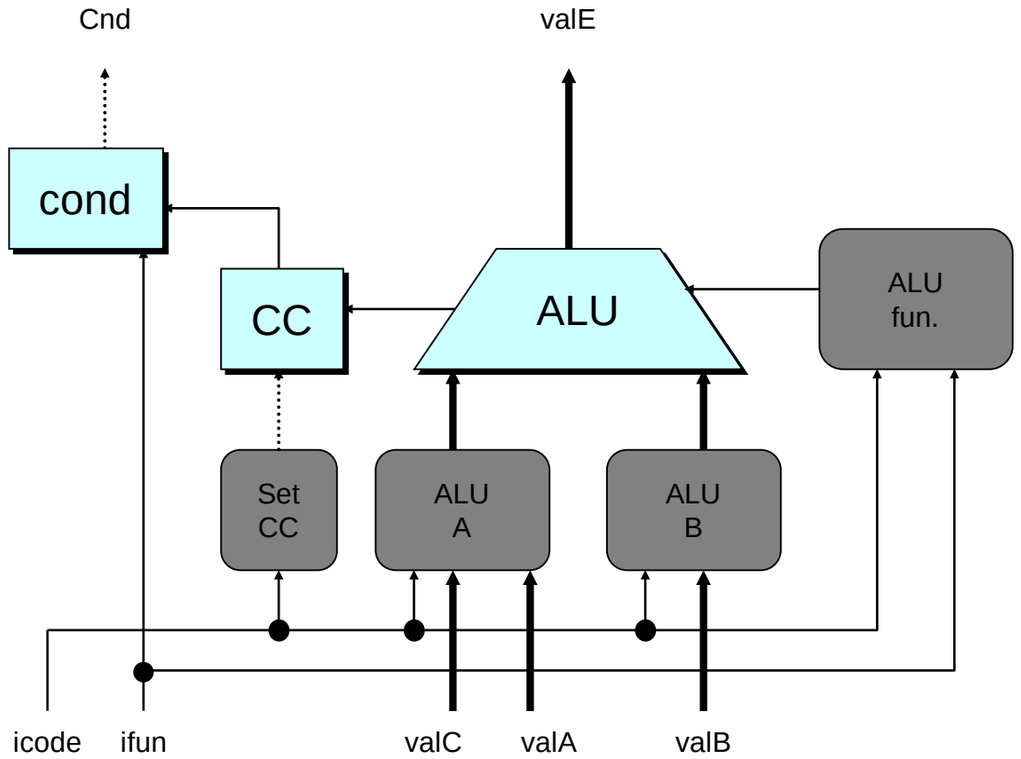
Execute Logic

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

	OPq rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	cmovXX rA, rB	
Execute	valE ← 0 + valA	Pass valA through ALU
	rmmovq 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 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;
];
```

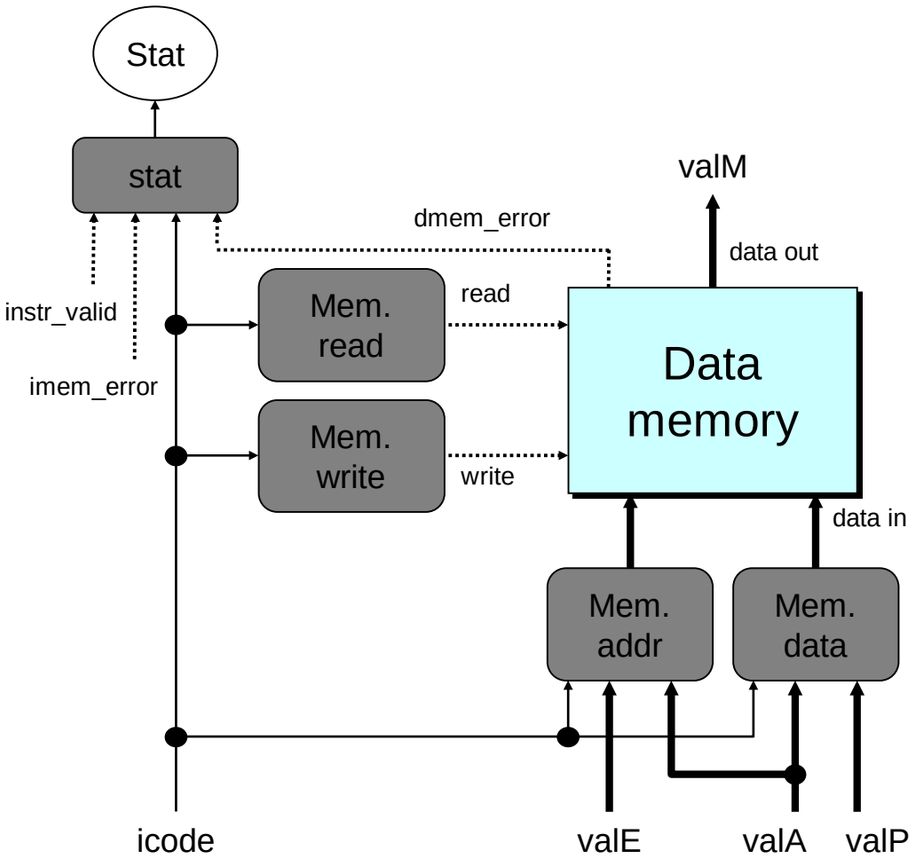
Memory Logic

Memory

- Reads or writes memory word

Control Logic

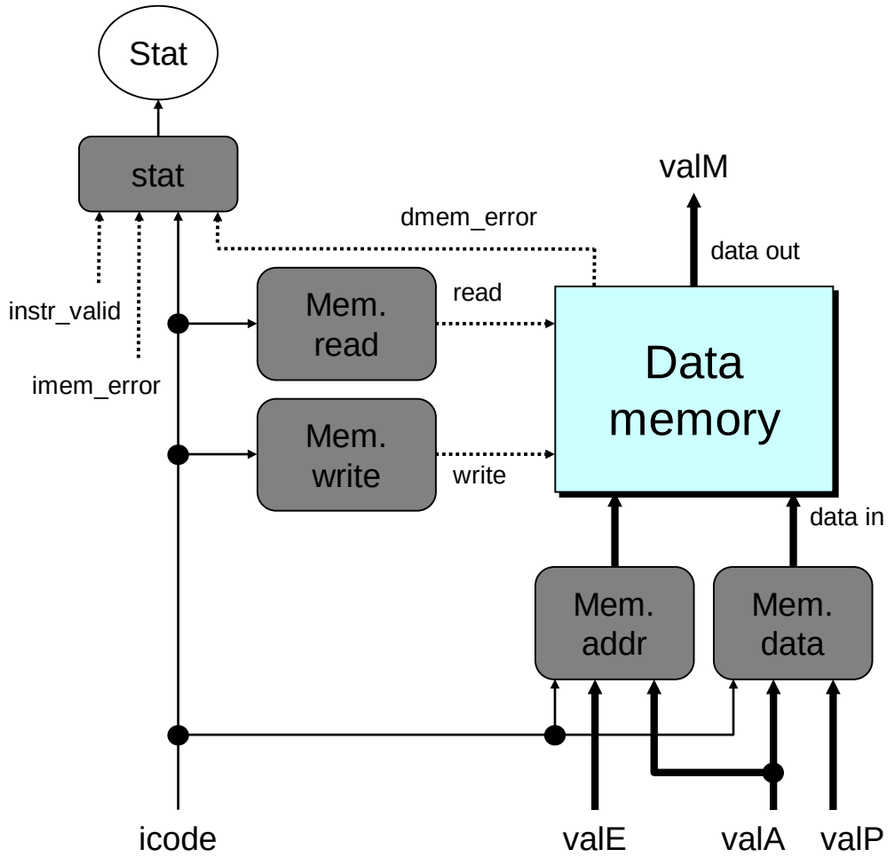
- stat: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



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

	OPq rA, rB	
Memory		No operation
	rmmovq rA, D(rB)	
Memory	M_8[valE] ← valA	Write value to memory
	popq rA	
