Operating Systems & C

Lecture 2: Processors

Willard Rafnsson
IT University of Copenhagen

ITU CPH

Digital Logic

ITU CPH

Overview of Logic Design

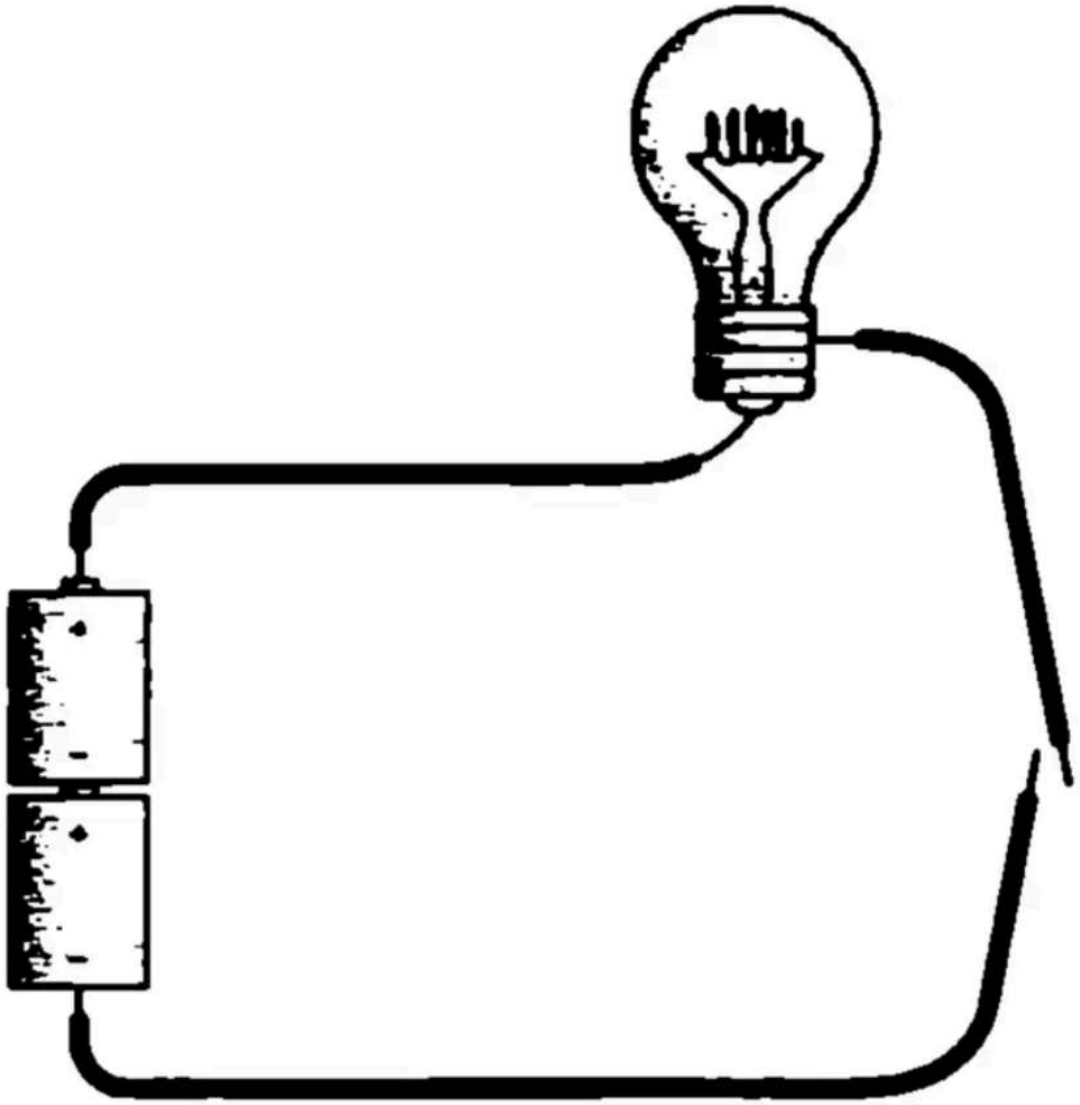
Fundamental Hardware Requirements

- Communication
  - How to get values from one place to another
- Computation
- Storage

