

CS330 - Computer Organization and Assembly Language Programming

Lecture 18

-ISA-

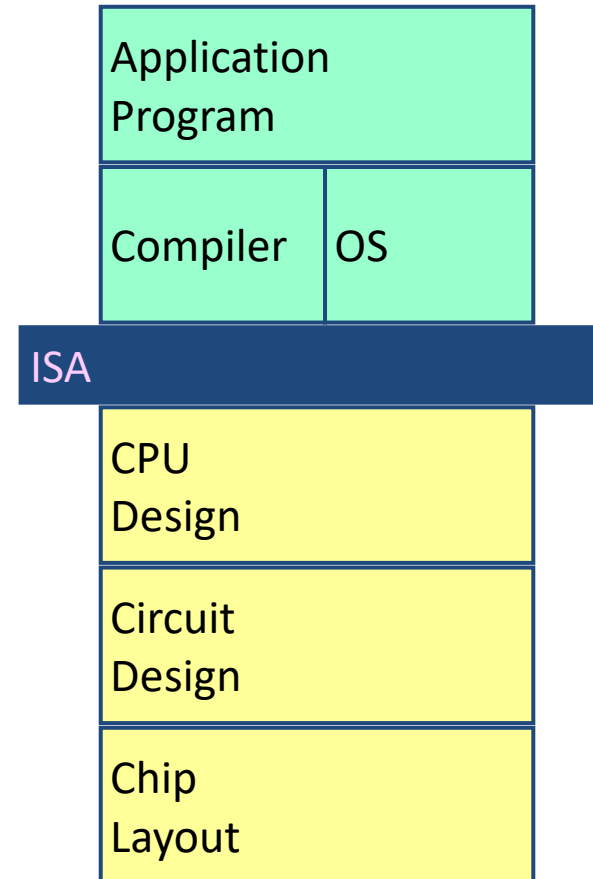
Professor : Mahmut Unan – UAB CS

Agenda

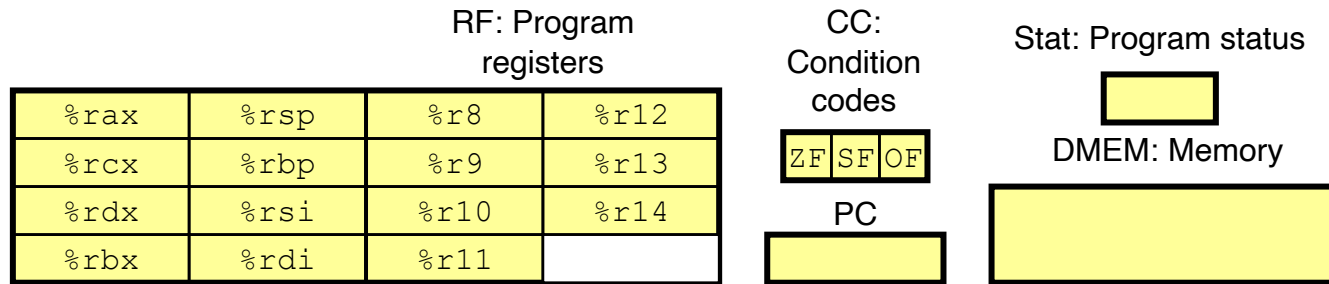
- The Y86-64 Instruction Set Architecture
 - Programmer Visible State
 - Y86-64 Instructions
 - Instruction Encoding
- Overview of Logic Design
- Computing with Logic Gates
- Bit Equality
- Arithmetic Logic Unit
- Hardware Control Language
- SEQ Hardware Structure
- SEQ Stages and Hardware

Instruction Set Architecture

- Assembly Language View
 - Processor state
 - Registers, memory, ...
 - Instructions
 - `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 built
 - Use variety of tricks 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 #1

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Y86-64 Instructions

- Format
 - **1–10 bytes** of information read from memory
 - Can determine instruction length from first byte
 - Not as many instruction types, and simpler encoding than with x86-64
 - Each accesses and modifies some part(s) of the program state
- Instruction encodings range between 1 and 10 bytes.
 - An instruction consists of a **1-byte instruction specifier, possibly a 1-byte register specifier, and possibly a 8-byte constant word.**

Y86-64 Instruction Set #2

Byte	0	1	2	3	4	5	6	7	8	9						
halt	0	0									rrmovq 7 0					
nop	1	0									cmovle 7 1					
cmovXX rA, rB	2	fn	rA	rB							cmovl 7 2					
irmovq V, rB	3	0	F	rB						V					cmove 7 3	
rrmovq rA, D(rB)	4	0	rA	rB						D					cmovne 7 4	
mrmovq D(rB), rA	5	0	rA	rB						D					cmovge 7 5	
OPq rA, rB	6	fn	rA	rB											cmovg 7 6	
jXX Dest	7	fn	Dest													
call Dest	8	0	Dest													
ret	9	0														
pushq rA	A	0	rA	F												
popq rA	B	0	rA	F												

Y86-64 Instruction Set #3

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

addq 6 0

subq 6 1

andq 6 2

xorq 6 3

Y86-64 Instruction Set #4

Byte	0	1	2	3	4	5	6	7		
									jmp	<div>7</div> <div>0</div>
halt	<div>0</div> <div>0</div>								jle	<div>7</div> <div>1</div>
nop	<div>1</div> <div>0</div>								j1	<div>7</div> <div>2</div>
cmovXX rA, rB	<div>2</div>	<div>fn</div>	<div>rA</div>	<div>rB</div>					je	<div>7</div> <div>3</div>
irmovq V, rB	<div>3</div>	<div>0</div>	<div>F</div>	<div>rB</div>	V				jne	<div>7</div> <div>4</div>
rmmovq rA, D(rB)	<div>4</div>	<div>0</div>	<div>rA</div>	<div>rB</div>	D				jge	<div>7</div> <div>5</div>
rmovq D(rB), rA	<div>5</div>	<div>0</div>	<div>rA</div>	<div>rB</div>	D				jg	<div>7</div> <div>6</div>
OPq rA, rB	<div>6</div>	<div>fn</div>	<div>rA</div>	<div>rB</div>						
jXX Dest	<div>7</div>	<div>fn</div>	Dest							
call Dest	<div>8</div>	<div>0</div>	Dest							
ret	<div>9</div> <div>0</div>									
pushq rA	<div>A</div>	<div>0</div>	<div>rA</div>	<div>F</div>						
popq rA	<div>B</div>	<div>0</div>	<div>rA</div>	<div>F</div>						

Instruction Encoding

- Each instruction requires between 1 and 10 bytes, depending on which fields are required.
- Every instruction has an initial byte identifying the instruction type. This byte is split into two 4-bit parts: the high-order, or code, part, and the low-order, or function, part.