Memory	valM ← M_8[valA]	Read from stack
	jXX Dest	
Memory		No operation
	call Dest	
Memory	M_8[valE] ← valP	Write return value on stack
	ret	
Memory	valM ← M_8[valA]	Read return address

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

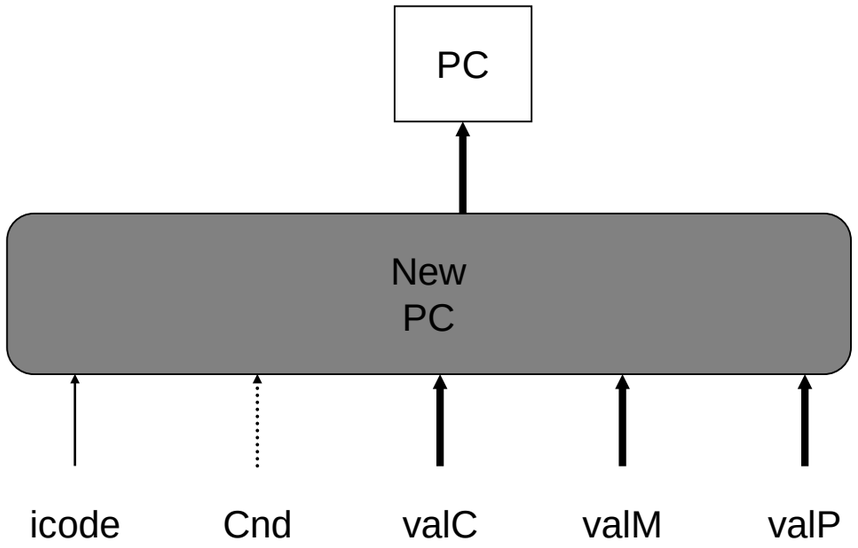
Memory Read

	OPq rA, rB	
Memory		No operation
	rmmovq rA, D(rB)	
Memory	M_8[valE] ← valA	Write value to memory
	popq rA	
Memory	valM ← M_8[valA]	Read from stack
	jXX Dest	
Memory		No operation
	call Dest	
Memory	M_8[valE] ← valP	Write return value on stack
	ret	
Memory	valM ← M_8[valA]	Read return address

```
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

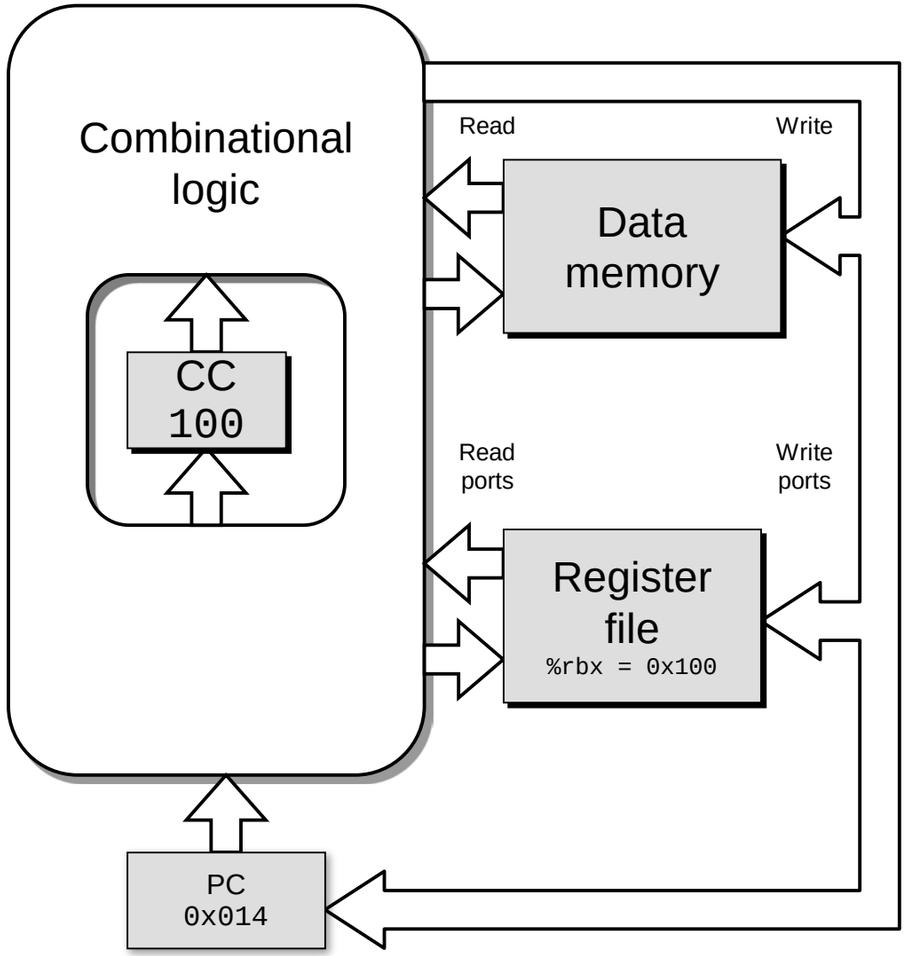
	call Dest	
PC update	PC ← valC	Set PC to destination