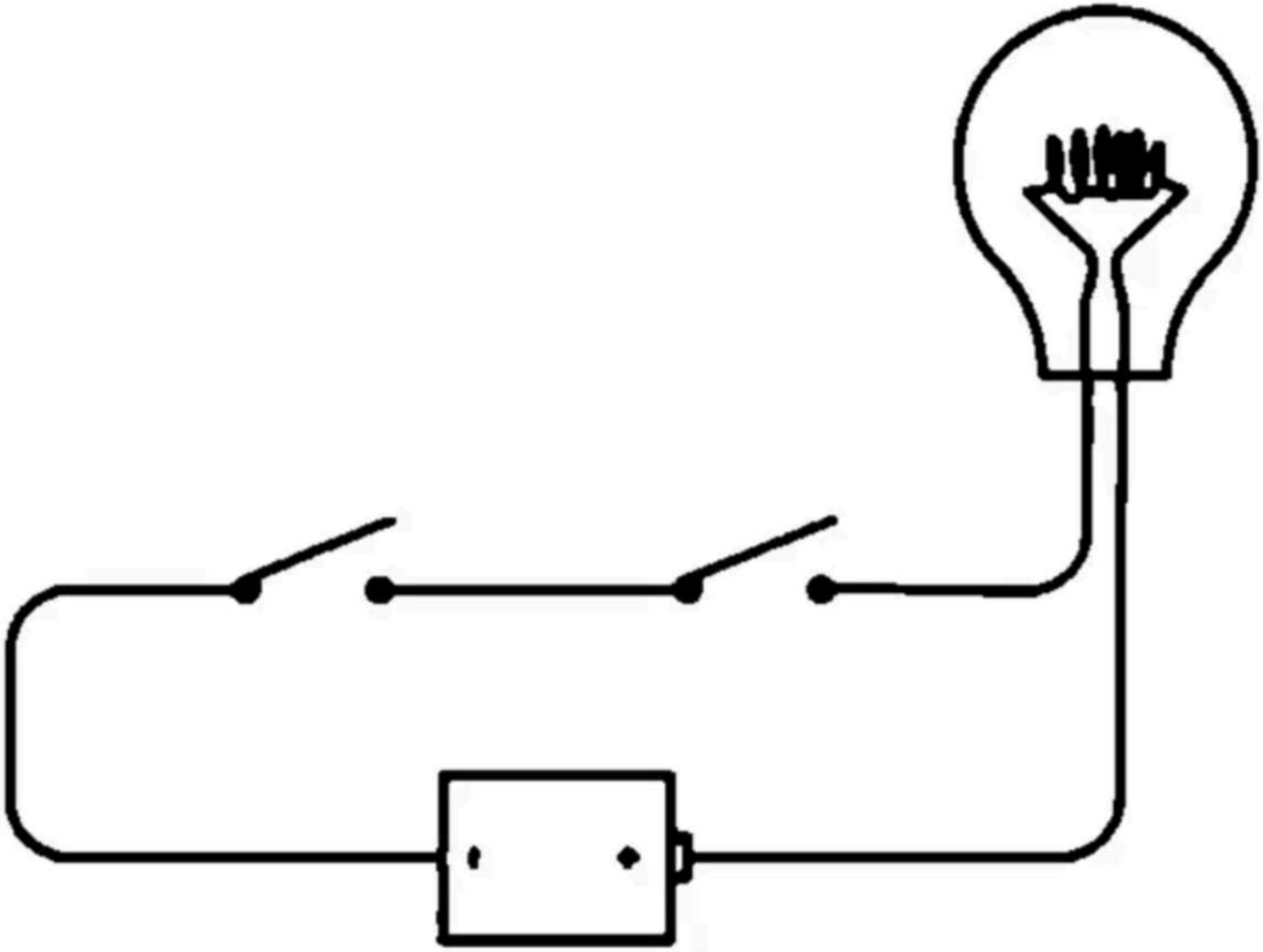
Bits are Our Friends

- 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

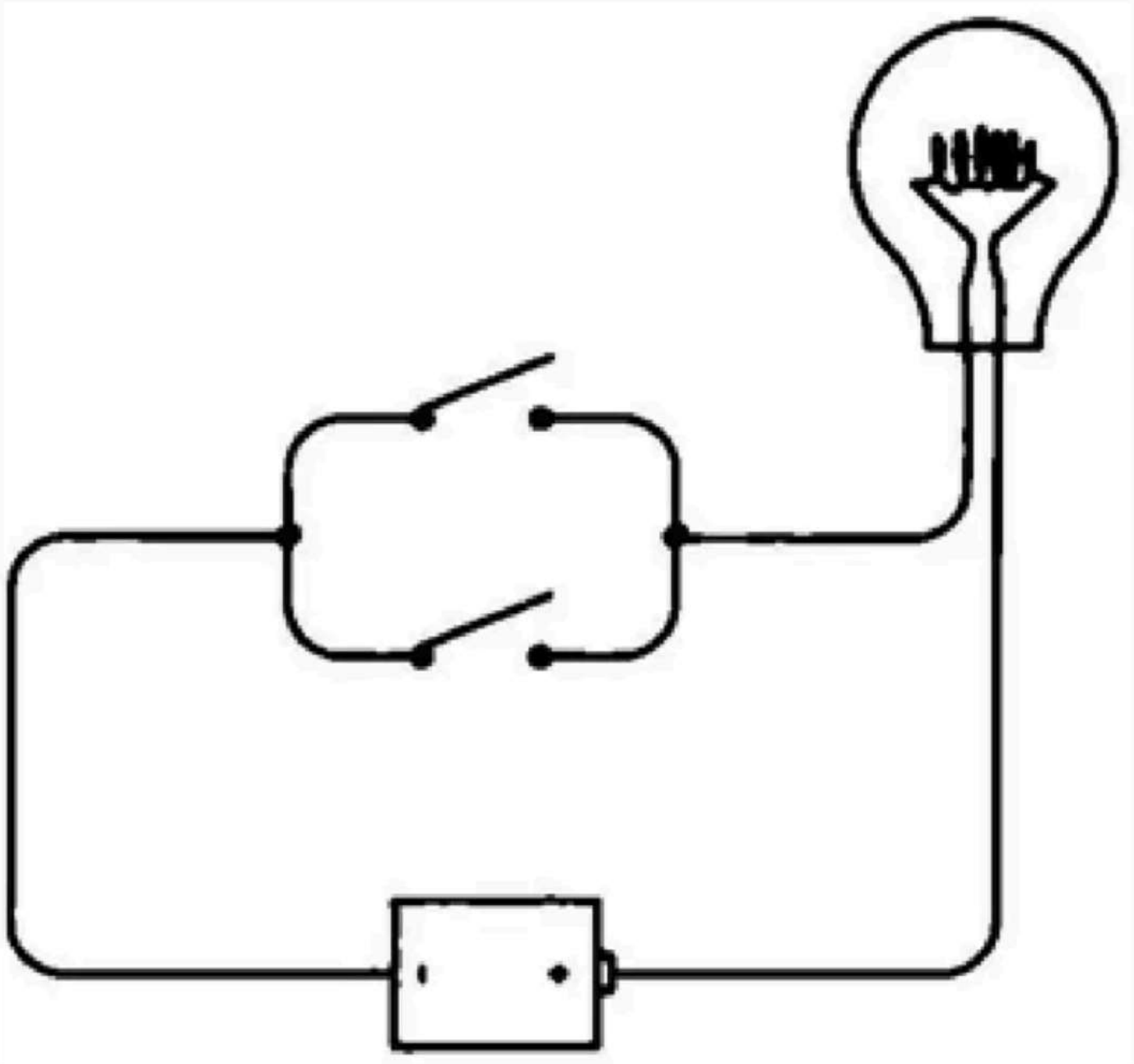
Anatomy of a Flashlight



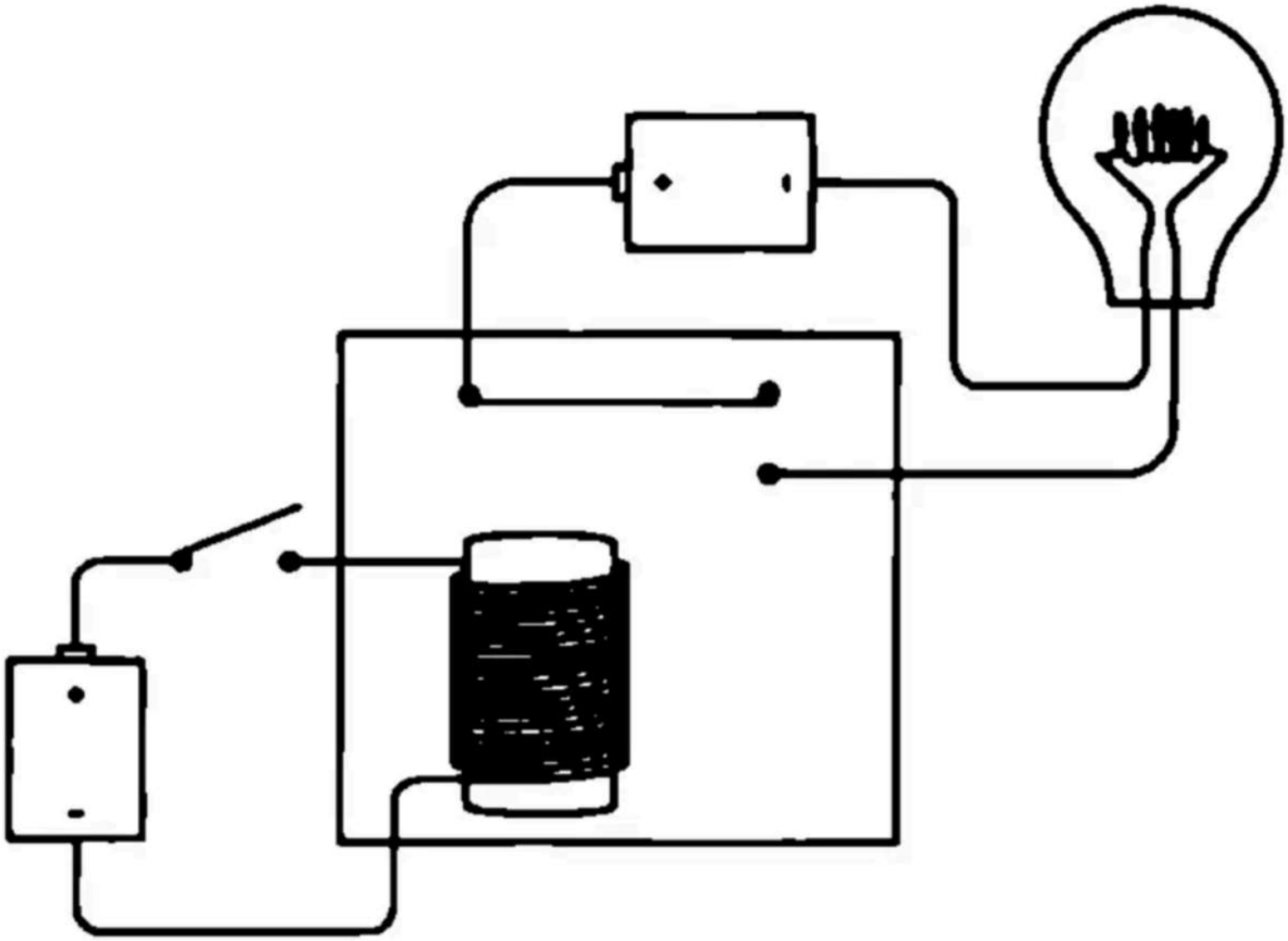
ITU CPH



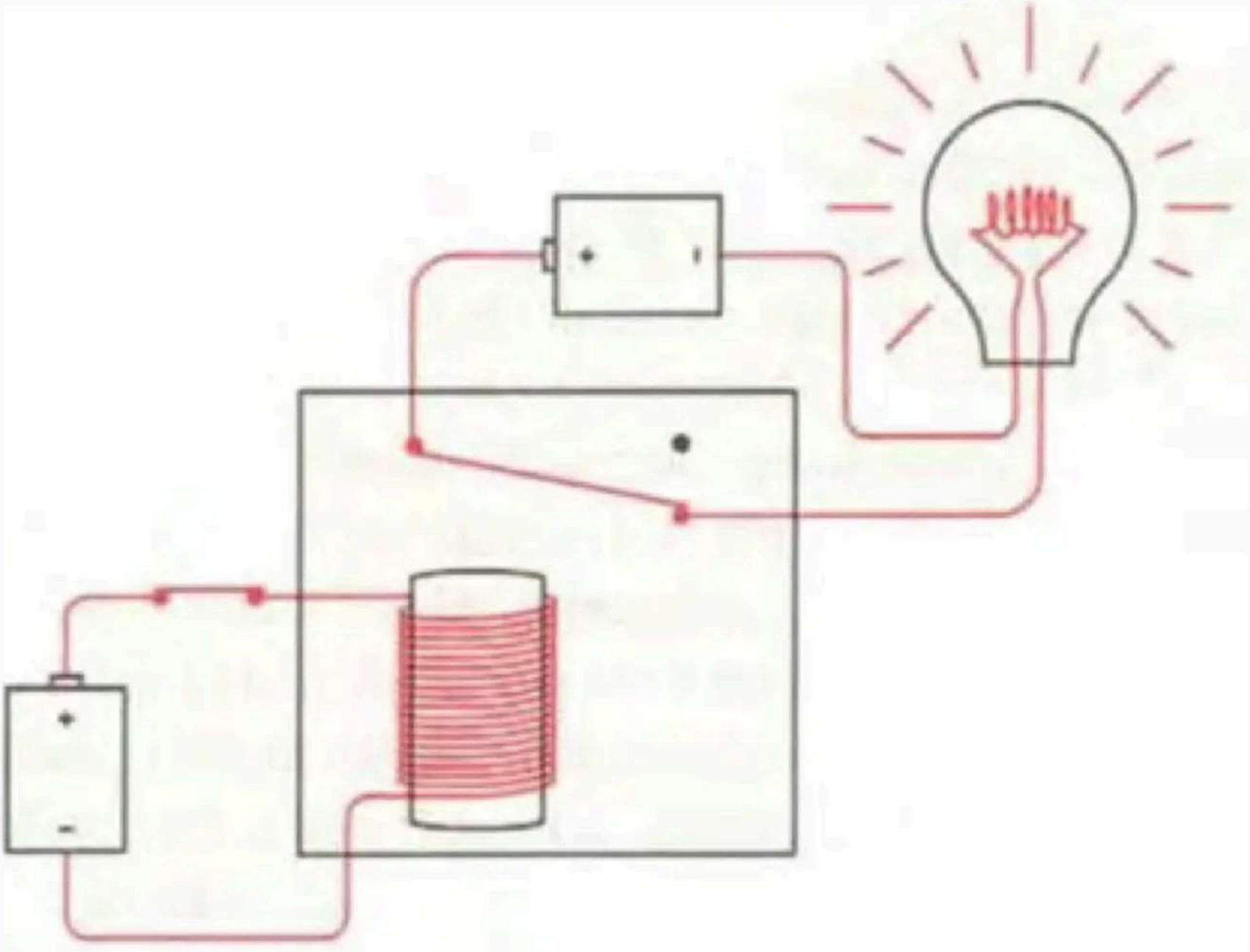
ITU CPH



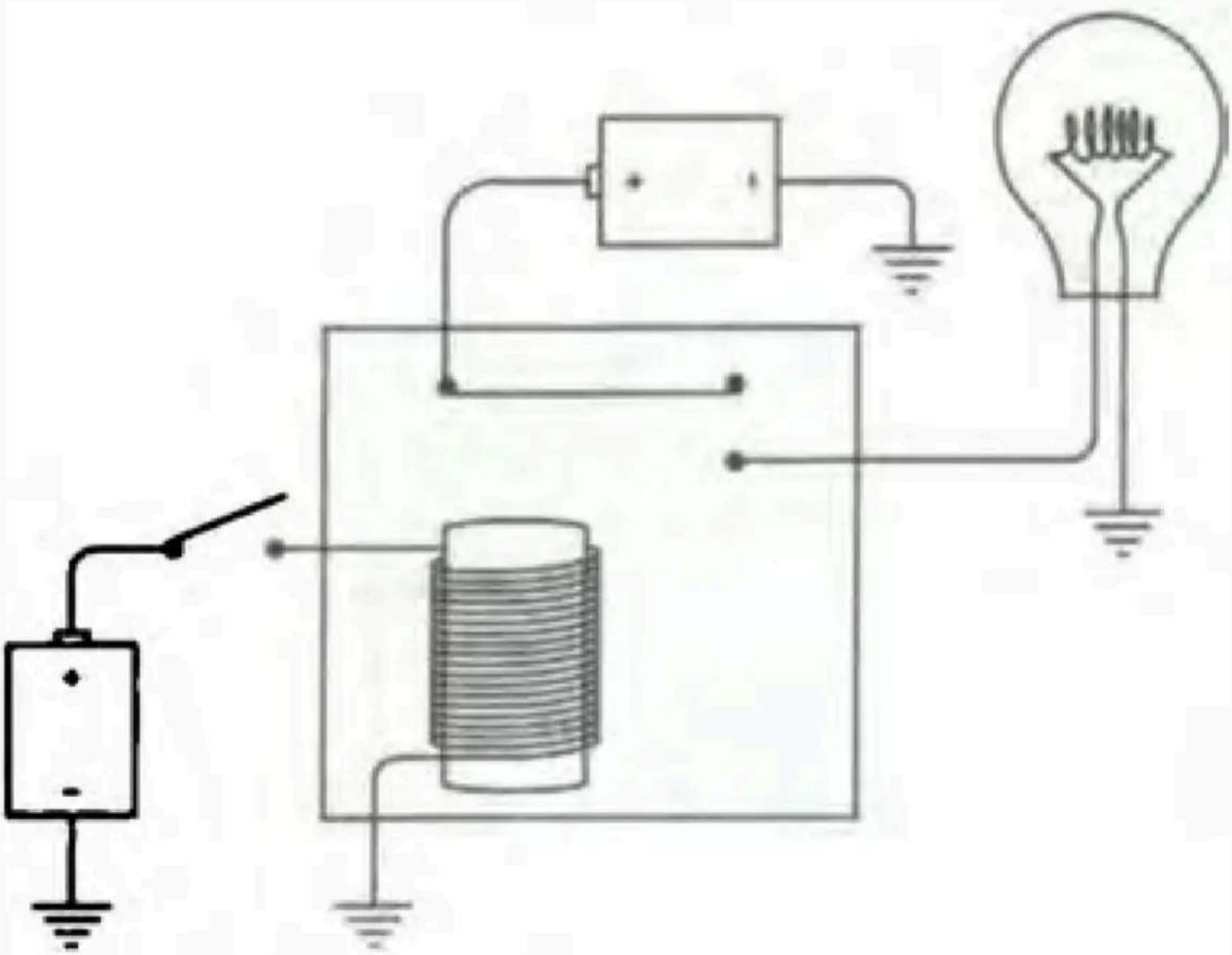
ITU CPH



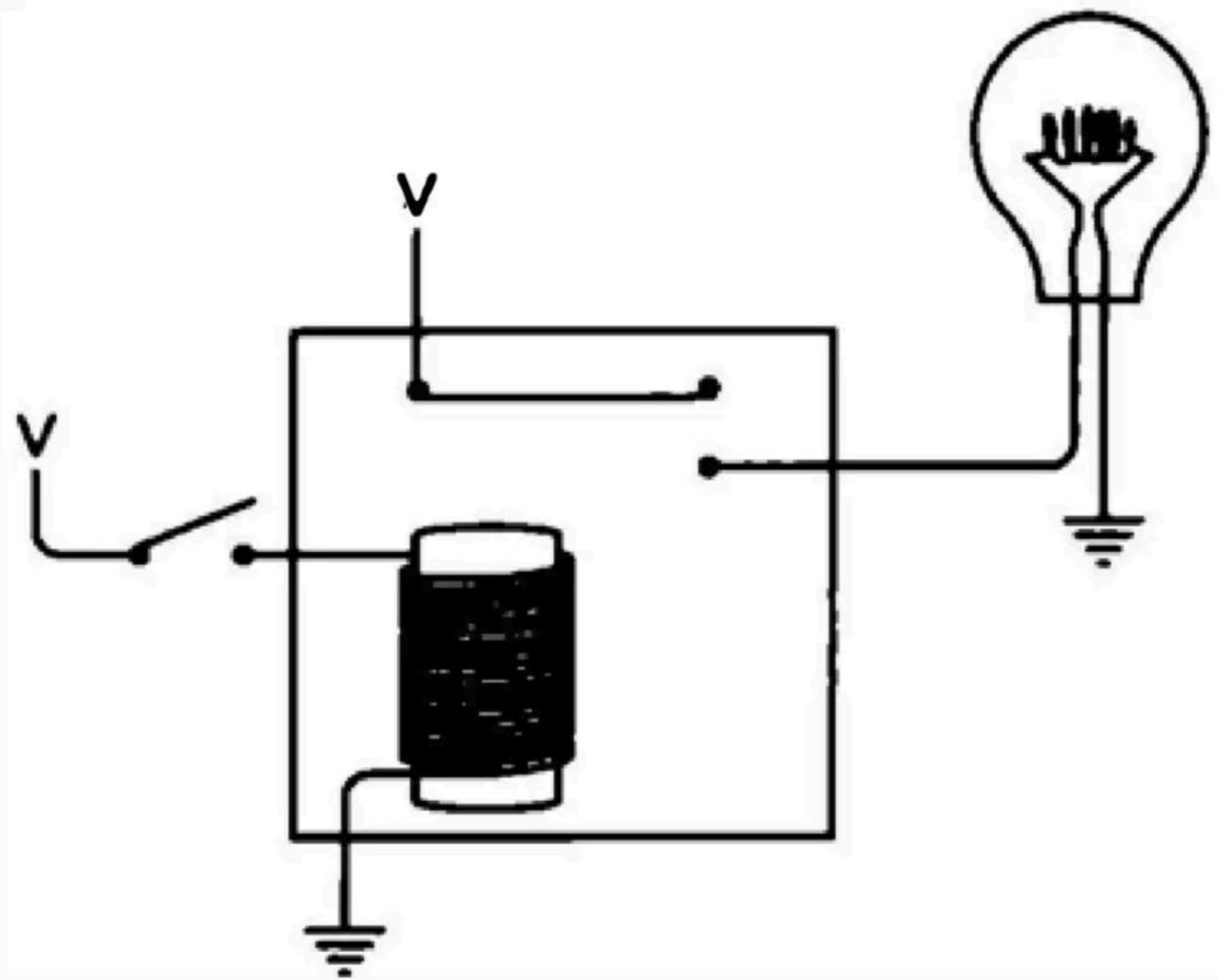
ITU CPH



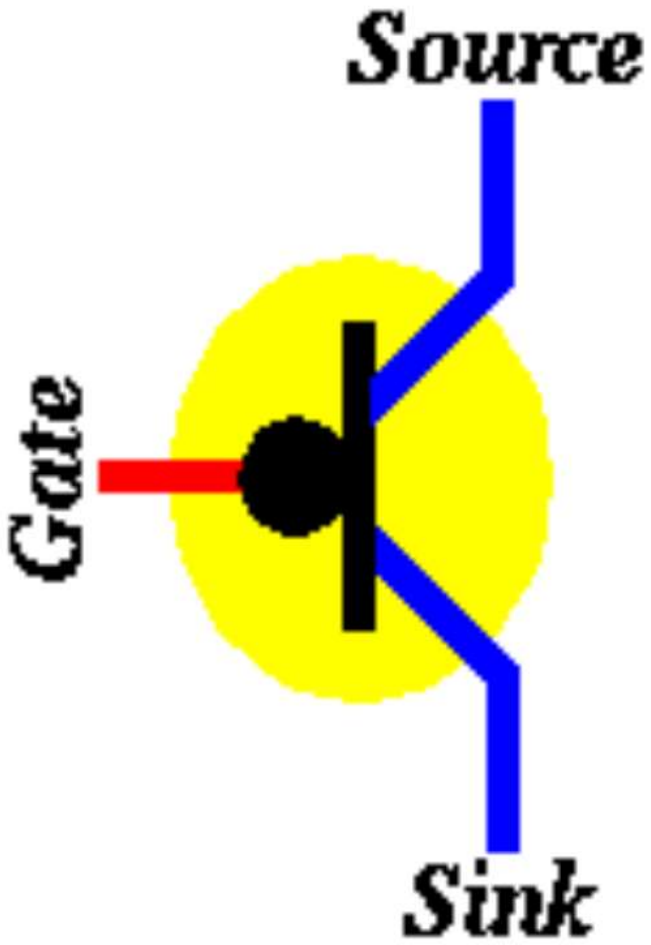
ITU CPH



ITU CPH



ITU CPH



gure 4: Symbol used to depict a complementary transisto