Encoding Registers

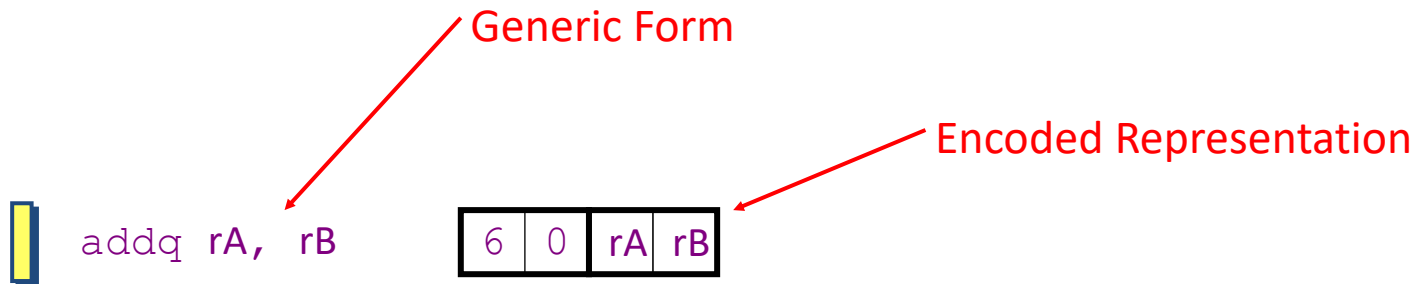
- Each register has 4-bit ID

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	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 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

Instruction Code

Add

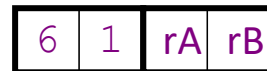
`addq rA, rB`

Function Code



Subtract (rA from rB)

`subq rA, rB`



And

`andq rA, rB`



Exclusive-Or

`xorq rA, rB`



- Refer to generically as “OPq”
- Encodings differ only by “function code”
 - Low-order 4 bytes in first instruction word
- Set condition codes as side effect

Move Operations

Register → Register

`rrmovq rA, rB`

2	0
---	---

Immediate → Register

`irmovq V, rB`

3	0	F	rB	V
---	---	---	----	---

Register → Memory

`rmmovq rA, D(rB)`

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

Memory → Register

`mrmmovq D(rB), rA`

5	0	rA	rB	D
---	---	----	----	---

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

Move Instruction Examples

X86-64

```
movq $0xabcd, %rdx
```

Encoding:

```
movq %rsp, %rbx
```

Encoding:

```
movq -12(%rbp), %rcx
```

Encoding:

```
movq %rsi, 0x41c(%rsp)
```

Encoding:

Y86-64

```
irmovq $0xabcd, %rdx
```

30 82 cd ab 00 00 00 00 00 00

```
rrmovq %rsp, %rbx
```

20 43

```
mrmovq -12(%rbp), %rcx
```

50 15 f4 ff ff ff ff ff ff ff

```
rmmovq %rsi, 0x41c(%rsp)
```

40 64 1c 04 00 00 00 00 00 00

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



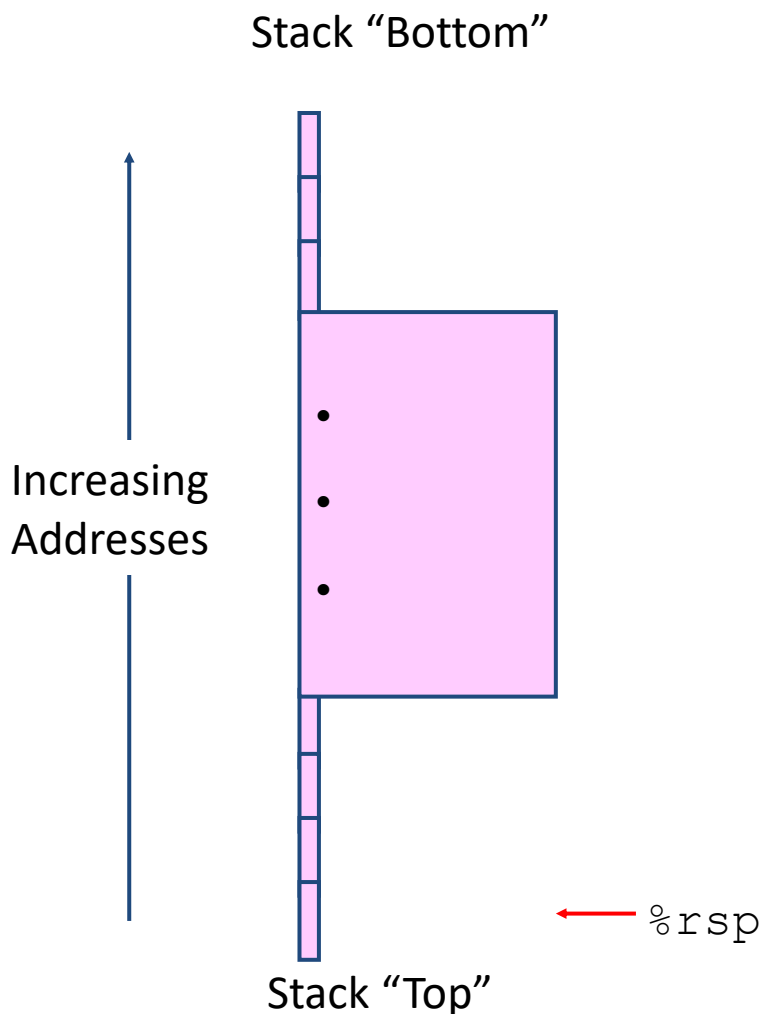
Jump When Greater or Equal



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
 - Top 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

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

popq rA



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

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 `rmovq` and `rmmovq`
- No Condition codes
 - Test instructions return 0/1 in register

CISC	Early RISC
A large number of instructions. The Intel document describing the complete set of instructions [28, 29] is over 1200 pages long.	Many fewer instructions. Typically less than 100.
Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.
Variable-length encodings. IA32 instructions can range from 1 to 15 bytes.	Fixed-length encodings. Typically all instructions are encoded as 4 bytes.
Multiple formats for specifying operands. In IA32, a memory operand specifier can have many different combinations of displacement, base and index registers, and scale factors.	Simple addressing formats. Typically just base and displacement addressing.

Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logical operations only use register operands. Memory referencing is only allowed by <i>load</i> instructions, reading from memory into a register, and <i>store</i> instructions, writing from a register to memory. This convention is referred to as a <i>load/store architecture</i> .
Implementation artifacts hidden from machine-level programs. The ISA provides a clean abstraction between programs and how they get executed.	Implementation artifacts exposed to machine-level programs. Some RISC machines prohibit particular instruction sequences and have jumps that do not take effect until the following instruction is executed. The compiler is given the task of optimizing performance within these constraints.
Condition codes. Special flags are set as a side effect of instructions and then used for conditional branch testing.	No condition codes. Instead, explicit test instructions store the test results in normal registers for use in conditional evaluation.
Stack-intensive procedure linkage. The stack is used for procedure arguments and return addresses.	Register-intensive procedure linkage. Registers are used for procedure arguments and return addresses. Some procedures can thereby avoid any memory references. Typically, the processor has many more (up to 32) registers.

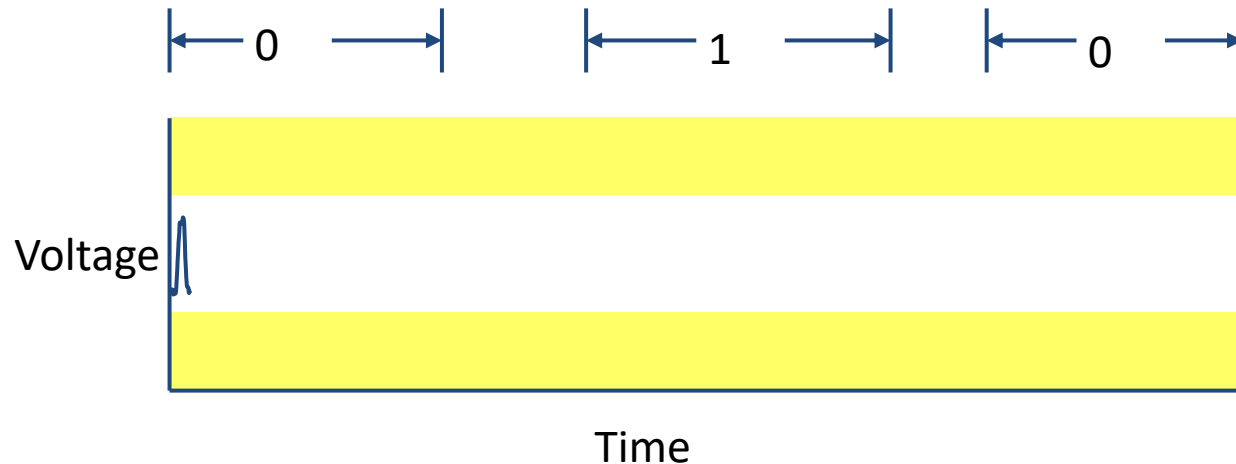
CISC vs. RISC

- Original Debate
 - Strong opinions!
 - CISC proponents---easy for compiler, fewer code bytes
 - RISC proponents---better for optimizing compilers, can make 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 cell phones use ARM processor