	ret	
PC update	PC ← valM	Set PC to return address

New PC

- Select next value of PC

SEQ Operation



State

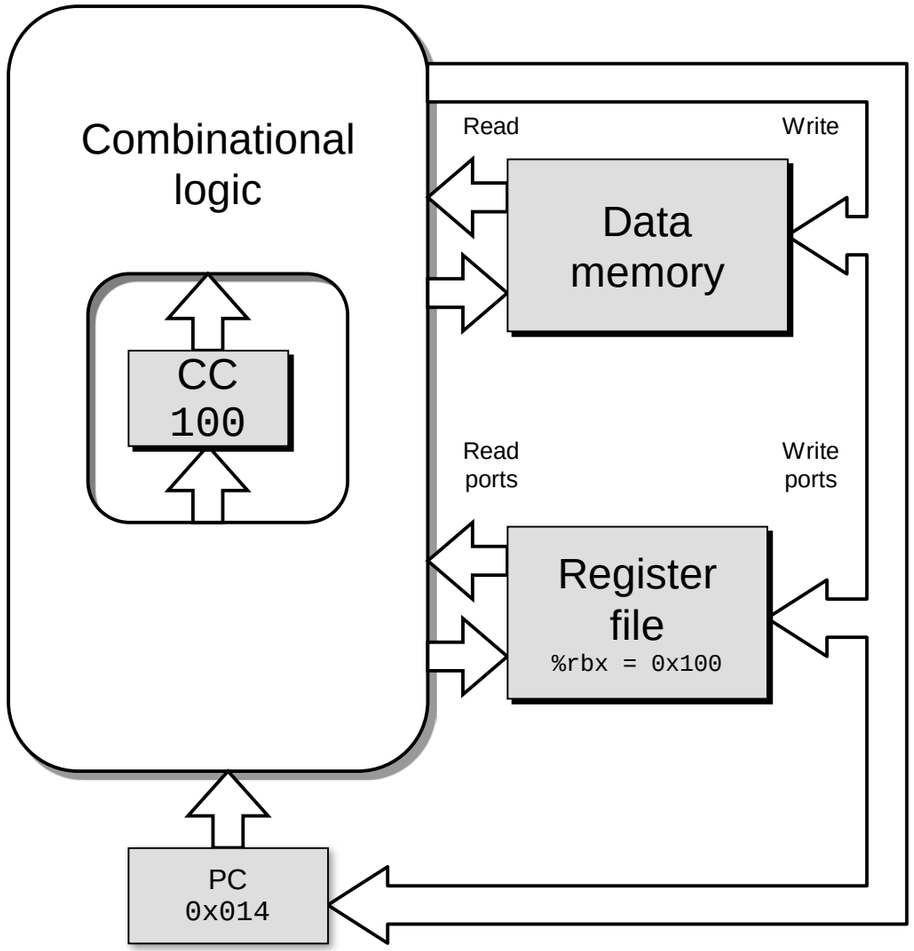
- PC register
- Cond. Code register
- Data memory
- Register file