ITU CPH

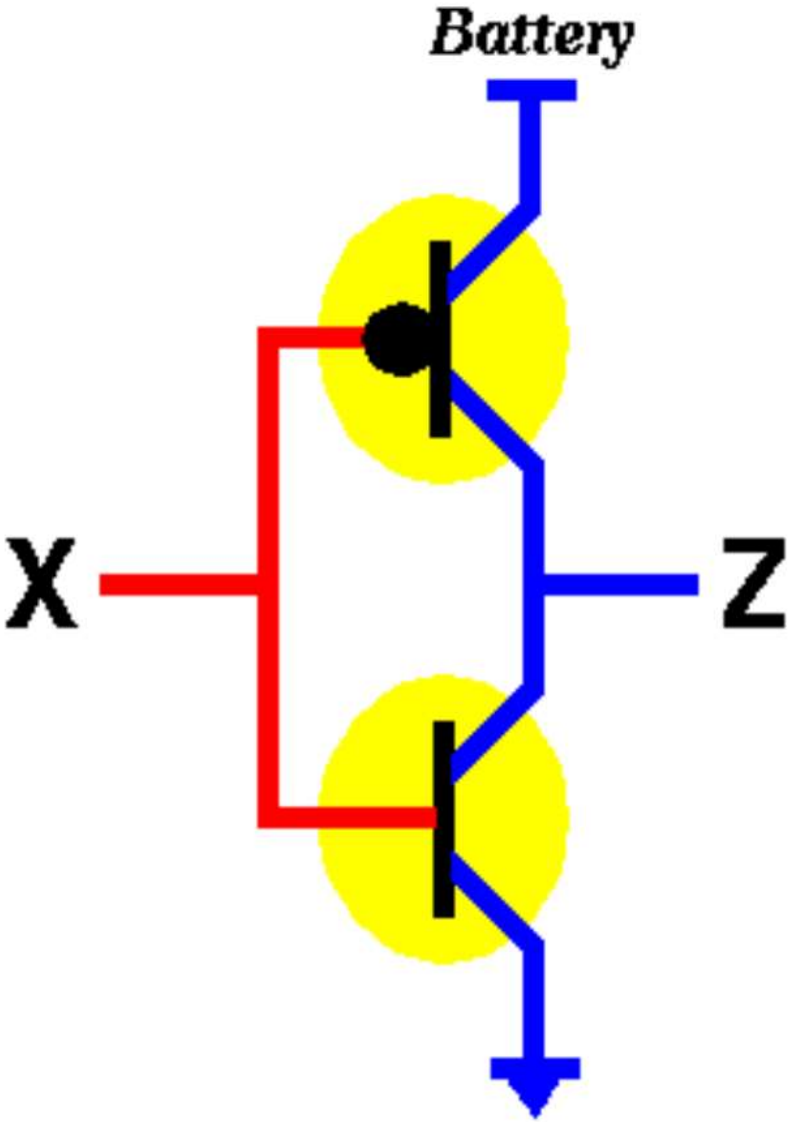


Figure 5: An inverter circuit (known as the NOT gate).

ITU CPH

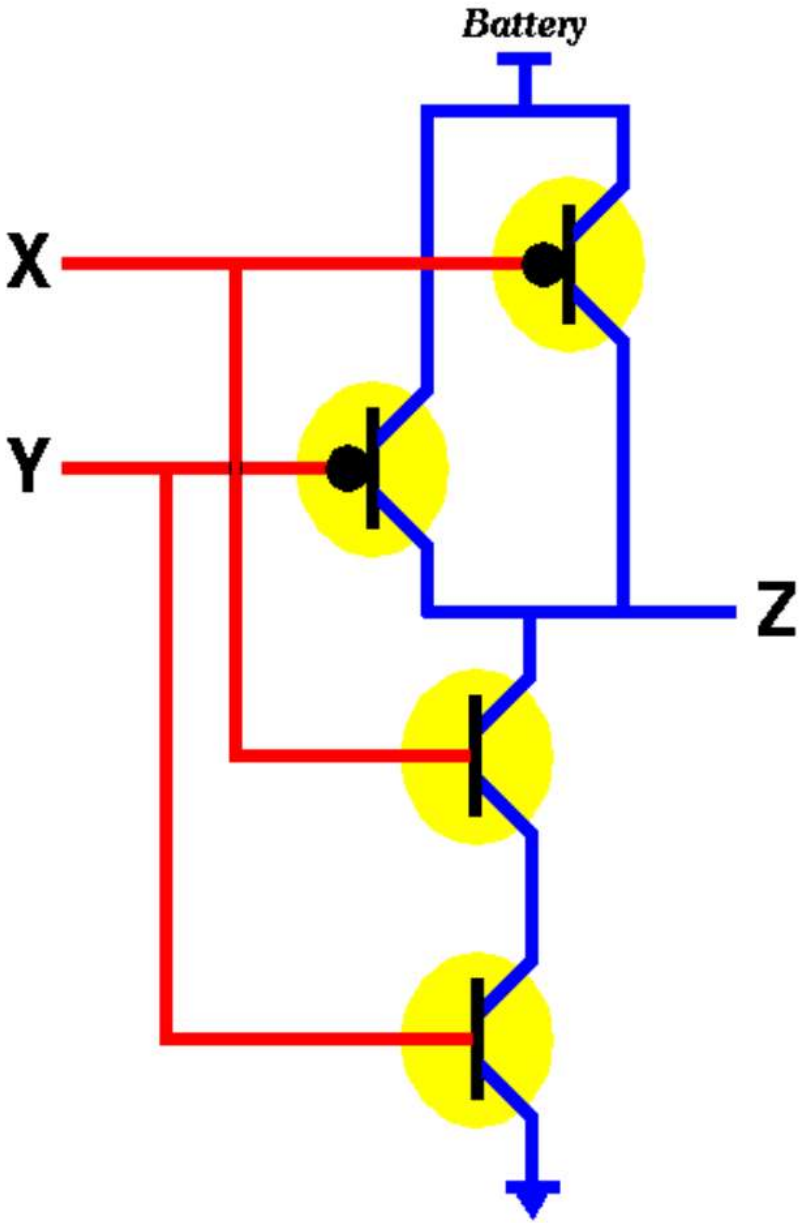
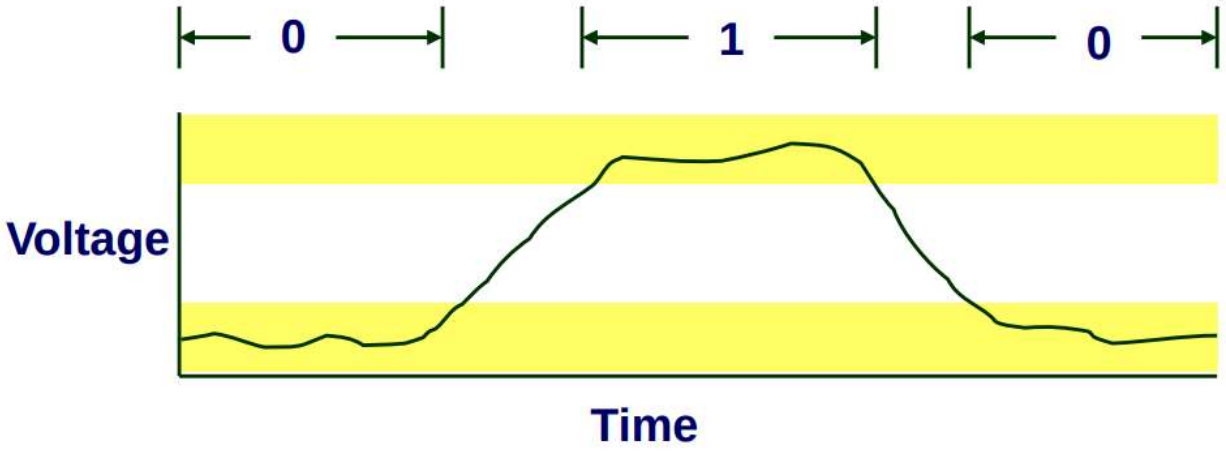


Figure 7: The "not both X and Y" circuit (the NAND function).

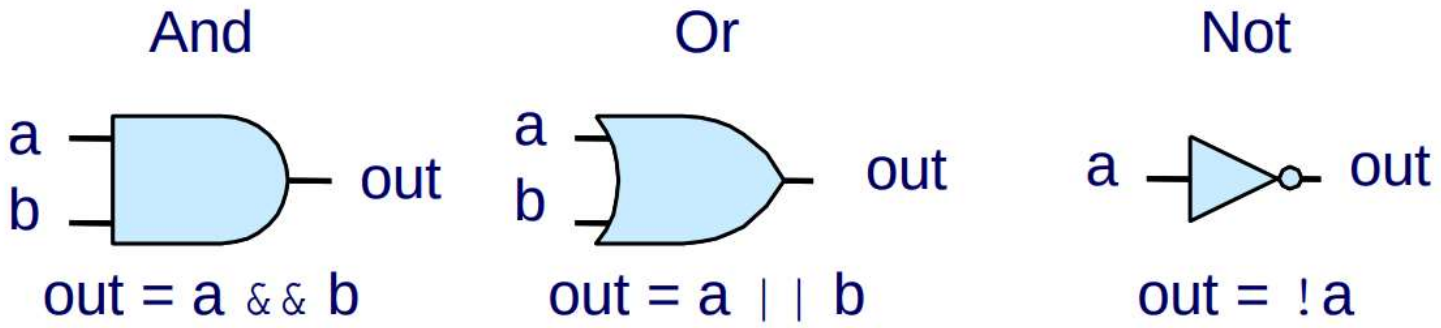
ITU CPH

Digital Signals

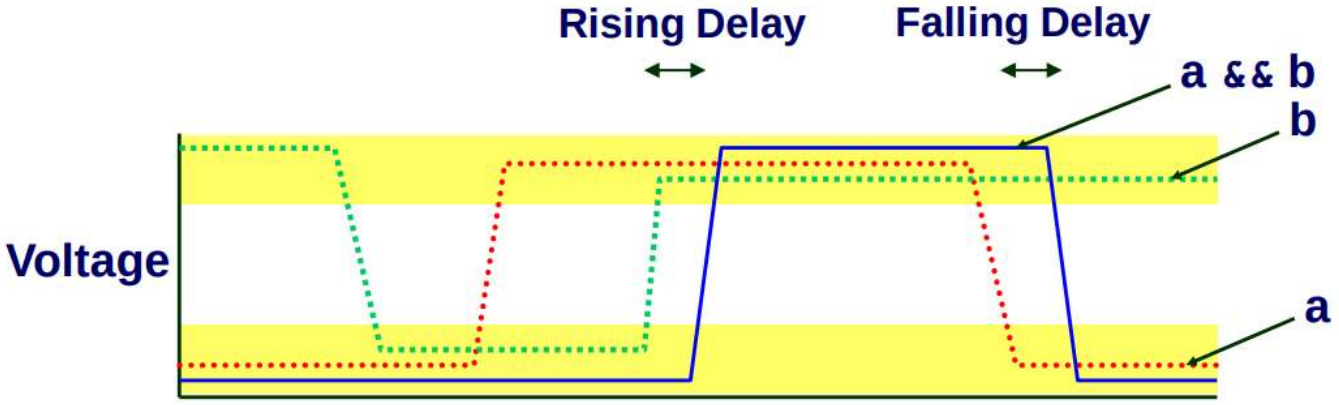


- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low quality circuit elements
  - Can make circuits simple, small, and fast