Overview of Logic Design

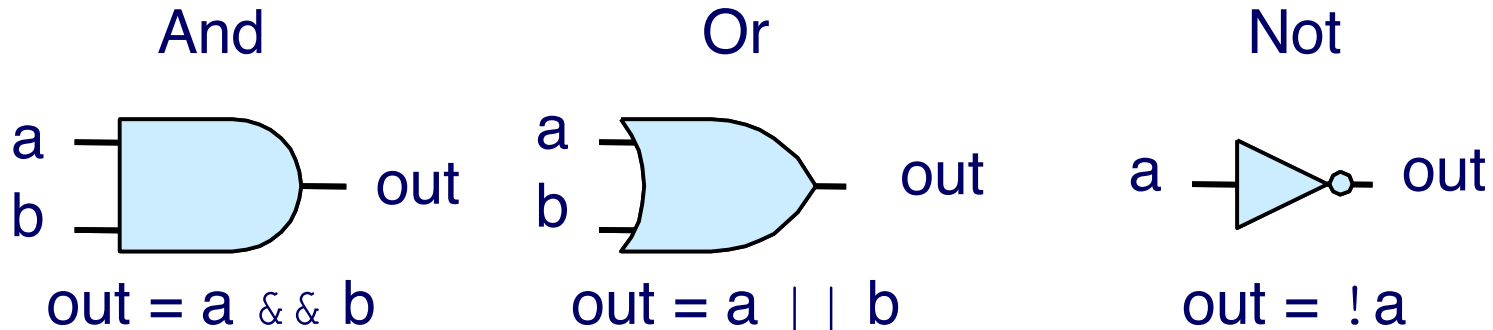
- Fundamental Hardware Requirements
 - Communication
 - How to get values from one place to another
 - Computation
 - Storage
- Bits are Our Friends
 - Everything expressed in terms of values 0 and 1
 - Communication
 - Low or high voltage on wire
 - Computation
 - Compute Boolean functions
 - Storage
 - Store bits of information

Digital Signals

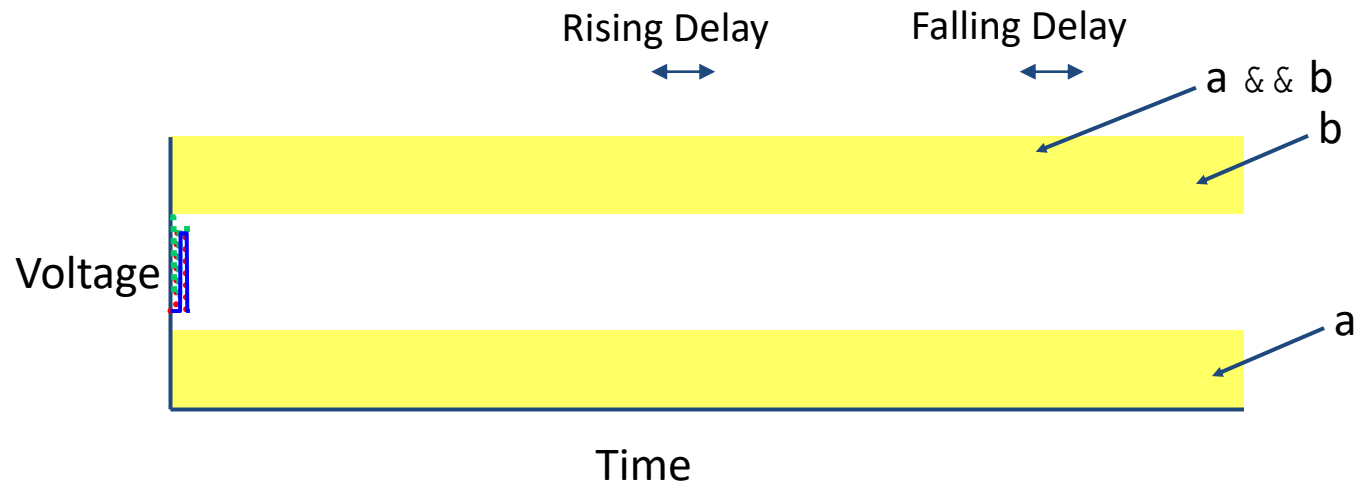


- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
 - Either high range (1) or low range (0)
 - With guard range between them
- Not strongly affected by noise or low quality circuit elements
 - Can make circuits simple, small, and fast

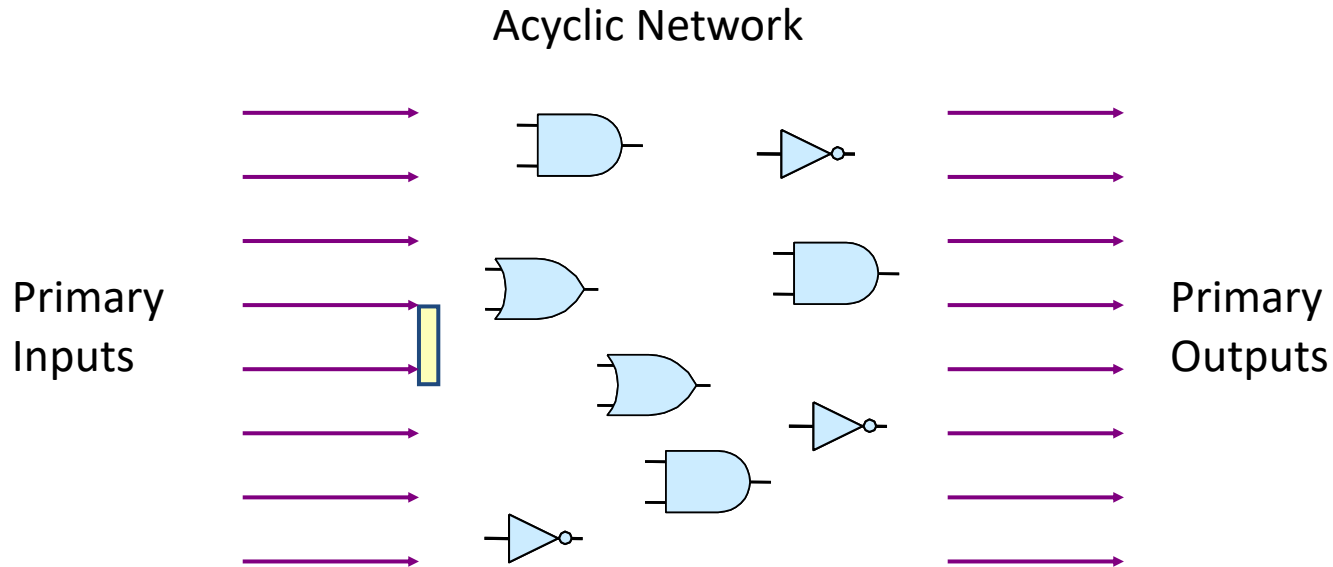
Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
 - With some, small delay