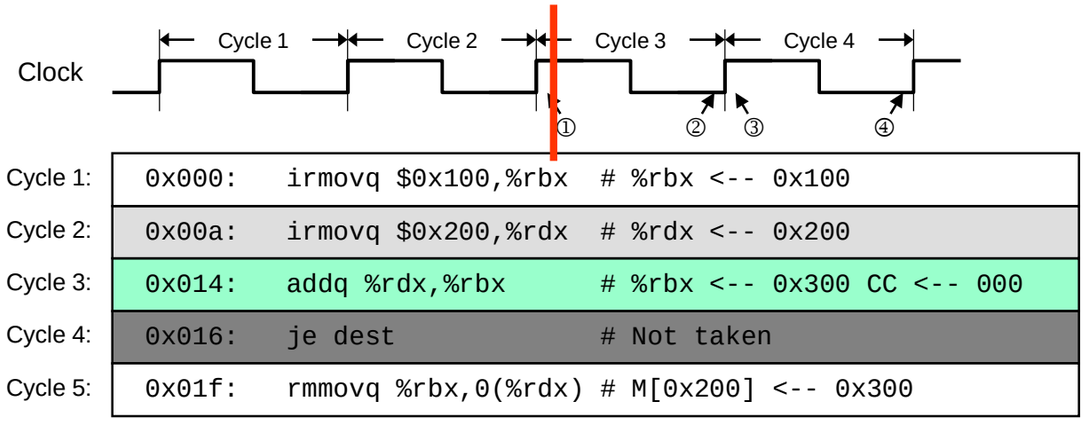
Updated as clock rises

Combinational Logic

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

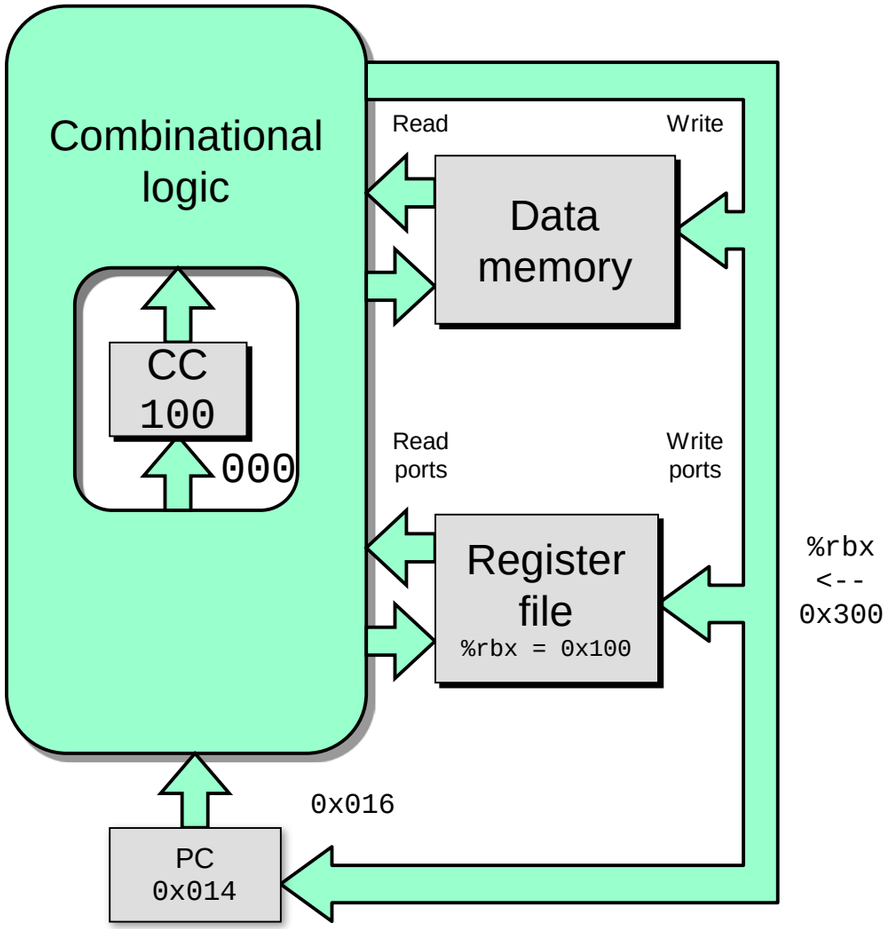
SEQ Operation #2

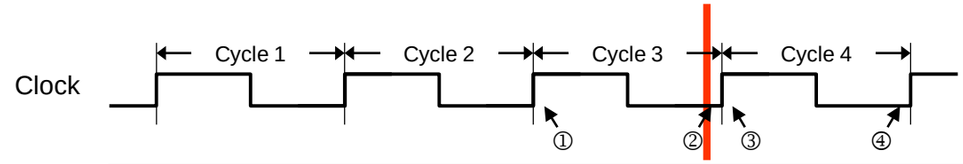




- **state set according to second irmovq instruction**
- **combinational logic starting to react to state changes**

SEQ Operation #3

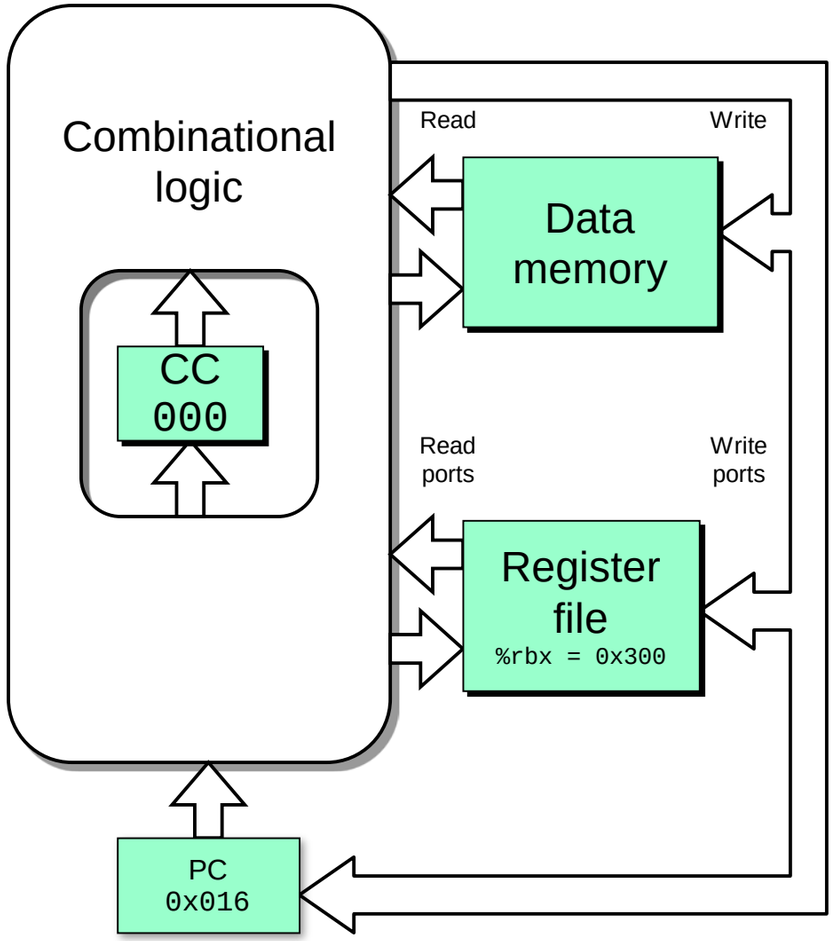


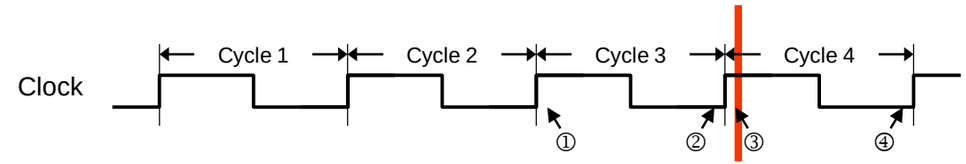


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

- **state set according to second irmovq instruction**
- **combinational logic generates results for addq instruction**