Computing with Logic Gates



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



Today

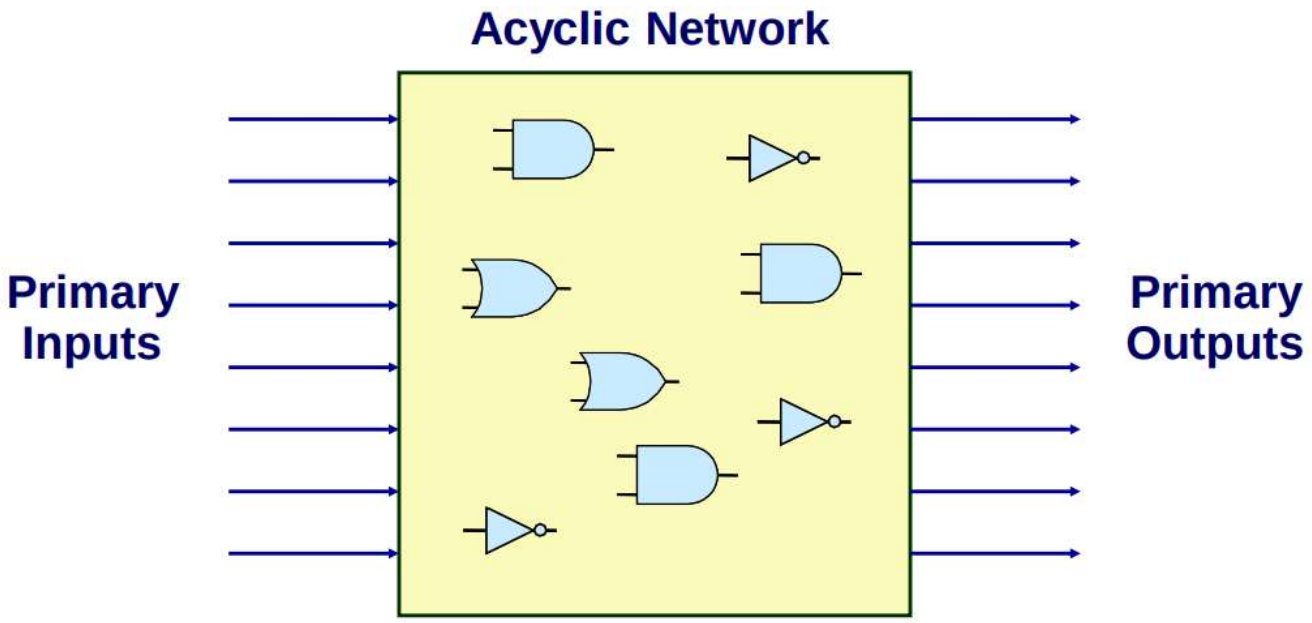
- Digital Logic
  - Combinatorial Logic Circuits (functions)
  - Sequential Logic Circuits (state)
- Processor: Y86-64
  - Design (ISA), Semantics, Implementation (briefly)
- Stack
  - Procedure calls
  - Buffer overflow

ITU CPH

Combinatorial Logic Circuits

ITU CPH

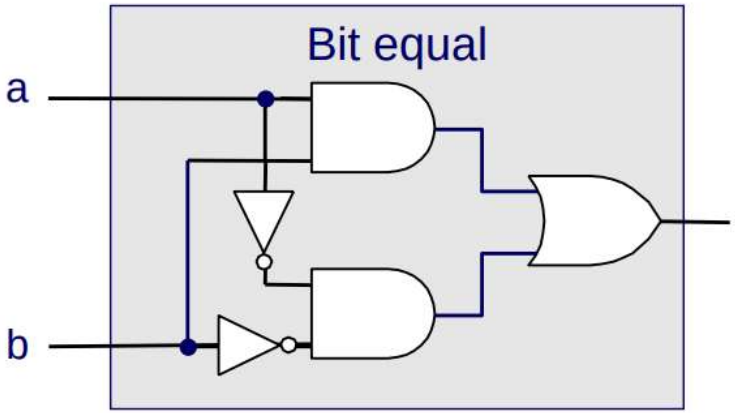
Combinational Circuits



Acyclic Network of Logic Gates

- Continously responds to changes on primary inputs
- Primary outputs become (after some delay) Boolean functions of primary inputs

Bit Equality



HCL Expression

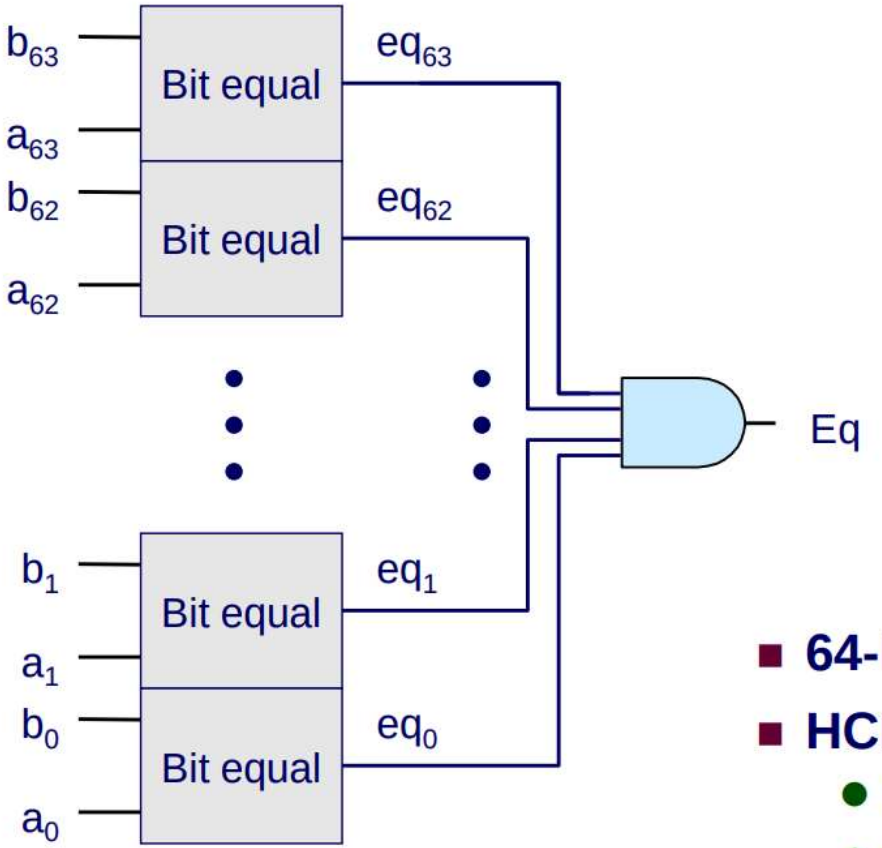
```
bool eq = (a&&b)||(!a&&!b)
```

- Generate 1 if a and b are equal

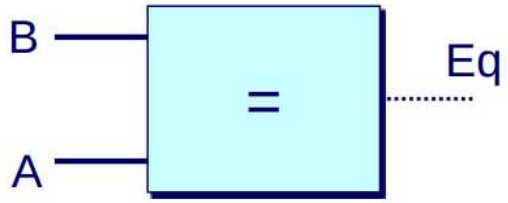
Hardware Control Language (HCL)