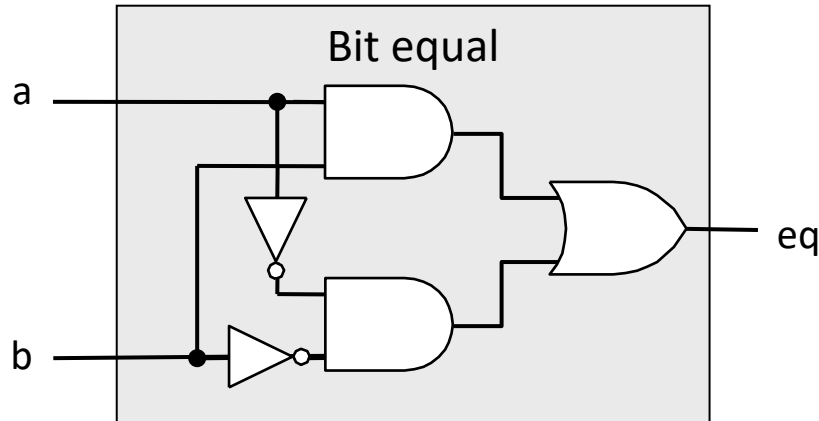


Combinational Circuits



- Acyclic Network of Logic Gates
 - Continuously responds to changes on primary inputs
 - Primary outputs become (after some delay) Boolean functions of primary inputs

Bit Equality

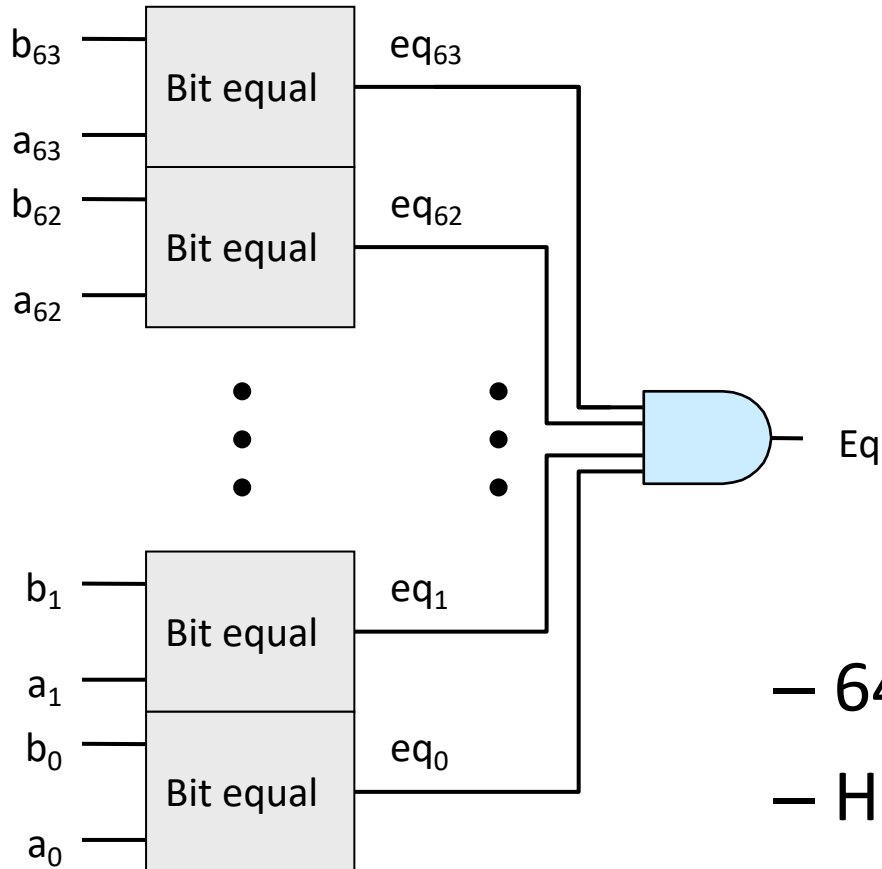


HCL Expression

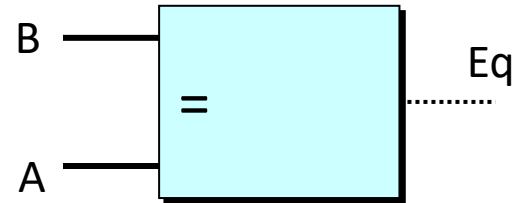
```
bool eq = (a&&b) || (!a&&!b)
```

- Generate 1 if a and b are equal
- Hardware Control Language (HCL)
 - Very simple hardware description language
 - Boolean operations have syntax similar to C logical operations
 - We'll use it to describe control logic for processors

Word Equality



Word-Level Representation

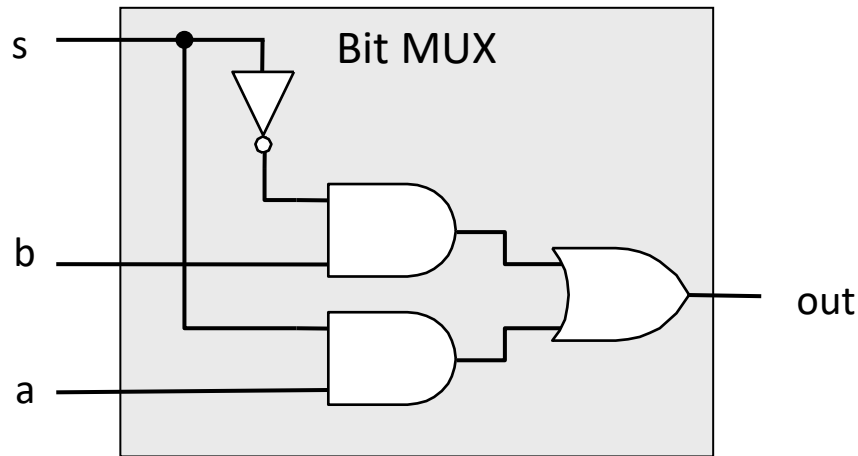


HCL Representation

```
bool Eq = (A == B)
```

- 64-bit word size
- HCL representation
 - Equality operation
 - Generates Boolean value

Bit-Level Multiplexor



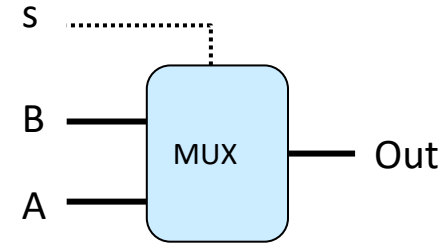
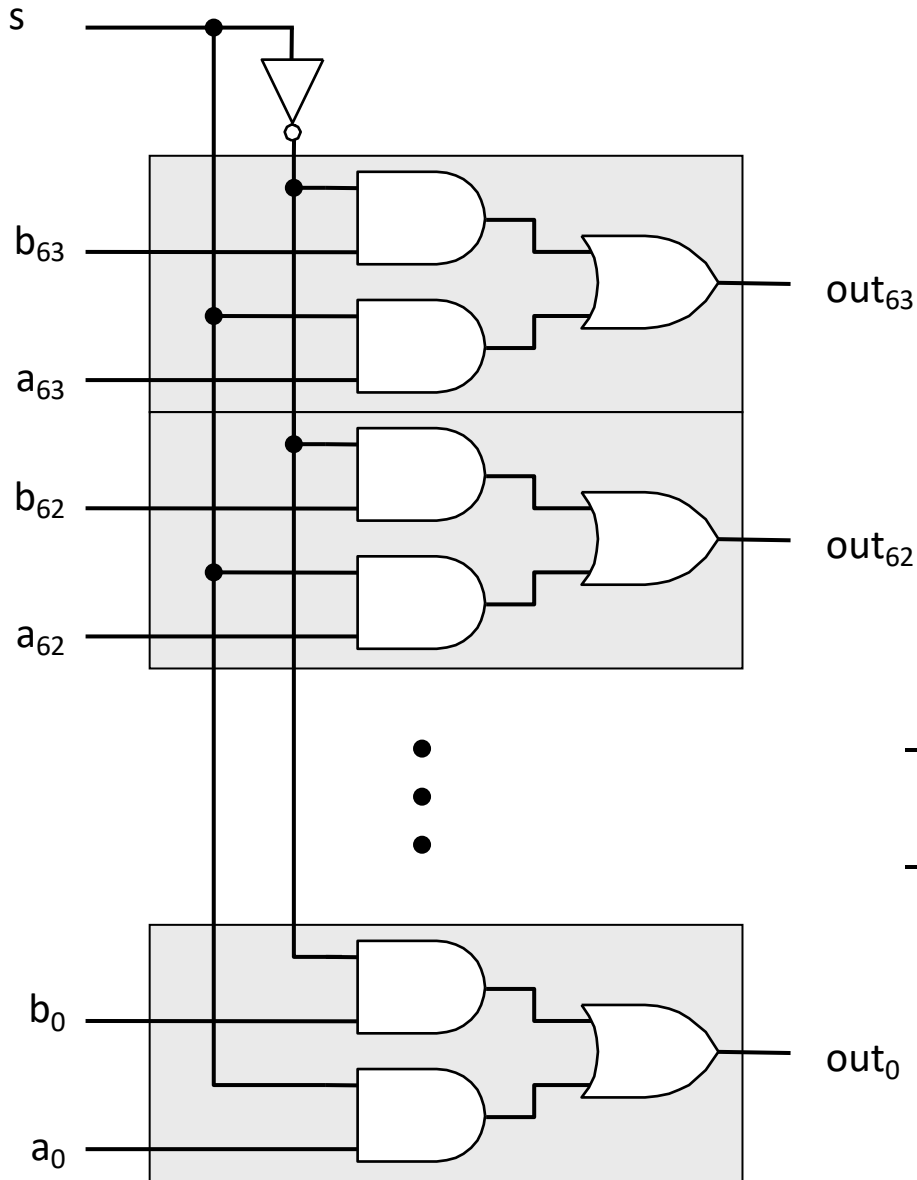
HCL Expression

```
bool out = (s&&a) || (!s&&b)
```