SEQ Operation #4

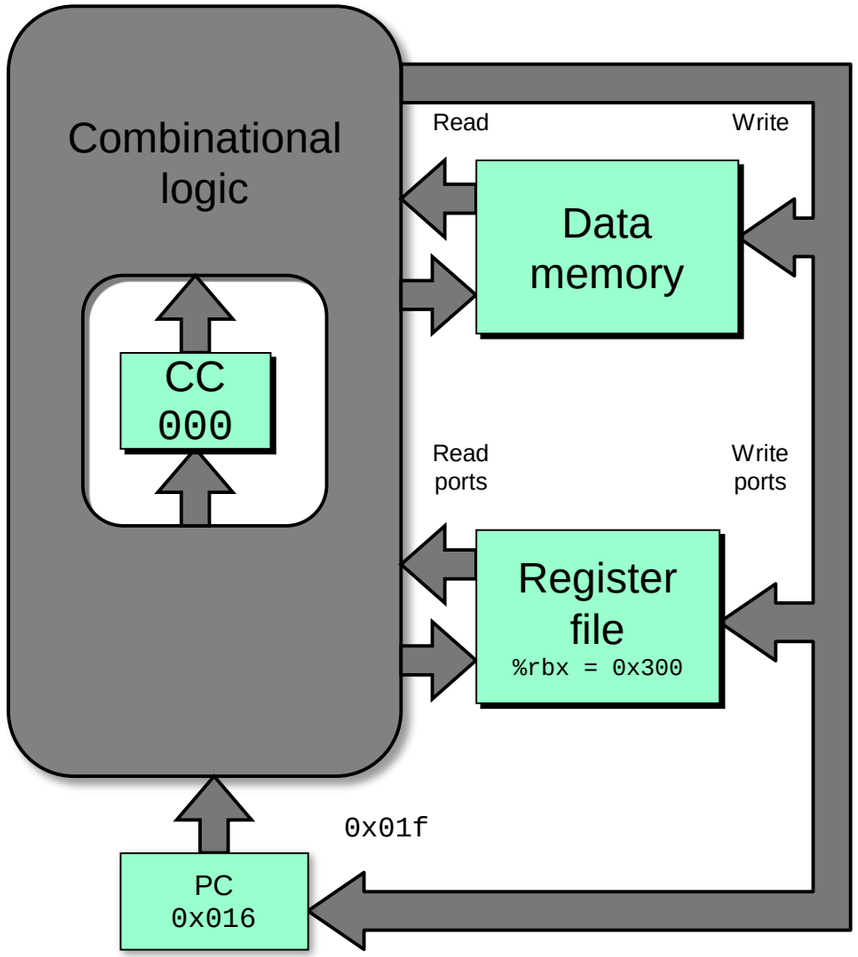


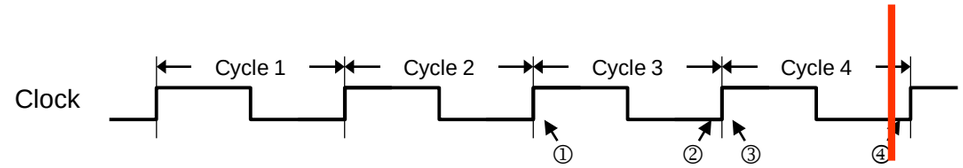


Cycle 1:	`0x000: irmovq $0x100,%rbx # %rbx <-- 0x100`
Cycle 2:	`0x00a: irmovq $0x200,%rdx # %rdx <-- 0x200`
Cycle 3:	`0x014: addq %rdx,%rbx # %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016: je dest # Not taken`
Cycle 5:	`0x01f: rmmovq %rbx,0(%rdx) # M[0x200] <-- 0x300`

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

SEQ Operation #5





Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

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

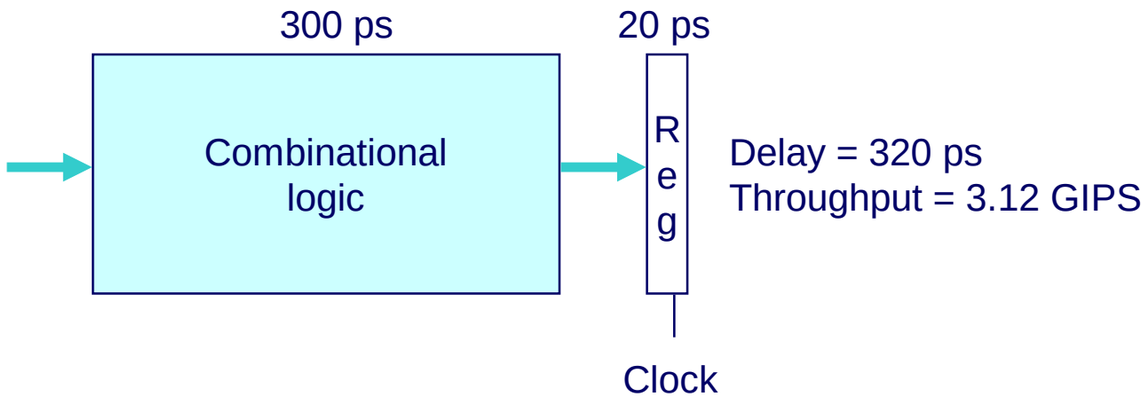
18-600 Foundations of Computer Systems

Lecture 8: "Processor Architecture and Design"