- 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

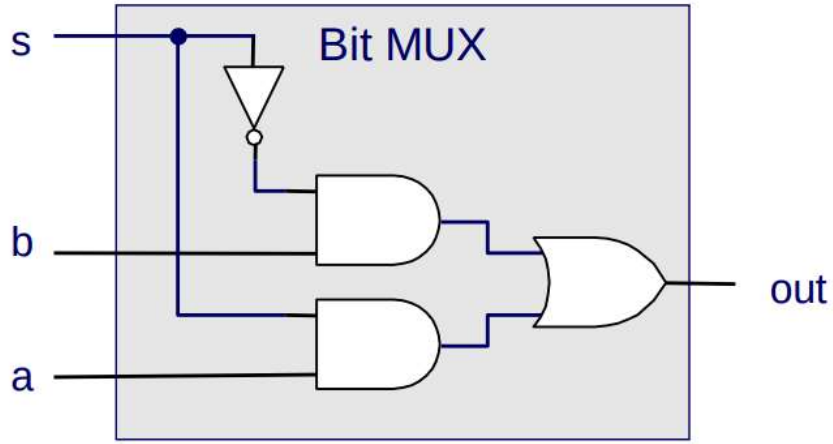


HCL Representation

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

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

Bit-Level Multiplexor

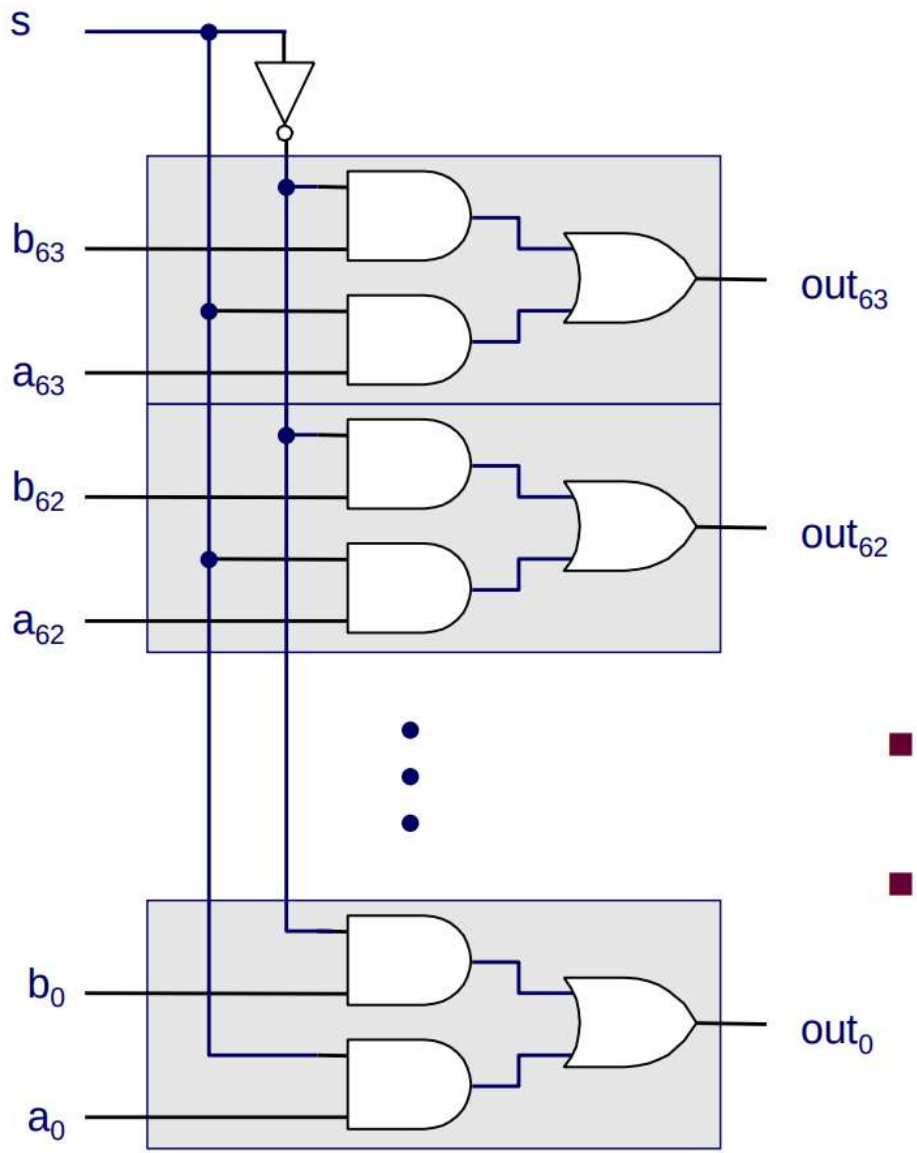


HCL Expression

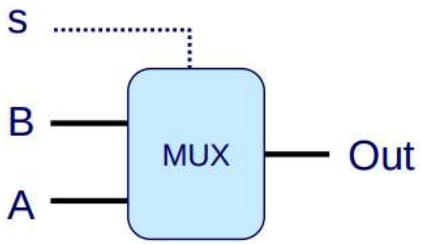
```
bool out = (s&&a)||(!s&&b)
```

- 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;
];
```

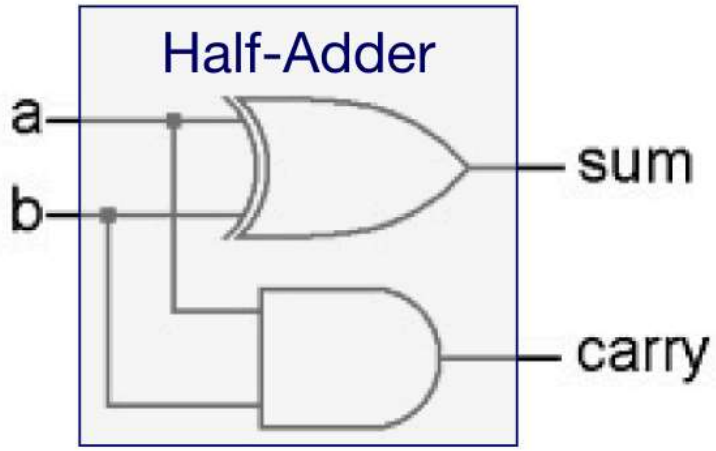
- Select input word A or B depending on control signal s
- HCL representation
  - Case expression
  - Series of test : value pairs
  - Output value for first successful test

CS:APP3e

– 9 –

ITU CPH

Adder, Half-

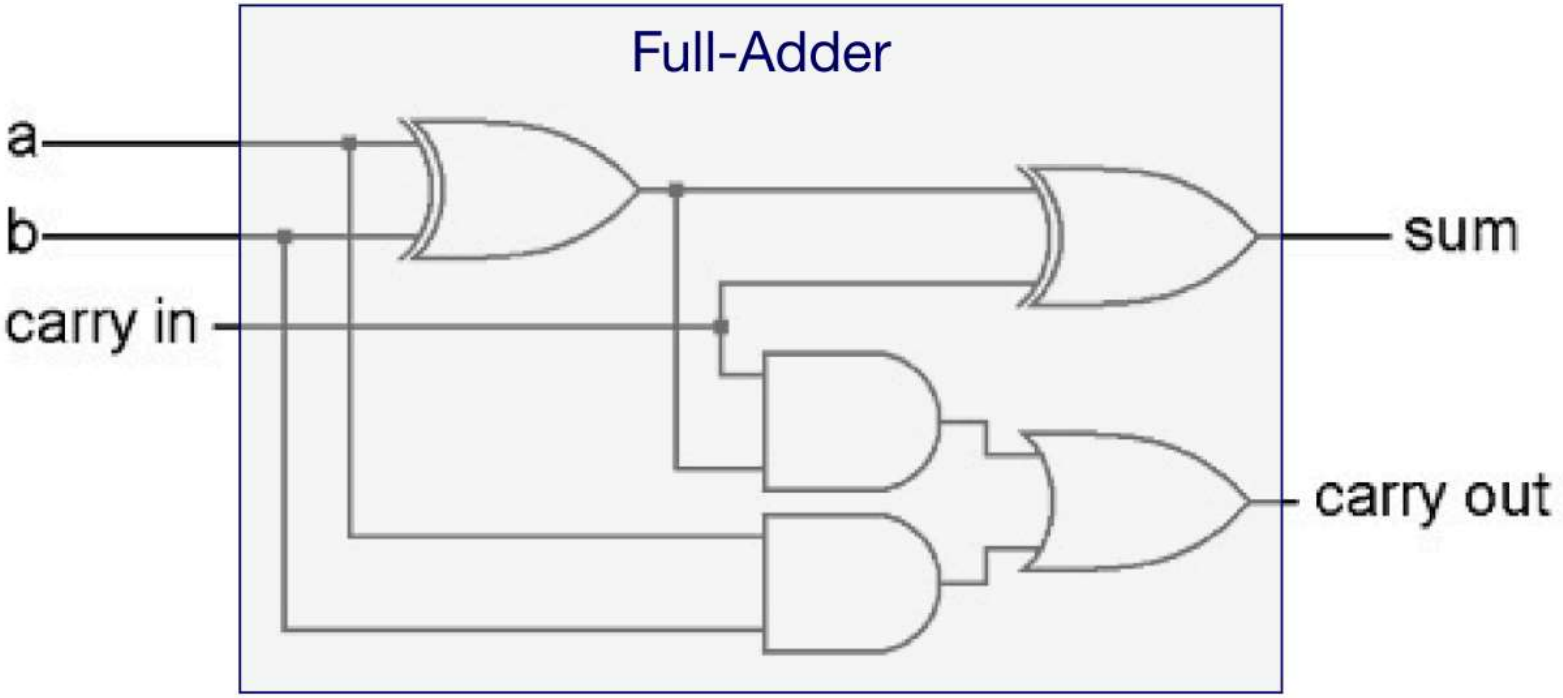


HCL Expression

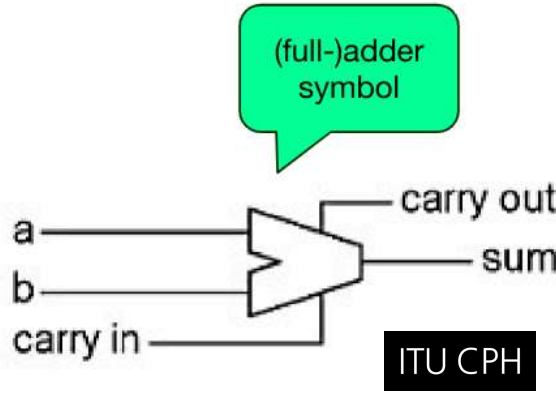
```
bool sum   = (a^^b)
bool carry = a&&b
```

- **Output binary-sum to `sum`, carry-bit to `carry`.**

Adder, Full-

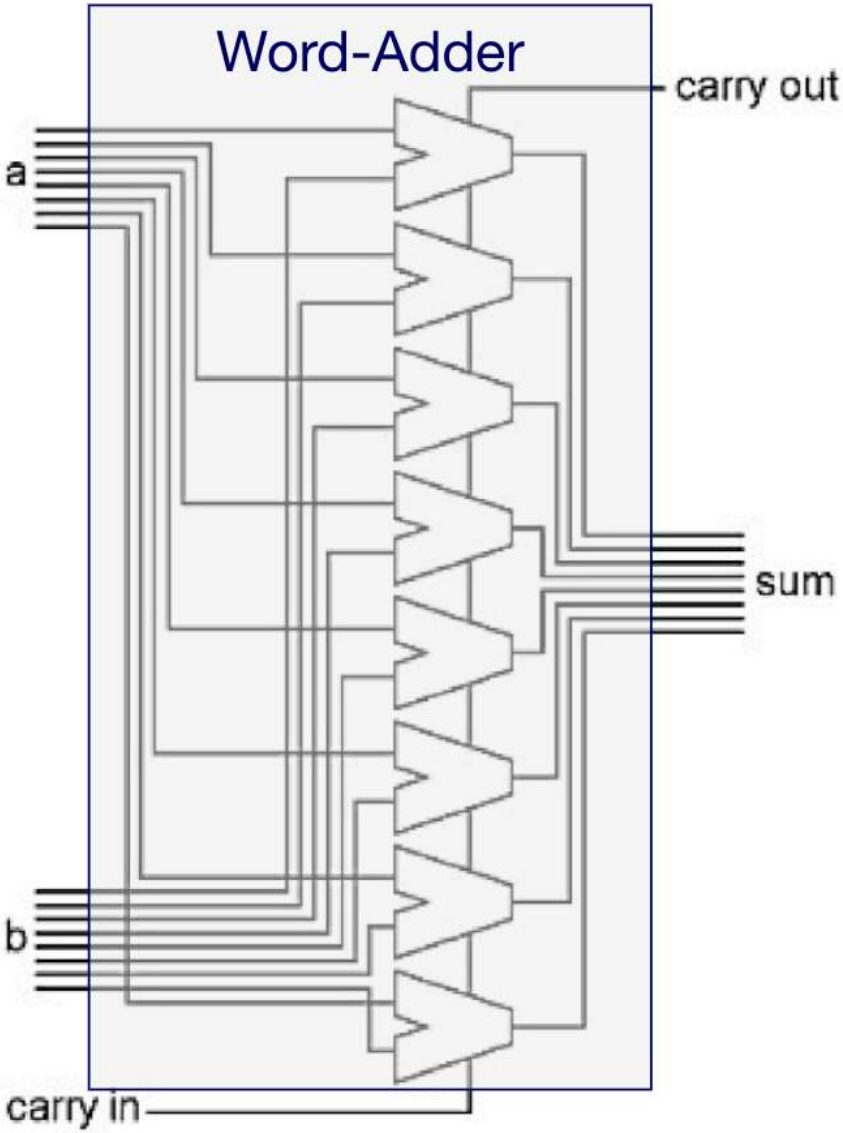


- A Half-adder with a carry-in bit.



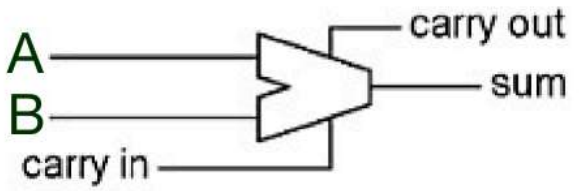
ITU CPH

Adder, Word-



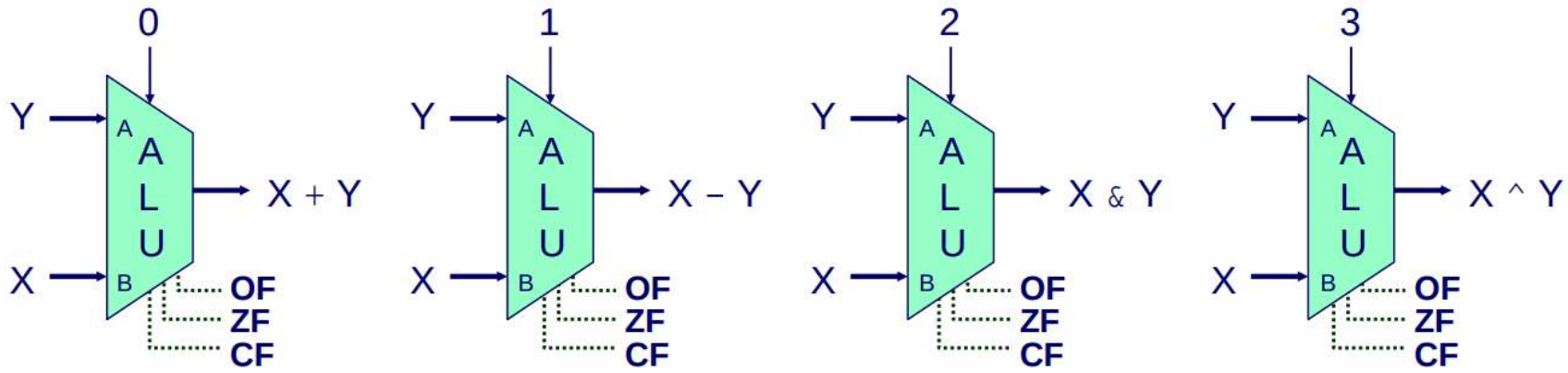
- **Chain the 1-bit adders, carry-out to carry-in, to get a word-adder.**

Word-Level Representation



ITU CPH

Arithmetic Logic Unit



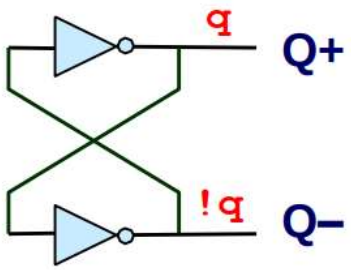
- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86-64