- Control signal *s*
- Data signals *a* and *b*
- Output *a* when *s*=1, *b* when *s*=0

Word Multiplexor

Word-Level Representation



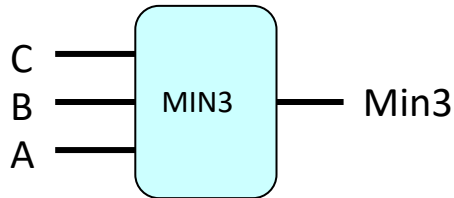
HCL Representation

```
int Out = [  
    s : A;  
    1 : B;  
];
```

- Select input word A or B depending on control signal s
- HCL representation
 - Case expression
 - Series of test : value pairs
 - Output value for first successful test

HCL Word-Level Examples

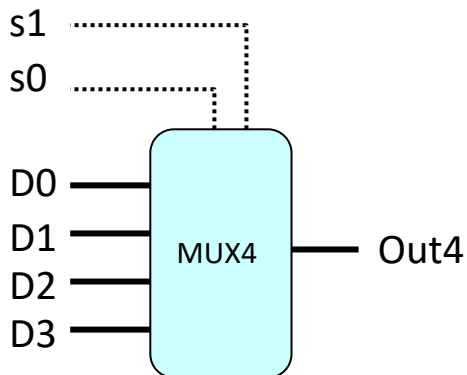
Minimum of 3 Words



```
int Min3 = [  
    A < B && A < C : A;  
    B < A && B < C : B;  
    1                : C;  
];
```

- Find minimum of three input words
- HCL case expression
- Final case guarantees match

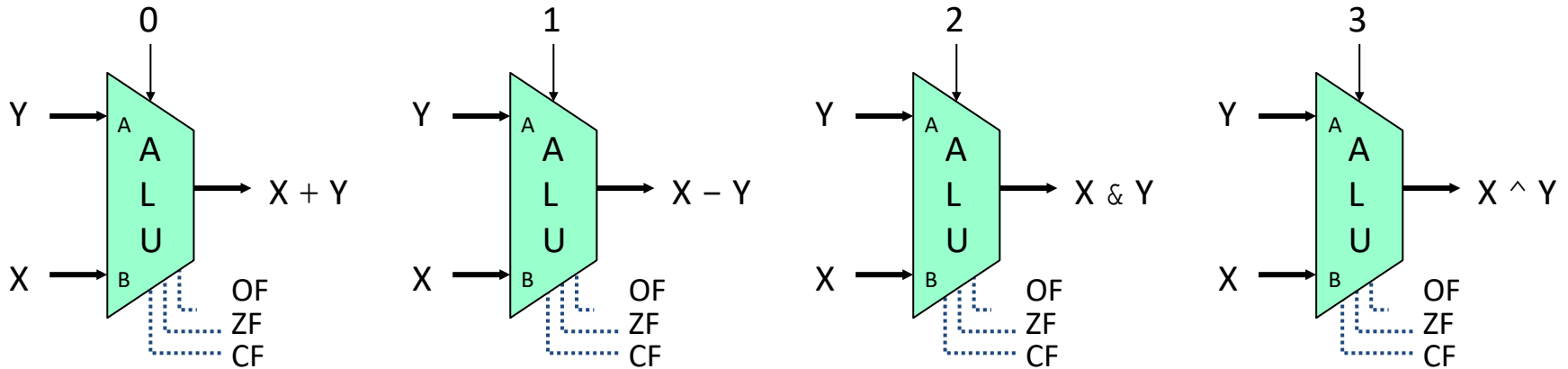
4-Way Multiplexor



```
int Out4 = [  
    !s1 && !s0 : D0;  
    !s1        : D1;  
    !s0        : D2;  
    1          : D3;  
];
```

- Select one of 4 inputs based on two control bits
- HCL case expression
- Simplify tests by assuming sequential matching

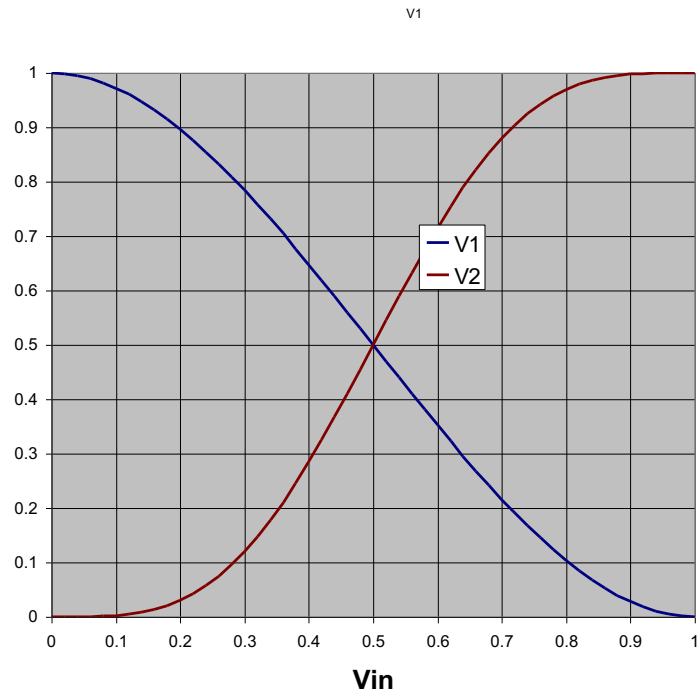
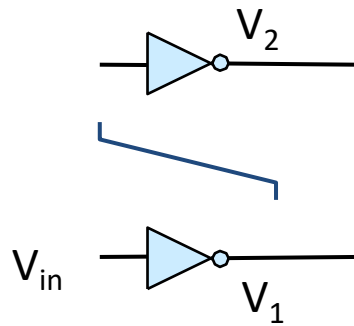
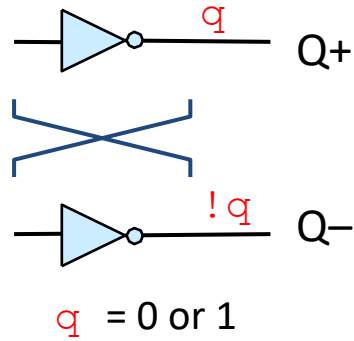
Arithmetic Logic Unit