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

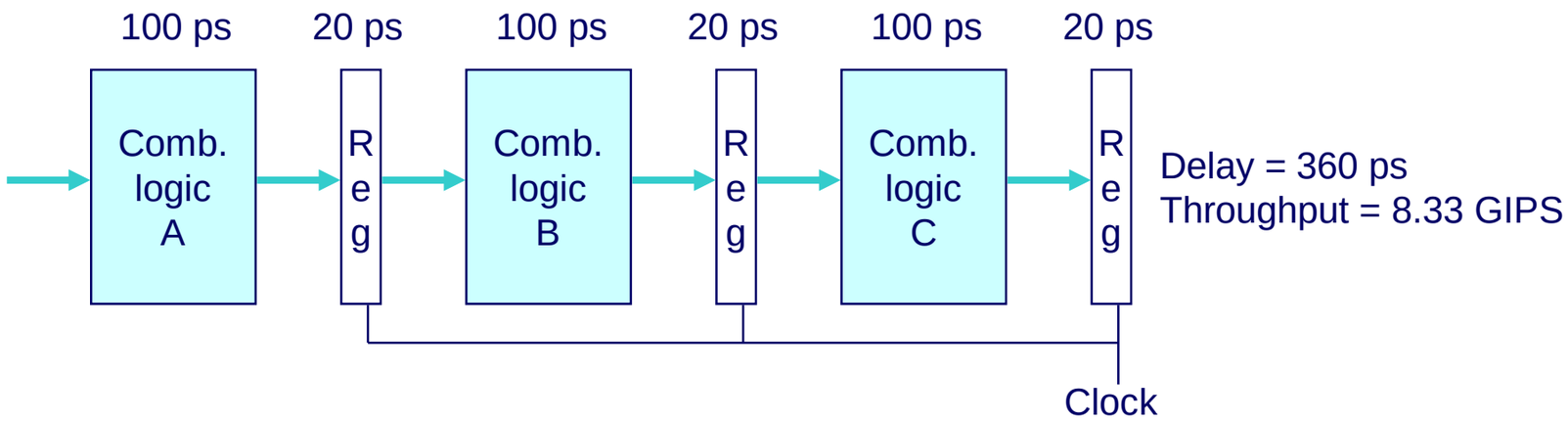
Electrical & Computer ENGINEERING

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

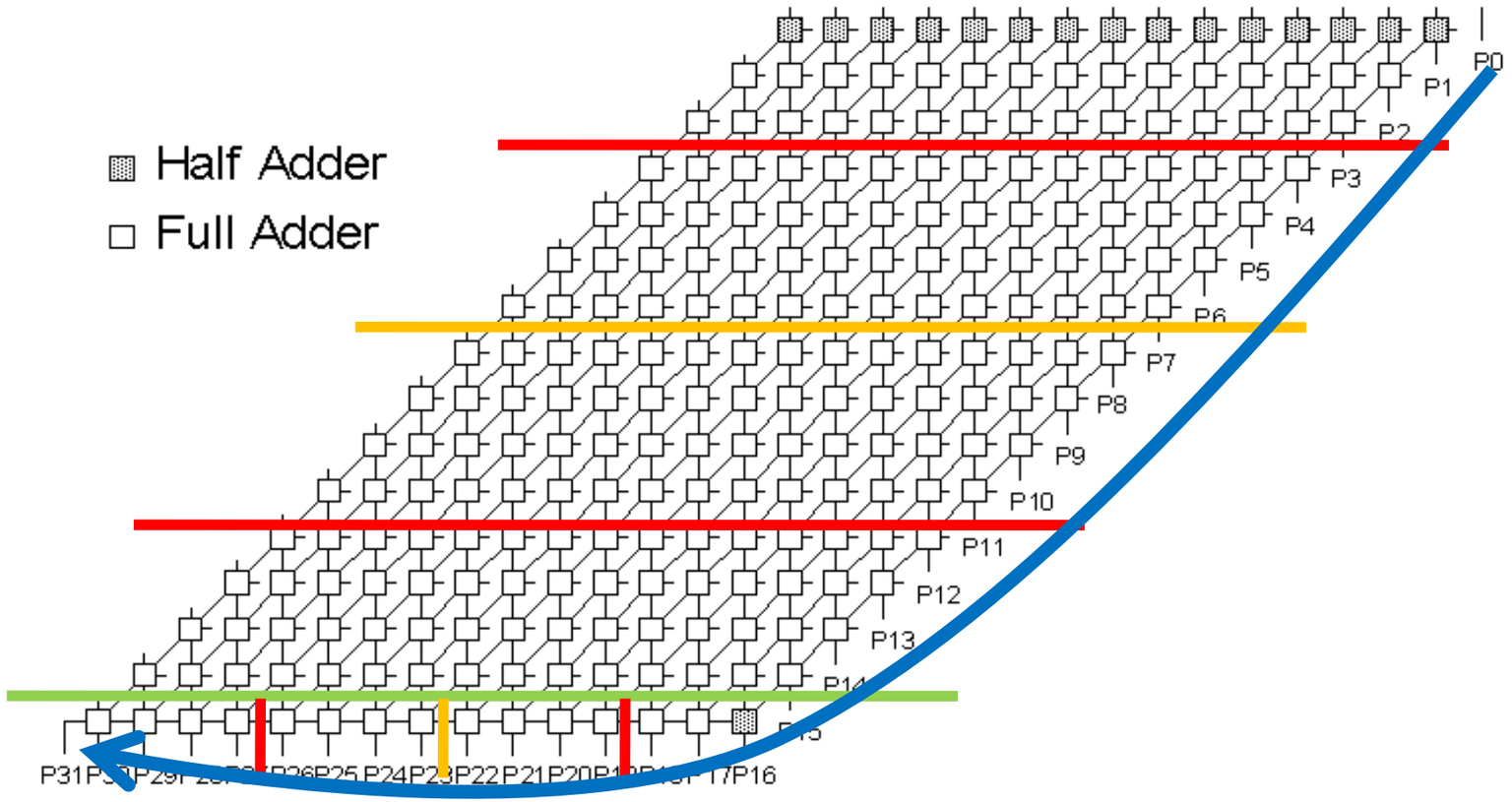
3-Way Pipelined Version



- 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

[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

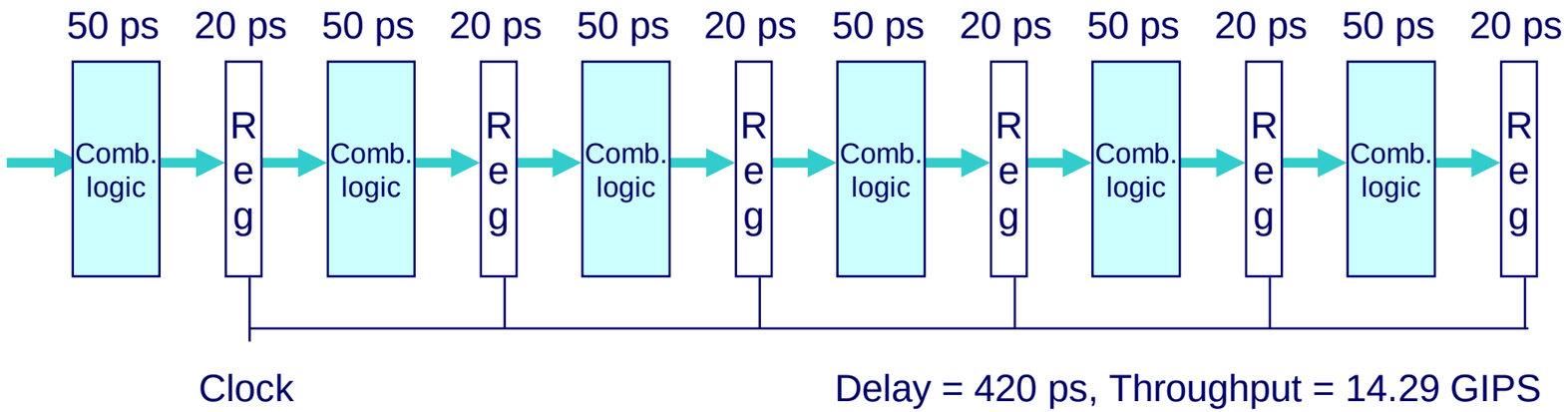
Pipelining Fundamentals

- Motivation:
  - Increase throughput with little increase in hardware.

Bandwidth or Throughput = Performance

- Bandwidth (BW) = no. of tasks/unit time
- For a system that operates on one task at a time:
  - BW = 1/delay (latency)
- BW can be increased by pipelining if many operands exist which need the same operation, i.e. many repetitions of the same task are to be performed.