- Also computes values for condition codes

Sequential Logic Circuits

ITU CPH

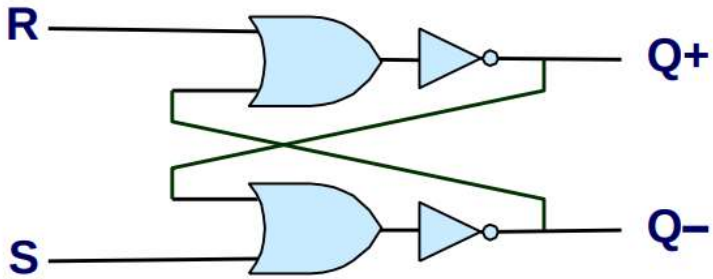
Storing and Accessing 1 Bit

Bistable Element

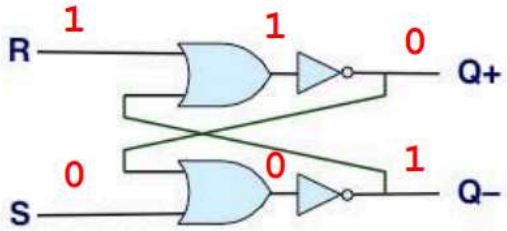


q = 0 or 1

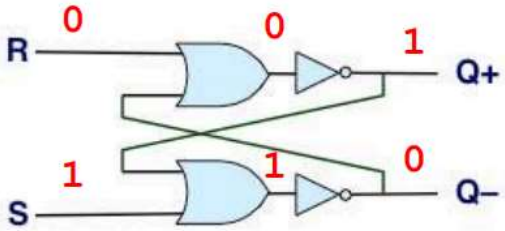
R-S Latch



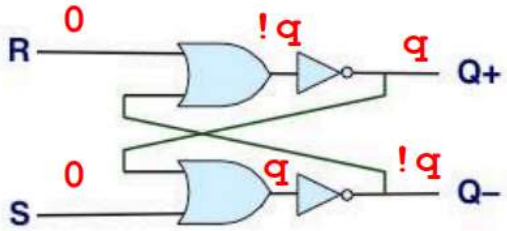
Resetting



Setting



Storing

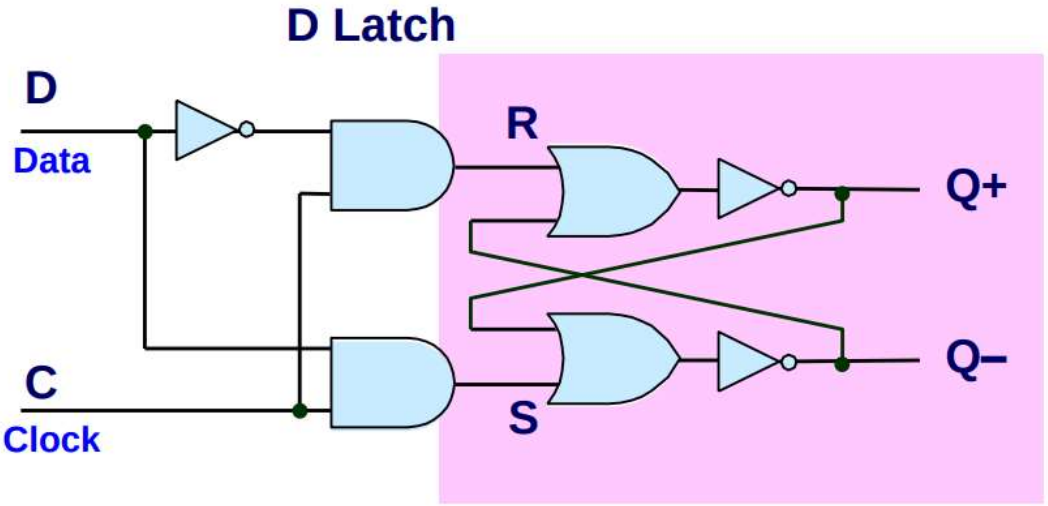


– 15 –

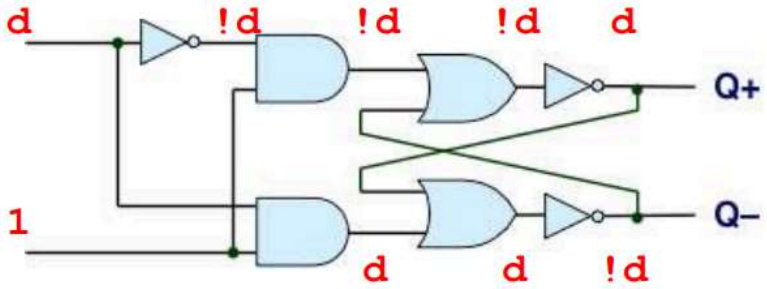
CS:APP3e

ITU CPH

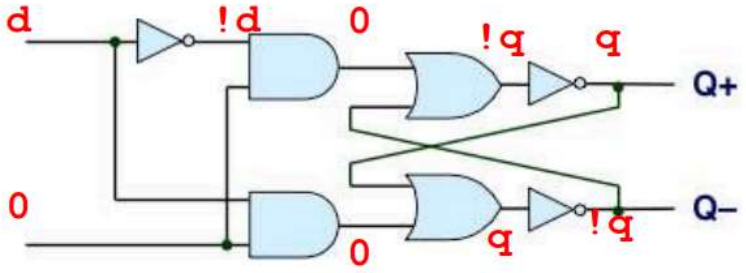
1-Bit Latch



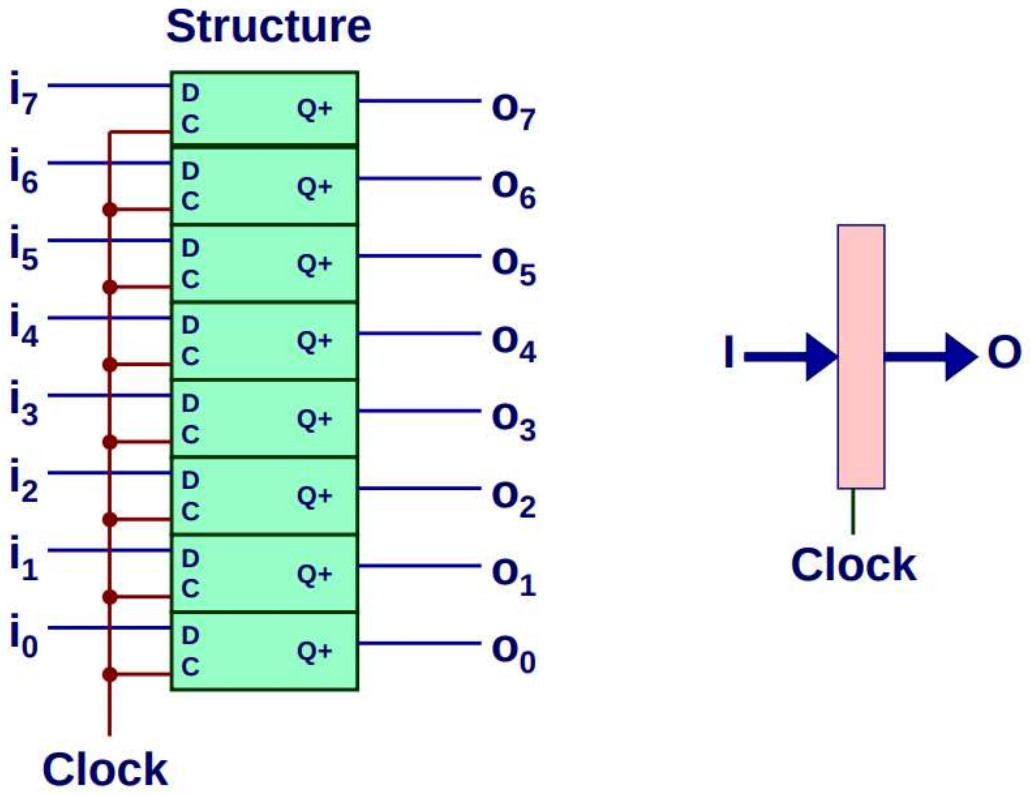
Latching



Storing

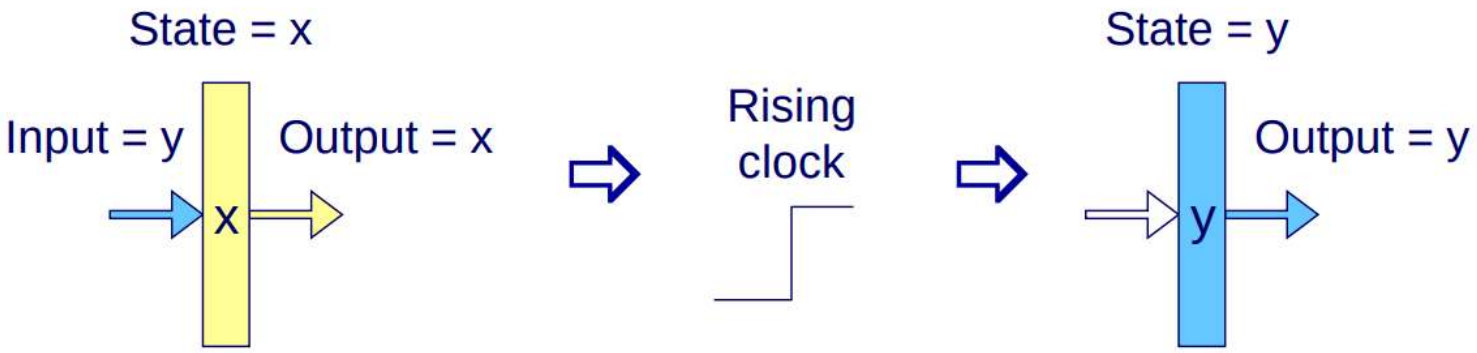


Registers



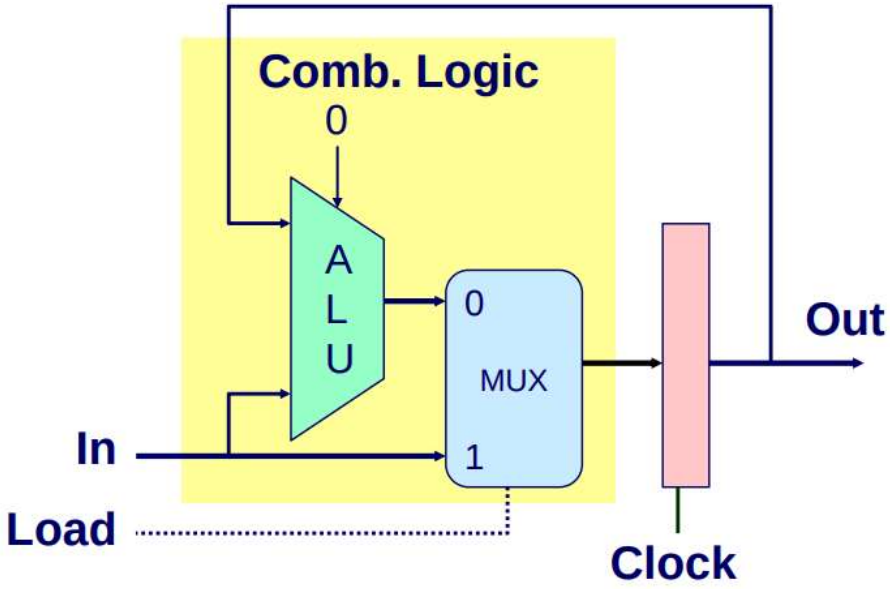
- Stores word of data
  - Different from *program registers* seen in assembly code
- Collection of edge-triggered latches
- Loads input on rising edge of clock

Register Operation

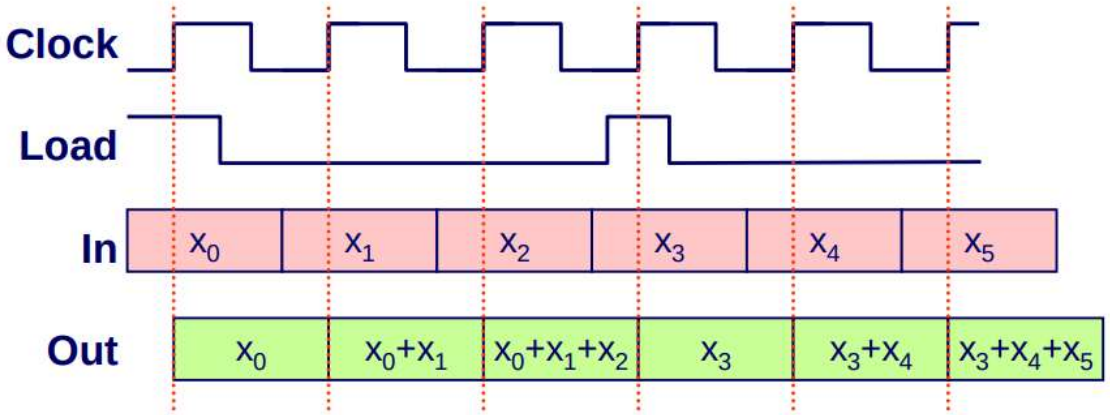


- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

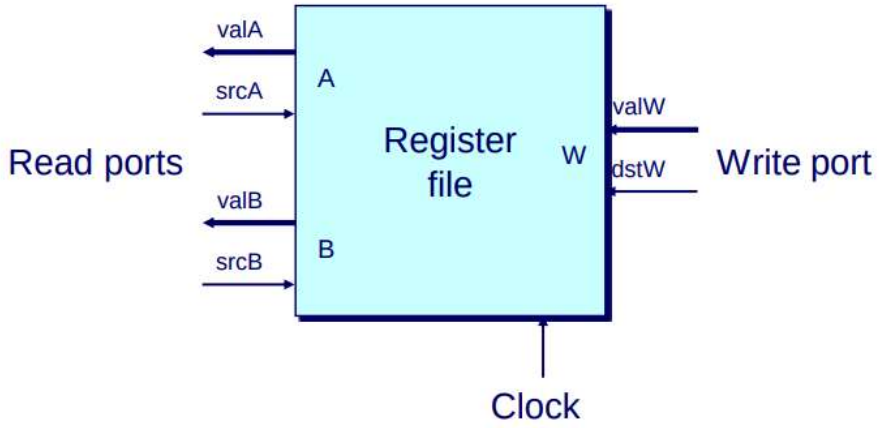
State Machine Example



- 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

Summary

Computation

- Performed by combinational logic
- Computes Boolean functions
- Continuously reacts to input changes

Storage

- Registers
  - Hold single words
  - Loaded as clock rises
- Random-access memories
  - Hold multiple words