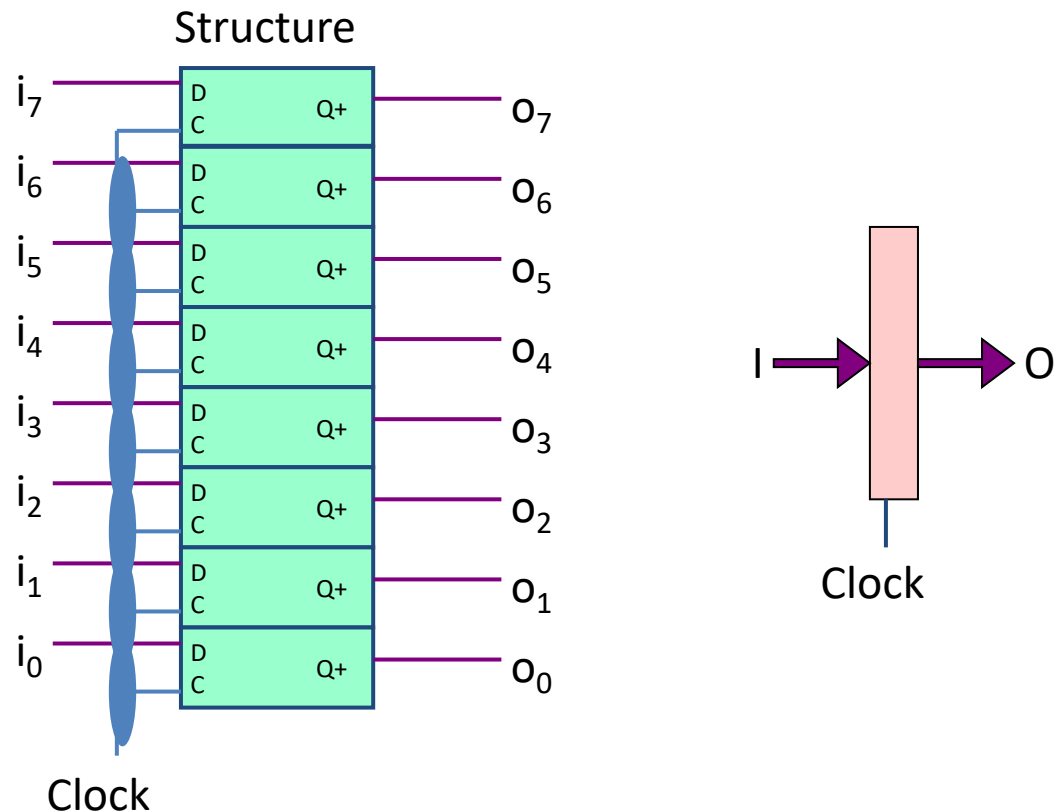
- Combinational logic
 - Continuously responding to inputs
- Control signal selects function computed
 - Corresponding to 4 arithmetic/logical operations in Y86-64
- Also computes values for condition codes

Storing 1 Bit

Bistable Element

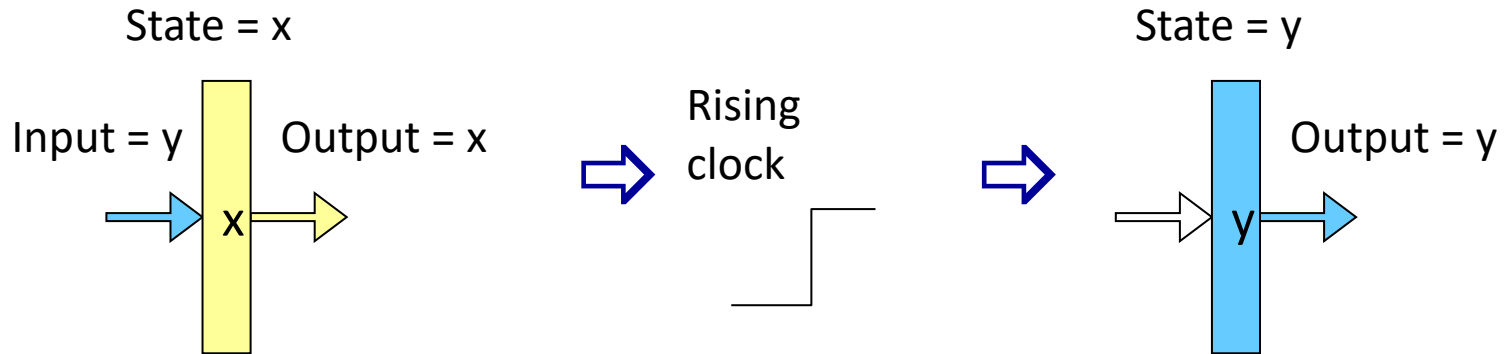


Registers



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

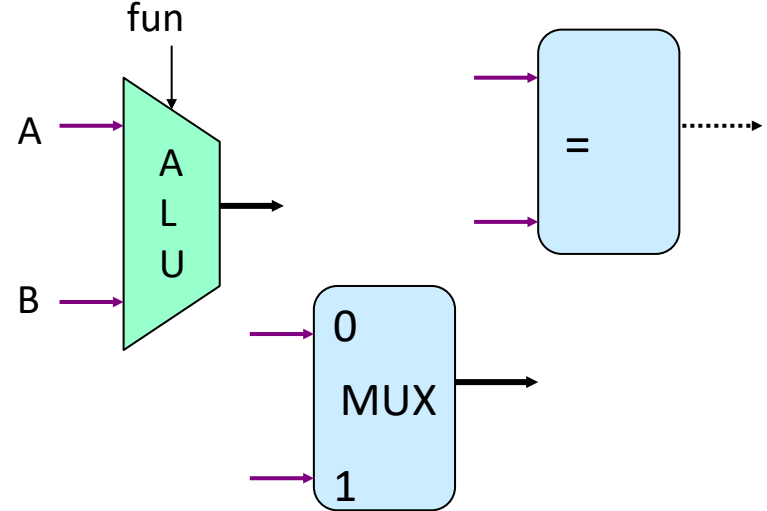
Register Operation



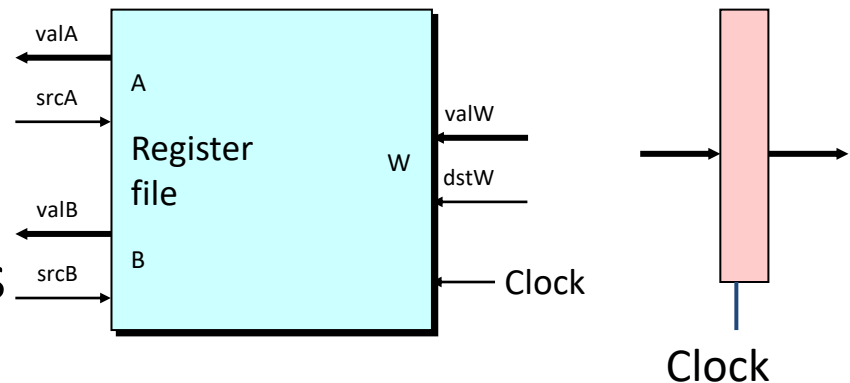
- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