- Latency required for each task remains the same or may even increase slightly.

Limitations: Register Overhead



- As we try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

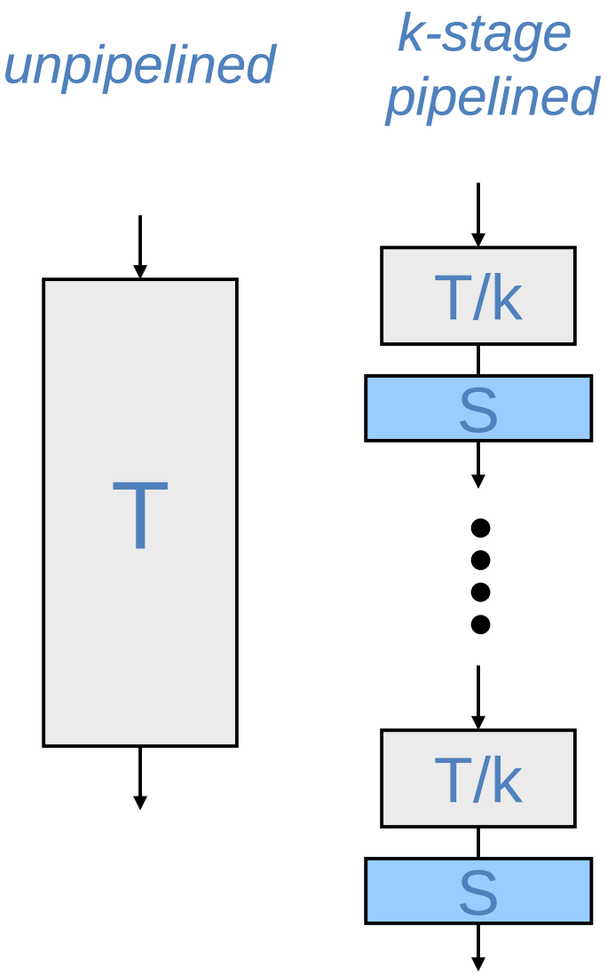
Pipelining Performance Model

- Starting from an un-pipelined version with propagation delay T and BW = 1/T

$P_{pipelined} = BW_{pipelined} = 1 / (T/ k + S)$

where

S = delay through latch and overhead



Hardware Cost Model

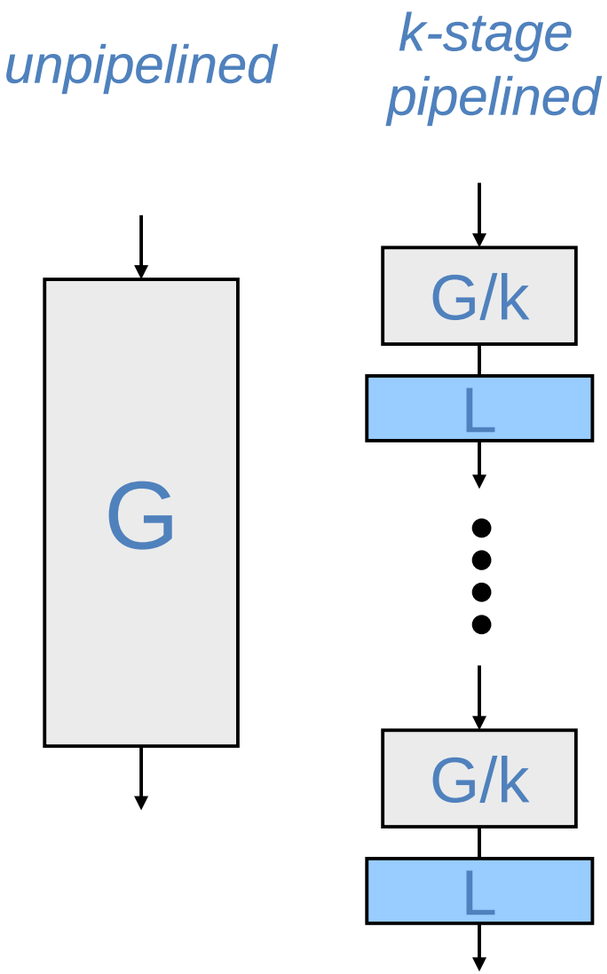
- Starting from an un-pipelined version with hardware cost G

$$Cost_{pipelined} = kL + G$$

where

L = cost of adding each latch, and

k = number of stages



[Peter M. Kogge, 1981]