  - Possible multiple read or write ports
  - Read word when address input changes
  - Write word as clock rises

A Processor: Y86-64 Design (ISA)

ITU CPH

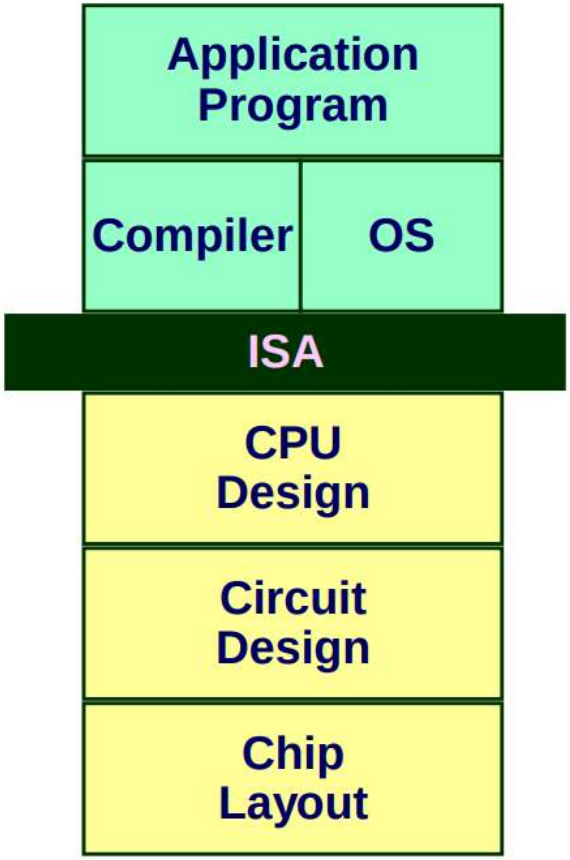
Instruction Set Architecture

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

CS:APP3e

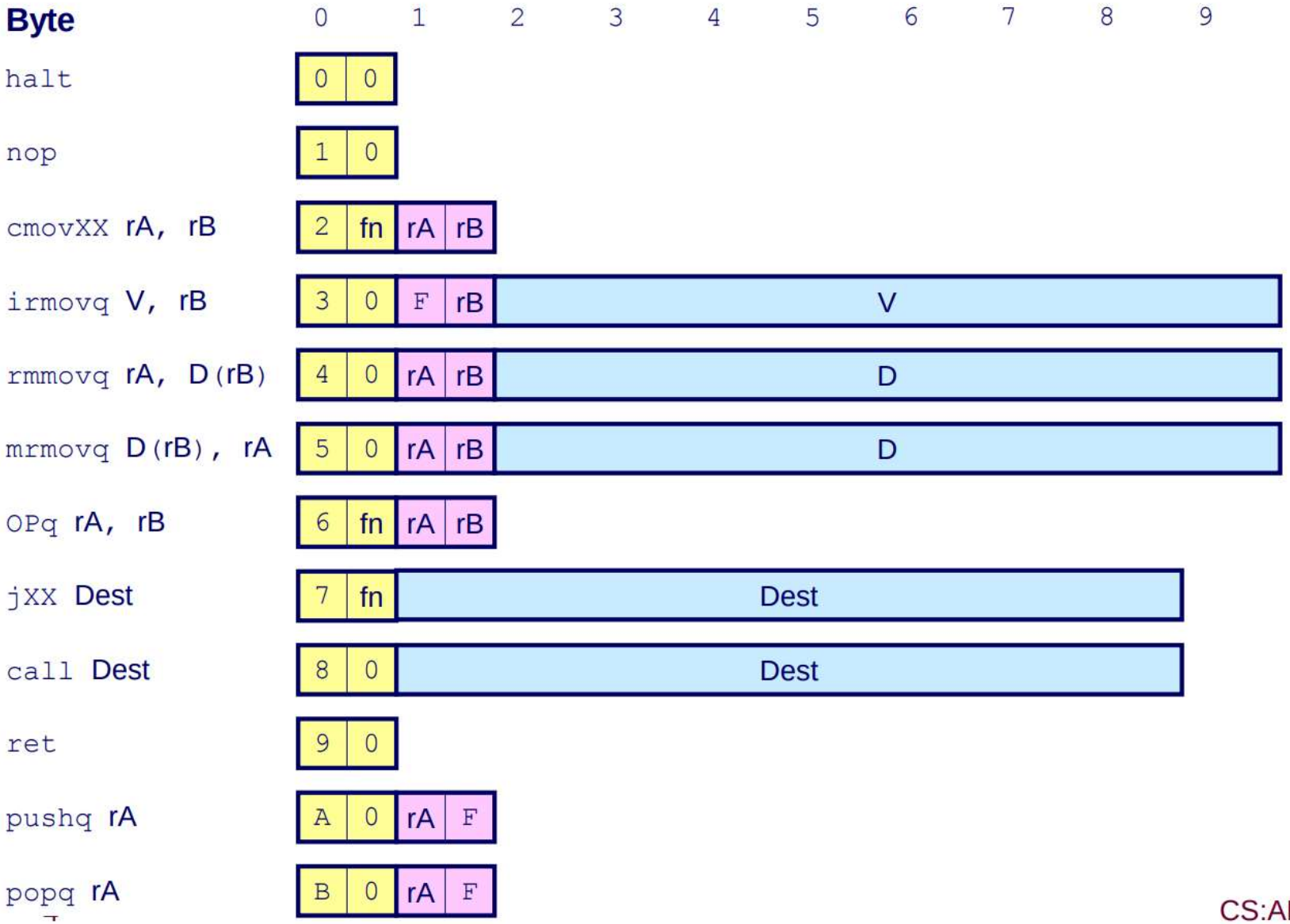
ITU CPH

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

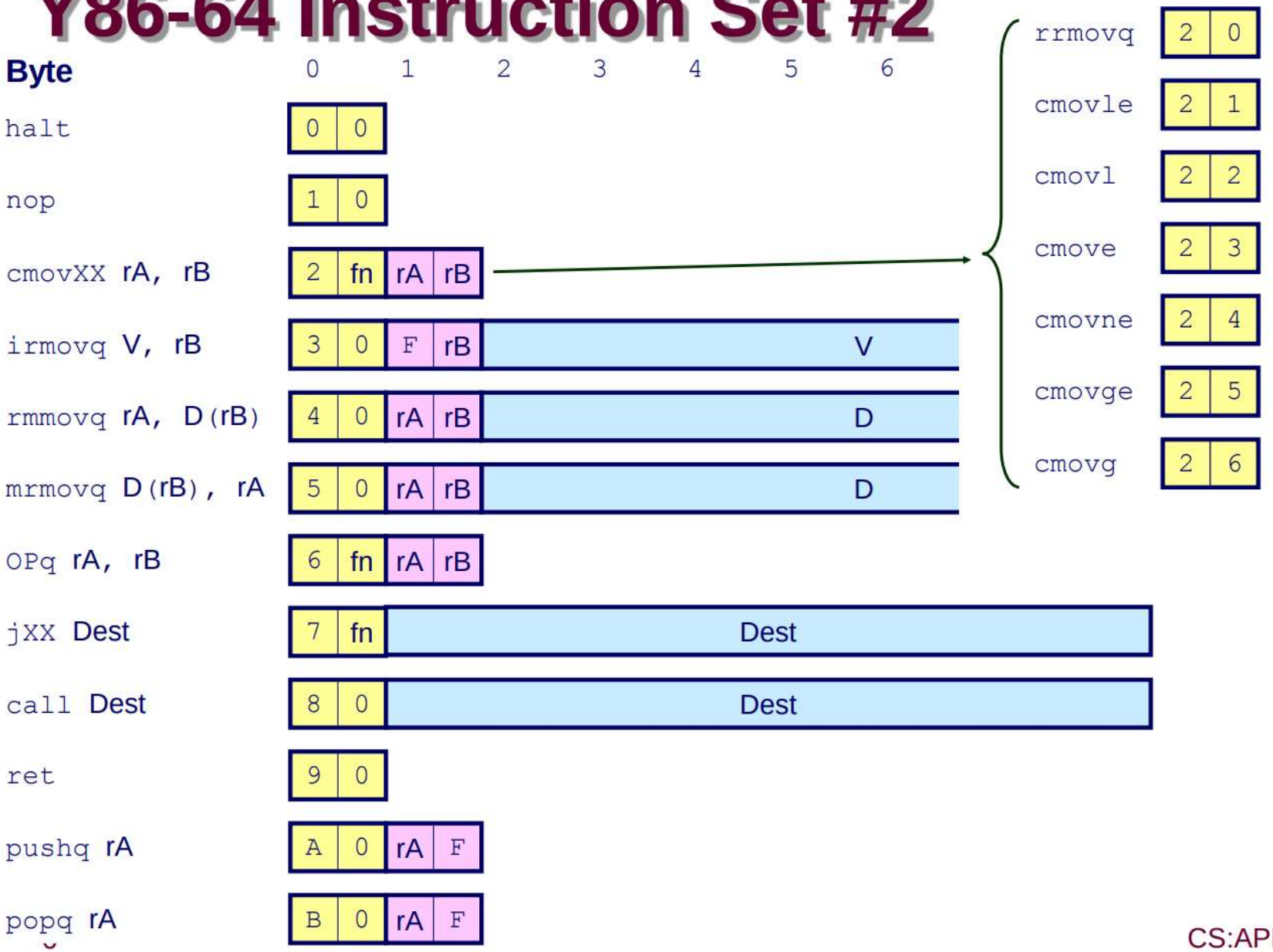
Y86-64 Instruction Set #1



CS:APP3e

ITU CPH

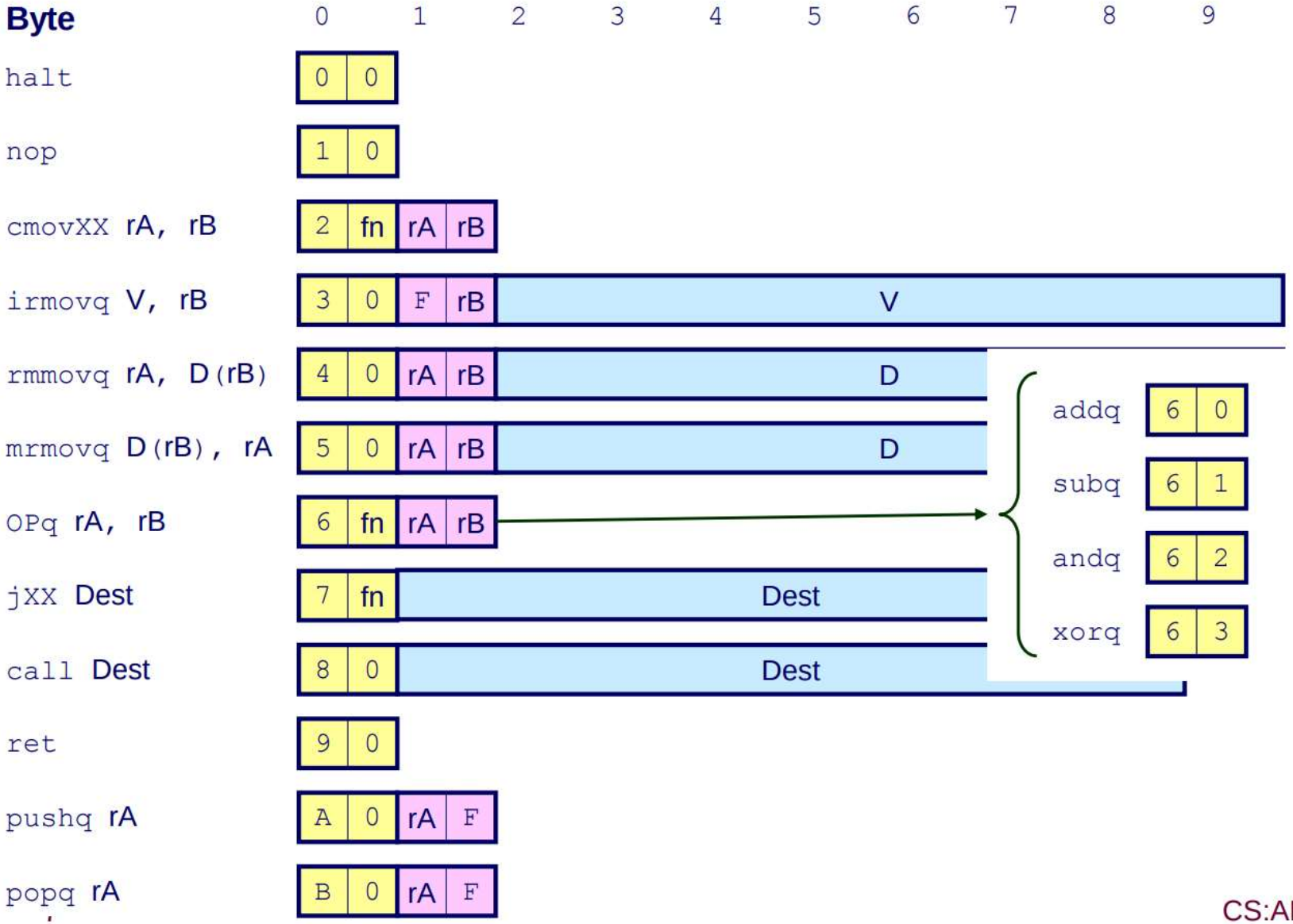
Y86-64 Instruction Set #2



CS:APP3e

ITU CPH

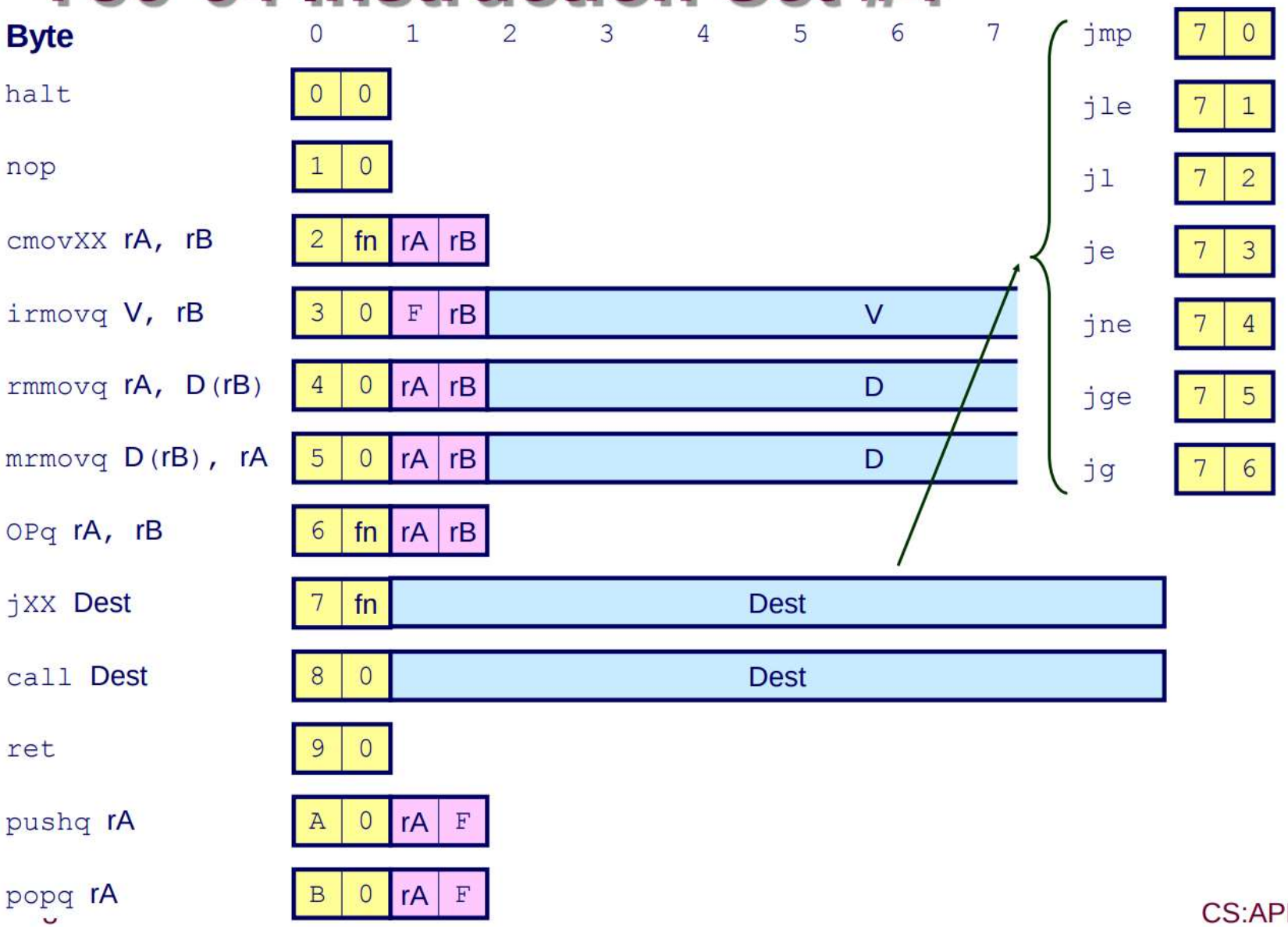
Y86-64 Instruction Set #3



CS:APP3e

ITU CPH

Y86-64 Instruction Set #4



CS:APP3e

ITU CPH

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

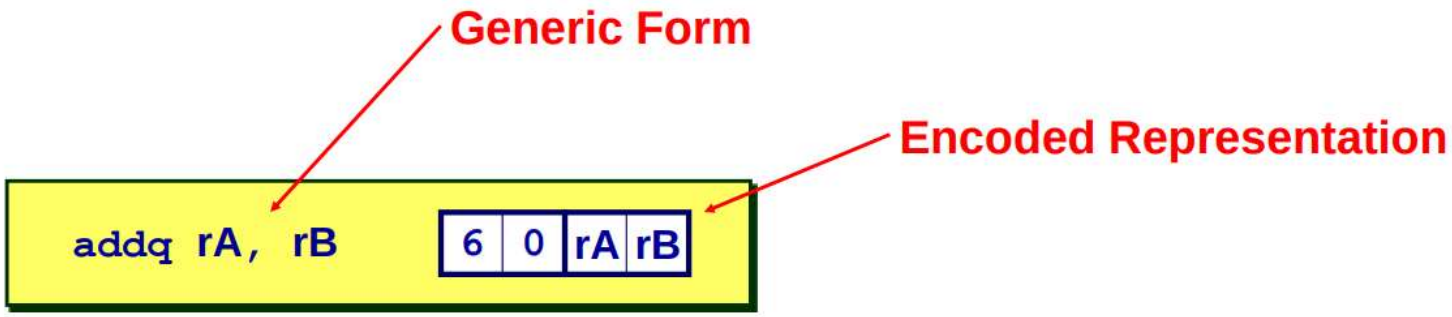
- Same encoding as in x86-64

Register ID 15 (0xF) indicates “no register”

- Will use this in our hardware design in multiple places

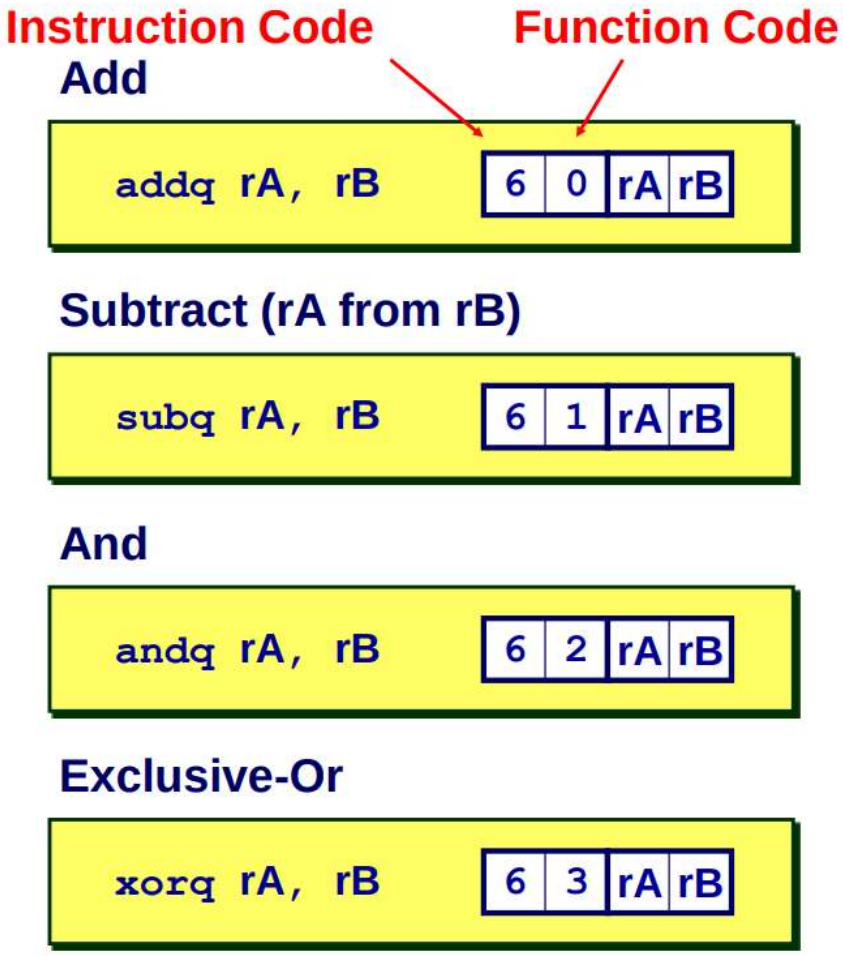
Instruction Example

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



- 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

Immediate → Register

`irmovq` V, rB

Register → Memory

`rmmovq` rA, D(rB)

Memory → Register

`mrmovq` D(rB), rA

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

Conditional Move Instructions