Building Blocks

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

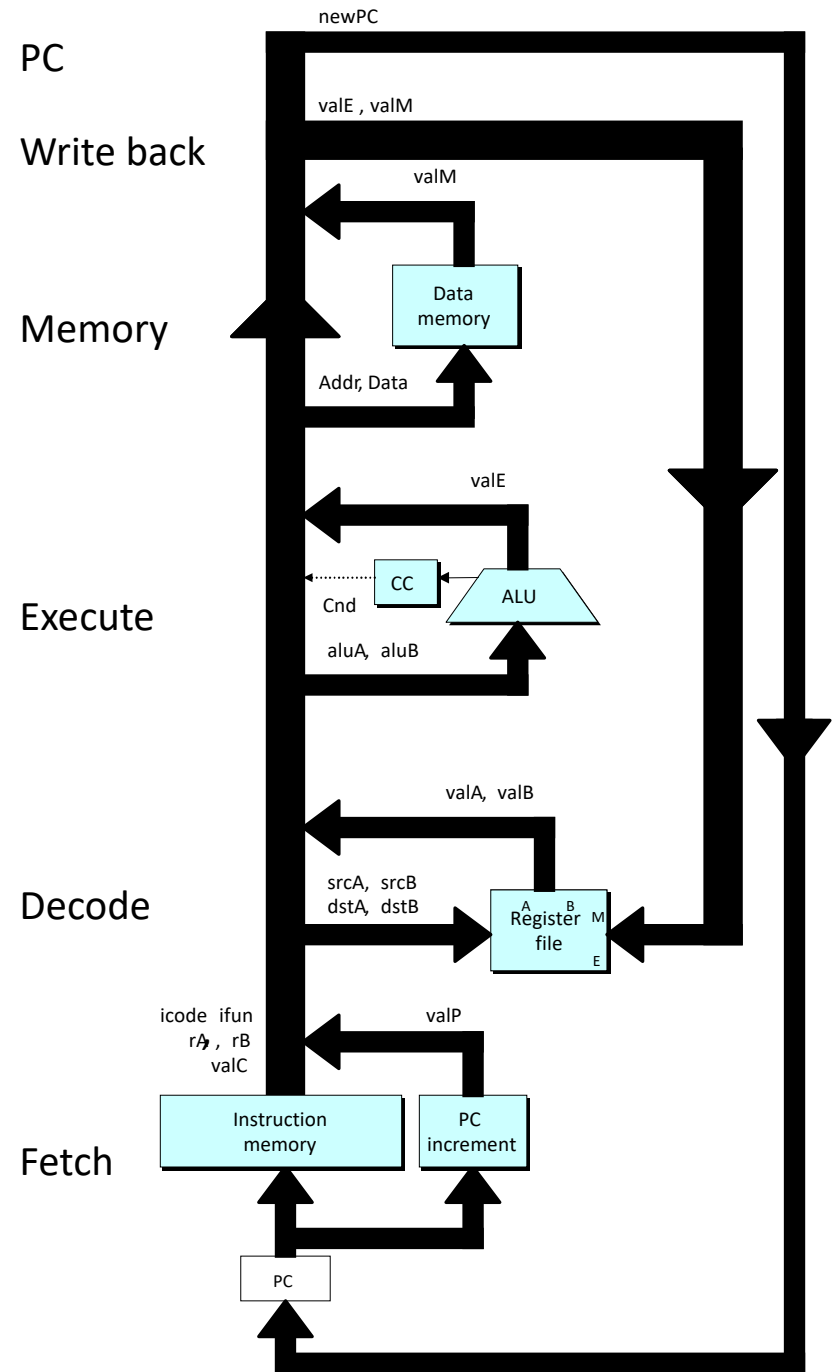


- Storage Elements
 - Store bits
 - Addressable memories
 - Non-addressable registers
 - Loaded only as clock rises