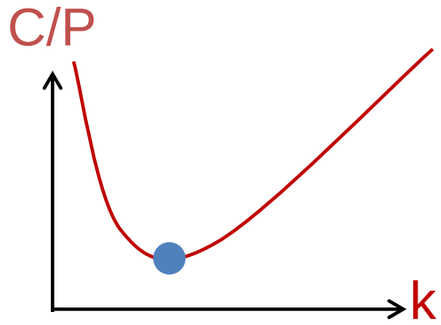
Cost/Performance Trade-off

Cost/Performance:

C/P = [Lk + G] / [1/(T/k + S)] = (Lk + G) (T/k + S)

= LT + GS + LSk + GT/k

Optimal Cost/Performance: find min. C/P w.r.t. choice of k

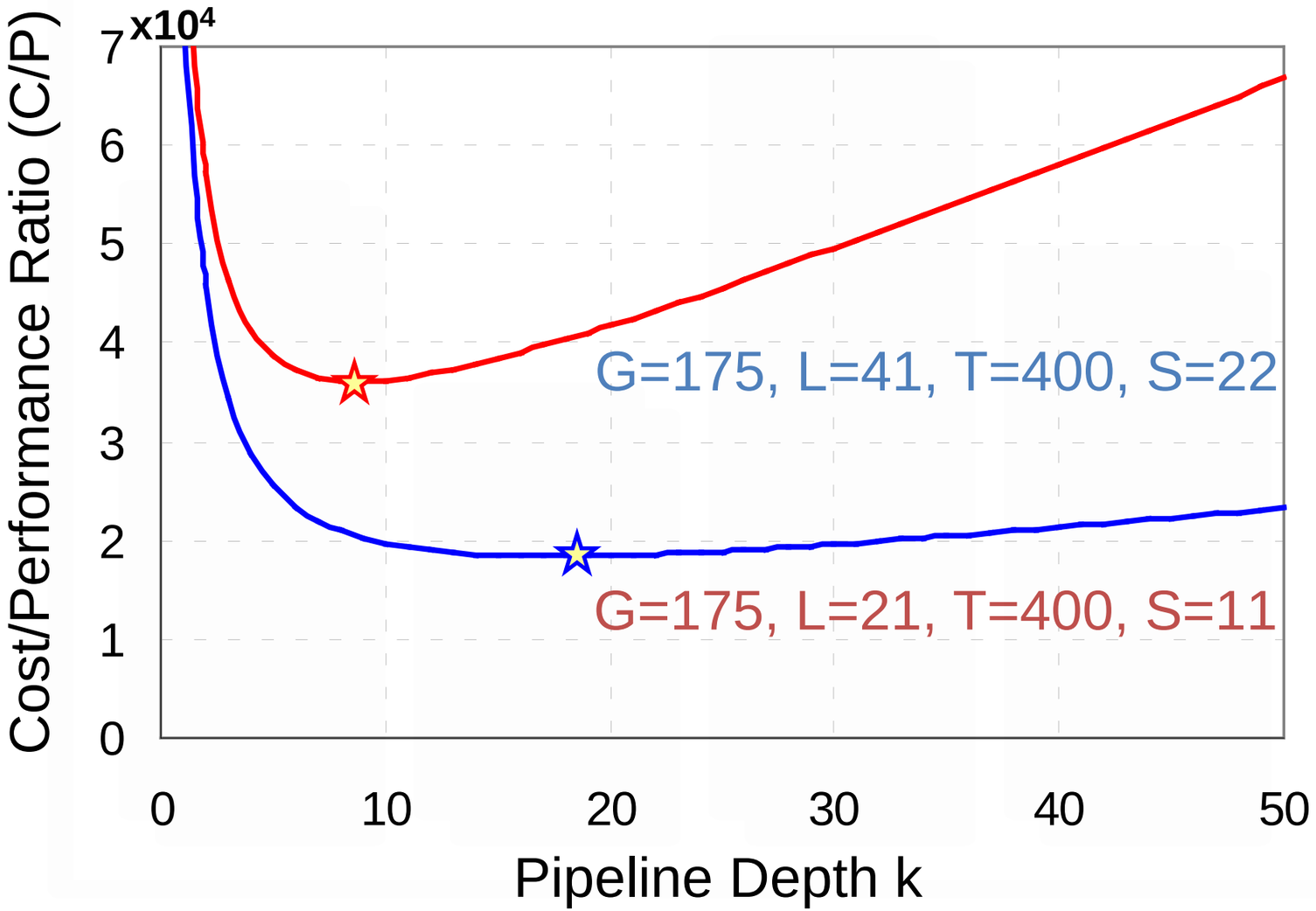


$$\frac{d}{dk}\left(\frac{Lk+G}{\frac{1}{\frac{T}{k}+S}}\right) = 0 + 0 + LS - \frac{GT}{k^2}$$

$$LS - \frac{GT}{k^2} = 0$$

$$k_{opt} = \sqrt{\frac{GT}{LS}}$$

"Optimal" Pipeline Depth (k_{opt}) Examples



Pipelining Idealistic Assumptions

- **Uniform Sub-computations**

  The computation to be pipelined can be evenly partitioned into uniform-latency sub-operations

- **Repetition of Identical Computations**

  The same computations are to be performed repeatedly on a large number of different inputs

- **Repetition of Independent Computations**

  All the repetitions of the same computation are mutually independent, i.e. no data dependency and no resource conflicts

Instruction Pipeline Design

- Uniform Sub-computations ... NOT!
  - ⇒ balancing pipeline stages
    - stage quantization to yield balanced pipe stages
    - minimize internal fragmentation (some waiting stages)
- Identical Computations ... NOT!
  - ⇒ unifying instruction types
    - coalescing instruction types into one multi-function pipe
    - minimize external fragmentation (some idling stages)
- Independent Computations ... NOT!
  - ⇒ resolving pipeline hazards
    - inter-instruction dependency detection and resolution
    - minimize performance loss due to pipeline stalls

18-600 Foundations of Computer Systems

Lecture 9: "Pipelined Processor Design"

John P. Shen & Zhiyi Yu

September 28, 2016

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