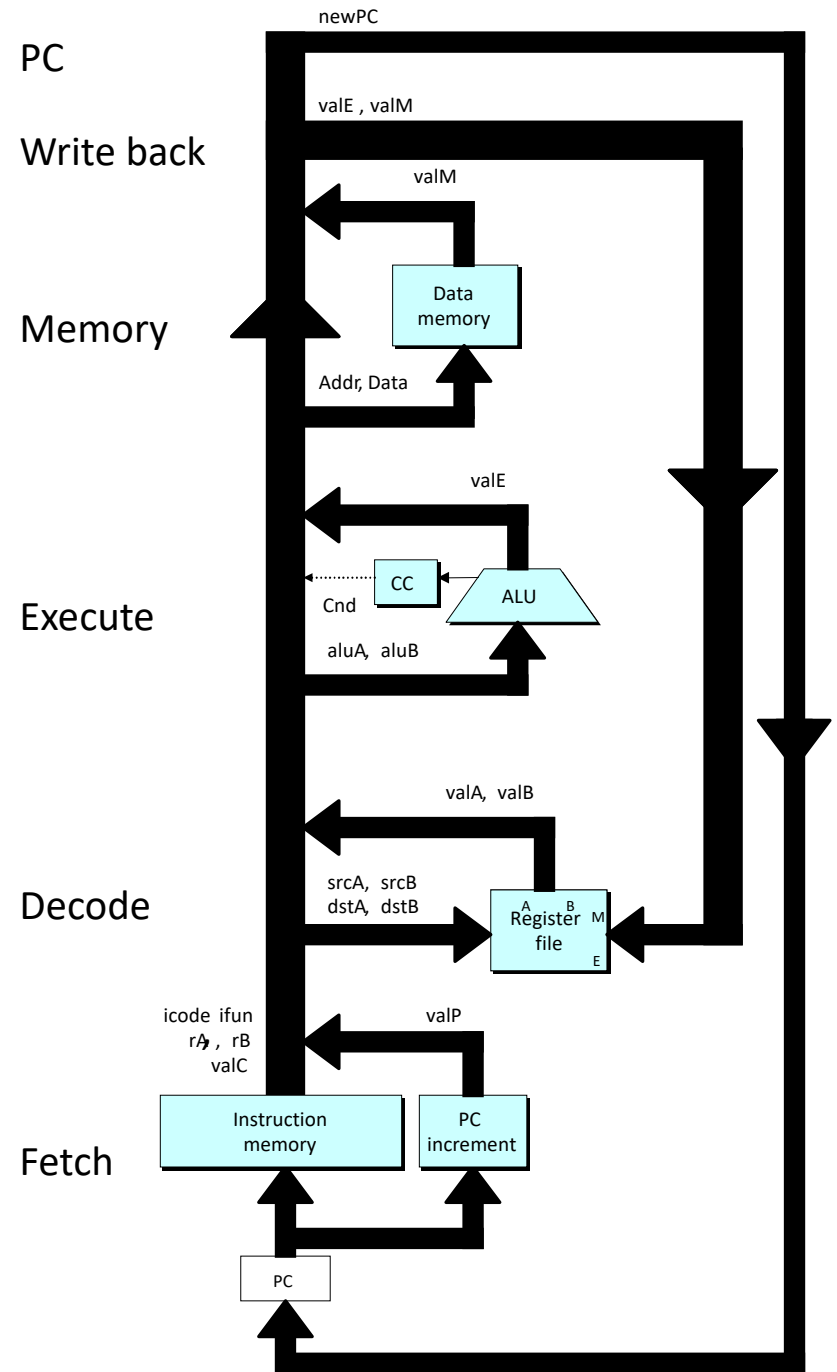
SEQ Hardware Structure

- 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

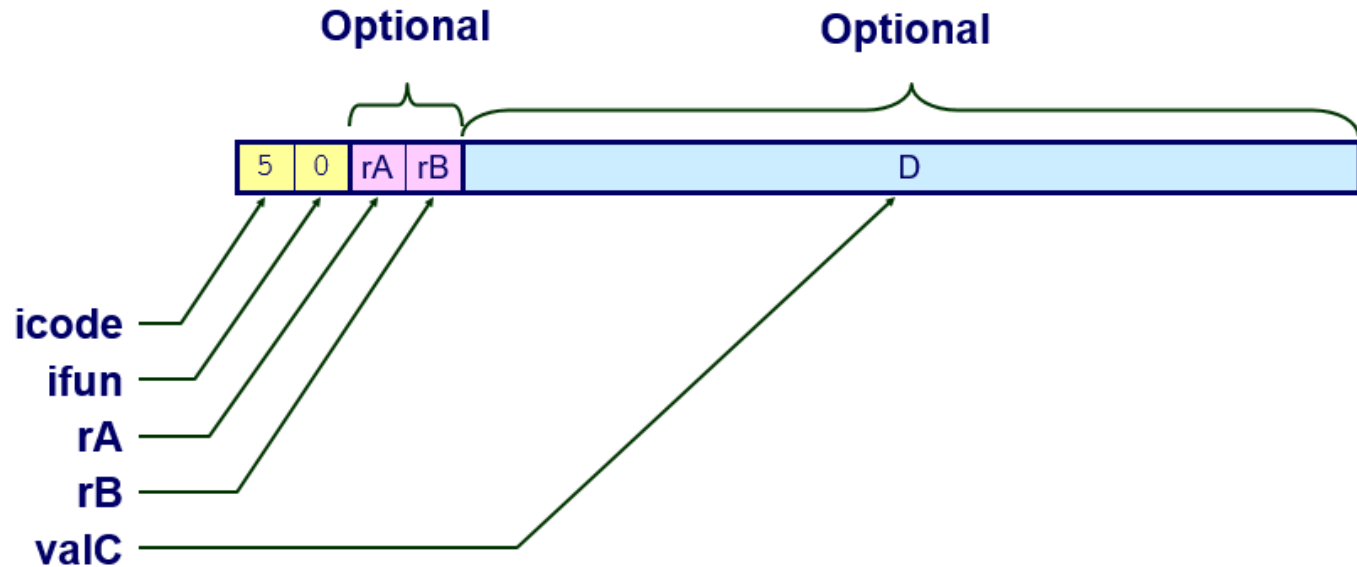


SEQ Stages

- 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 Arith./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	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+2$	Read instruction byte Read register byte Compute next PC
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	Read operand A Read operand B
Execute	$\text{valE} \leftarrow \text{valB OP valA}$ Set CC	Perform ALU operation Set condition code register
Memory		
Write back	$R[\text{rB}] \leftarrow \text{valE}$	Write back result
PC update	$\text{PC} \leftarrow \text{valP}$	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Executing `rmmovq`

`rmmovq rA, D(rB)`

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

- 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`

	<code>rmmovq rA, D(rB)</code>	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valC} \leftarrow M_8[\text{PC}+2]$ $\text{valP} \leftarrow \text{PC}+10$	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	Read operand A Read operand B
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Compute effective address
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	Write value to memory
Write back		
PC update	$\text{PC} \leftarrow \text{valP}$	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

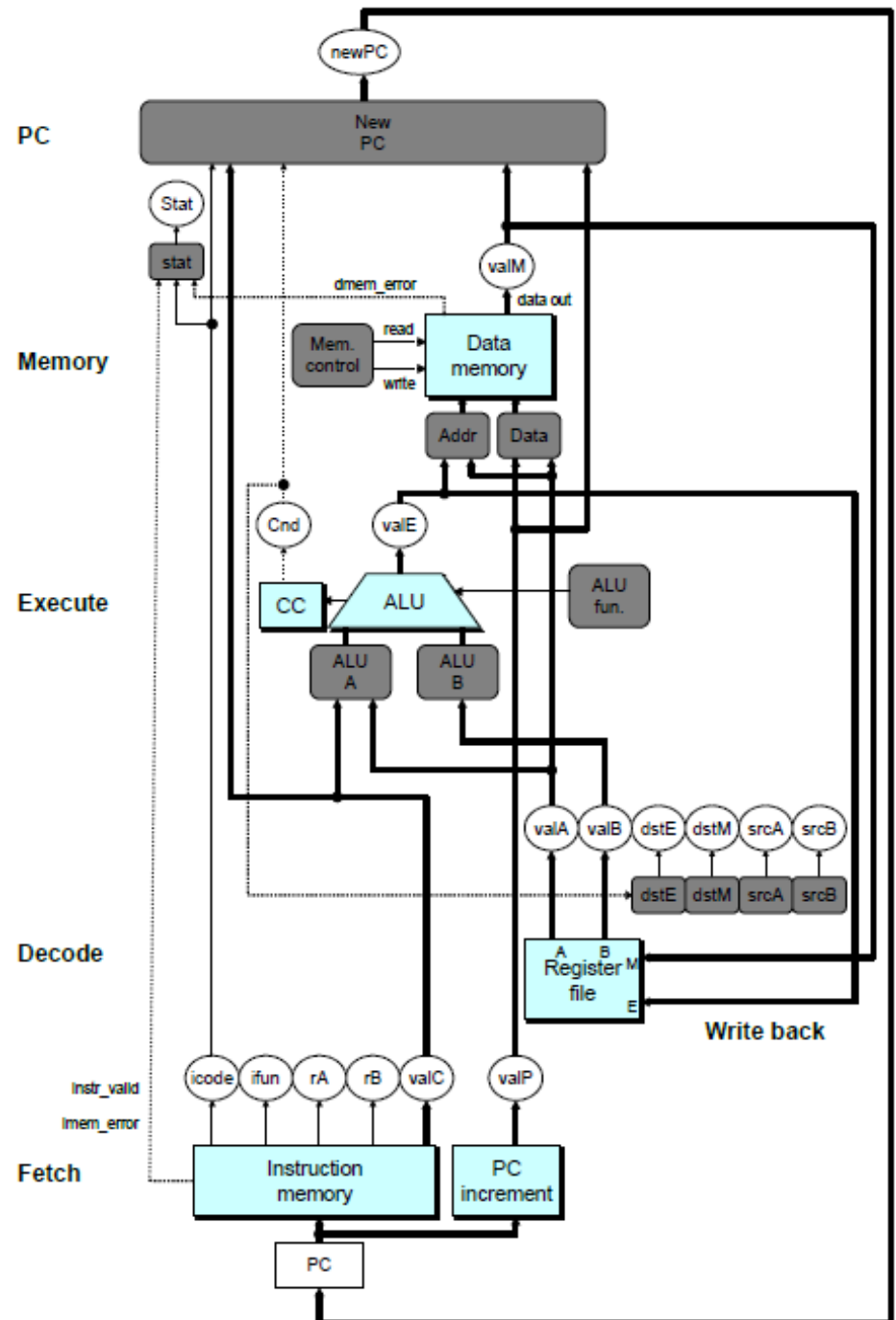
Stage Computation: popq

	popq rA	
Fetch	icode:ifun $\leftarrow M_1[PC]$	Read instruction byte
	rA:rB $\leftarrow M_1[PC+1]$	Read register byte
	valP $\leftarrow PC+2$	Compute next PC
Decode	valA $\leftarrow R[\%rsp]$	Read stack pointer
	valB $\leftarrow R[\%rsp]$	Read stack pointer
Execute	valE $\leftarrow valB + 8$	Increment stack pointer
Memory	valM $\leftarrow M_8[valA]$	Read from stack
Write back	$R[\%rsp] \leftarrow valE$	Update stack pointer
	$R[rA] \leftarrow valM$	Write back result
PC update	$PC \leftarrow valP$	Update PC

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

SEQ Hardware

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



SEQ Summary

- Implementation
 - Express every instruction as series of simple steps
 - Follow same general flow for each instruction type
 - Assemble registers, memories, predesigned combinational blocks
 - Connect with control logic
- Limitations
 - Too slow to be practical
 - In one cycle, must propagate through instruction memory, register file, ALU, and data memory
 - Would need to run clock very slowly
 - Hardware units only active for fraction of clock cycle