Move Unconditionally

`rrmovq` rA, rB | 2 | 0 | rA | rB

Move When Less or Equal

`cmovle` rA, rB | 2 | 1 | rA | rB

Move When Less

`cmovl` rA, rB | 2 | 2 | rA | rB

Move When Equal

`cmove` rA, rB | 2 | 3 | rA | rB

Move When Not Equal

`cmovne` rA, rB | 2 | 4 | rA | rB

Move When Greater or Equal

`cmovge` rA, rB | 2 | 5 | rA | rB

Move When Greater

`cmovg` rA, rB | 2 | 6 | rA | rB

- 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

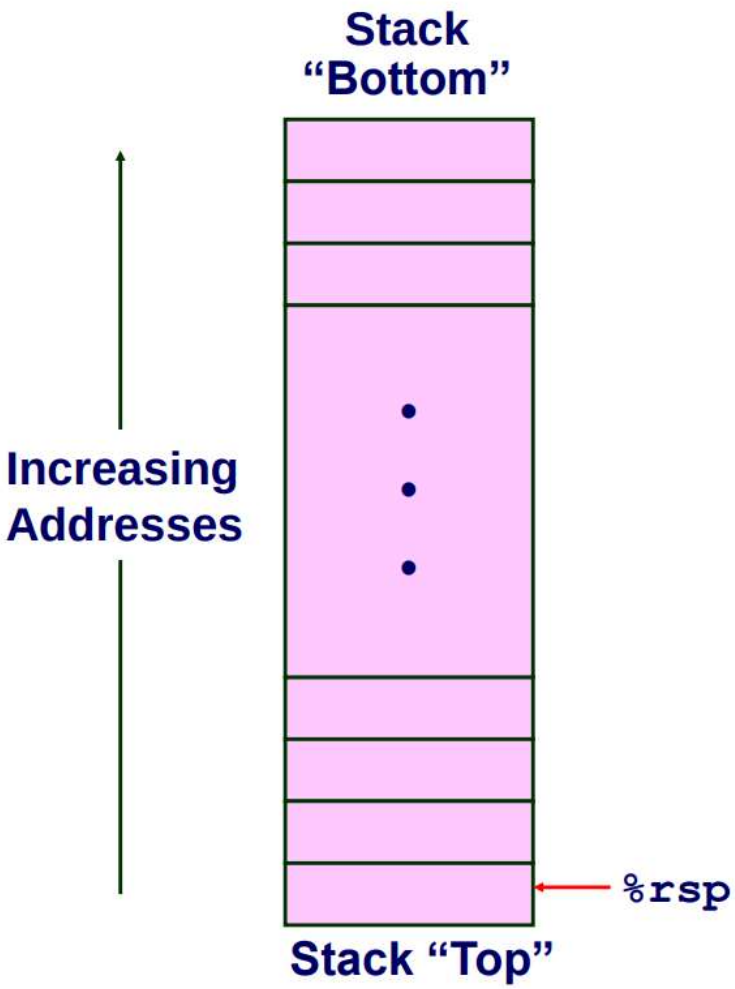
Jump When Equal

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
  - 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

`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

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

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

Summary

Y86-64 Instruction Set Architecture

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

How Important is ISA Design?

- Less now than before
  - With enough hardware, can make almost anything go fast

A Processor: Y86-64 Semantics (SEQ)

ITU CPH

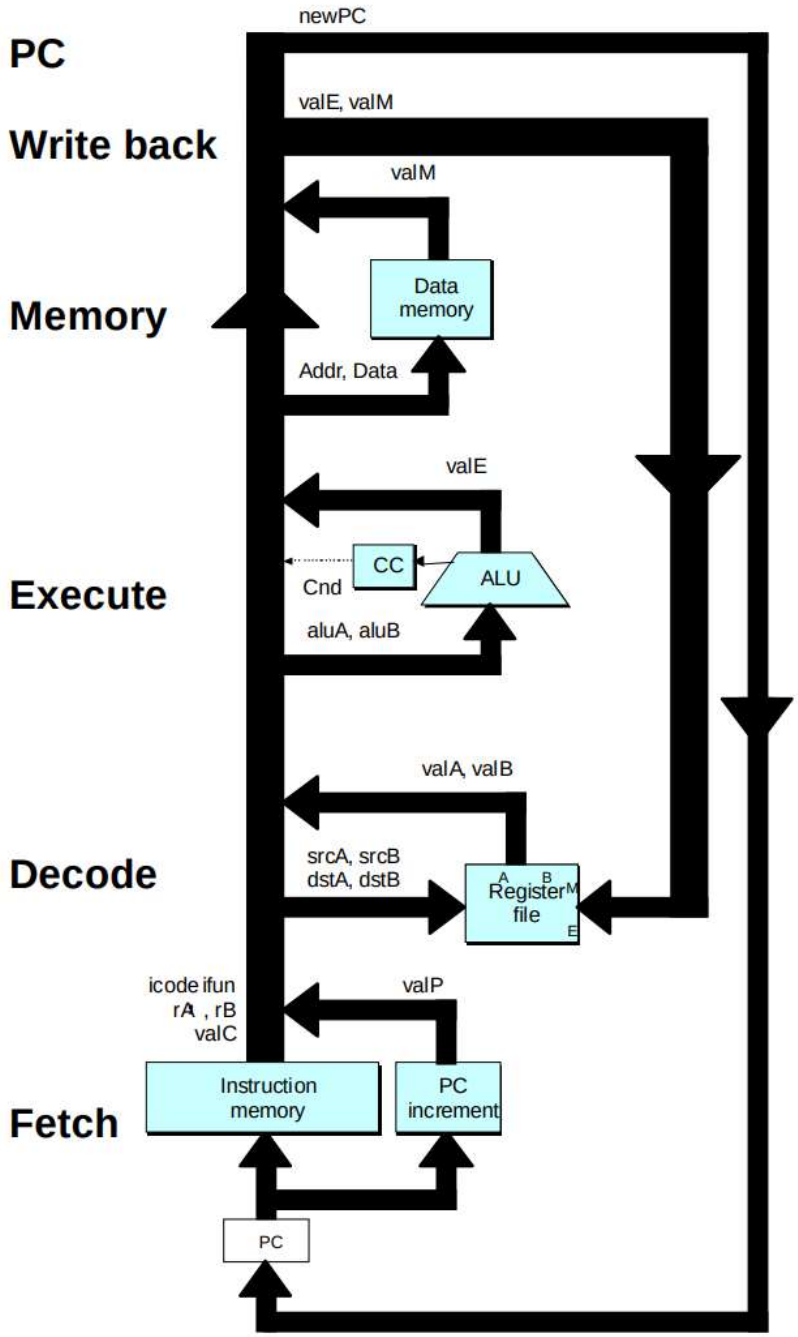
SEQ State & Flow

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



ITU CPH

SEQ Stages

Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

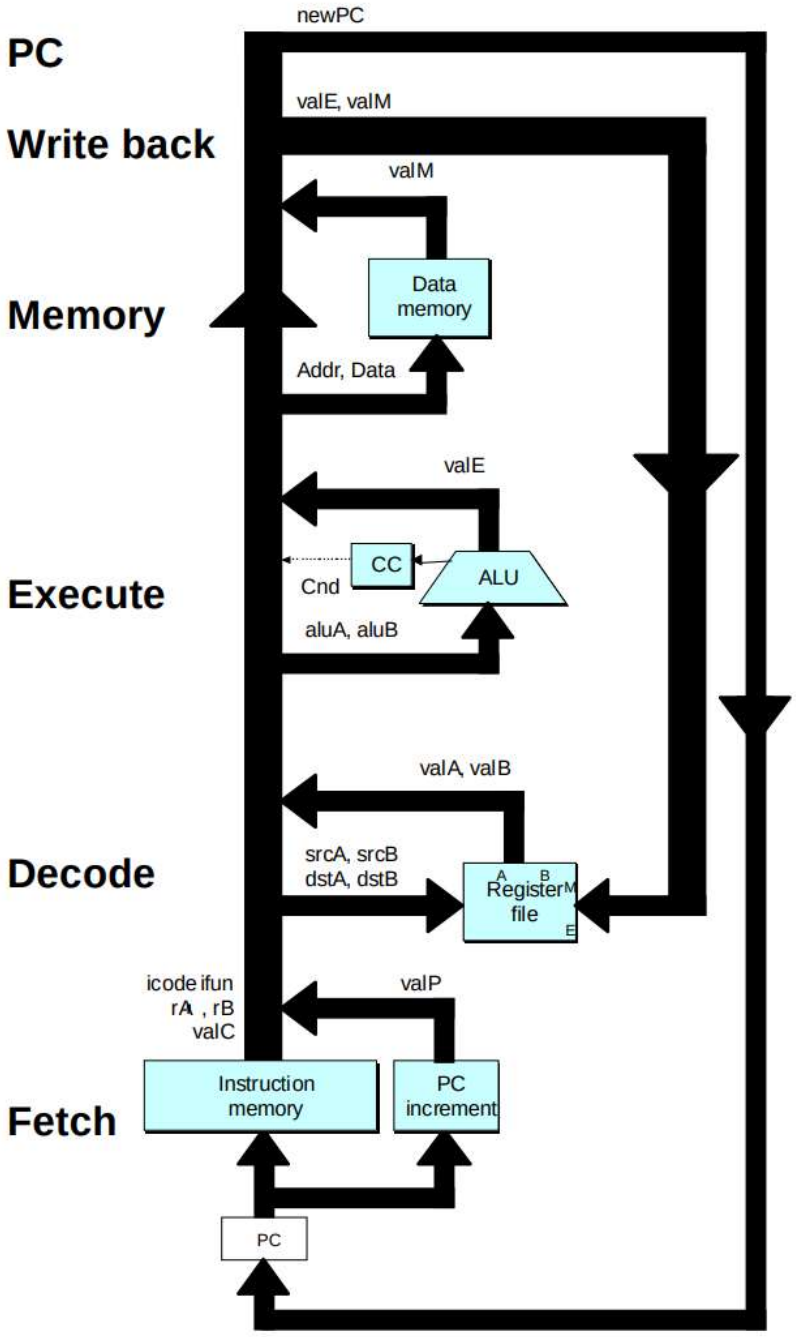
- Read or write data

Write Back

- Write program registers

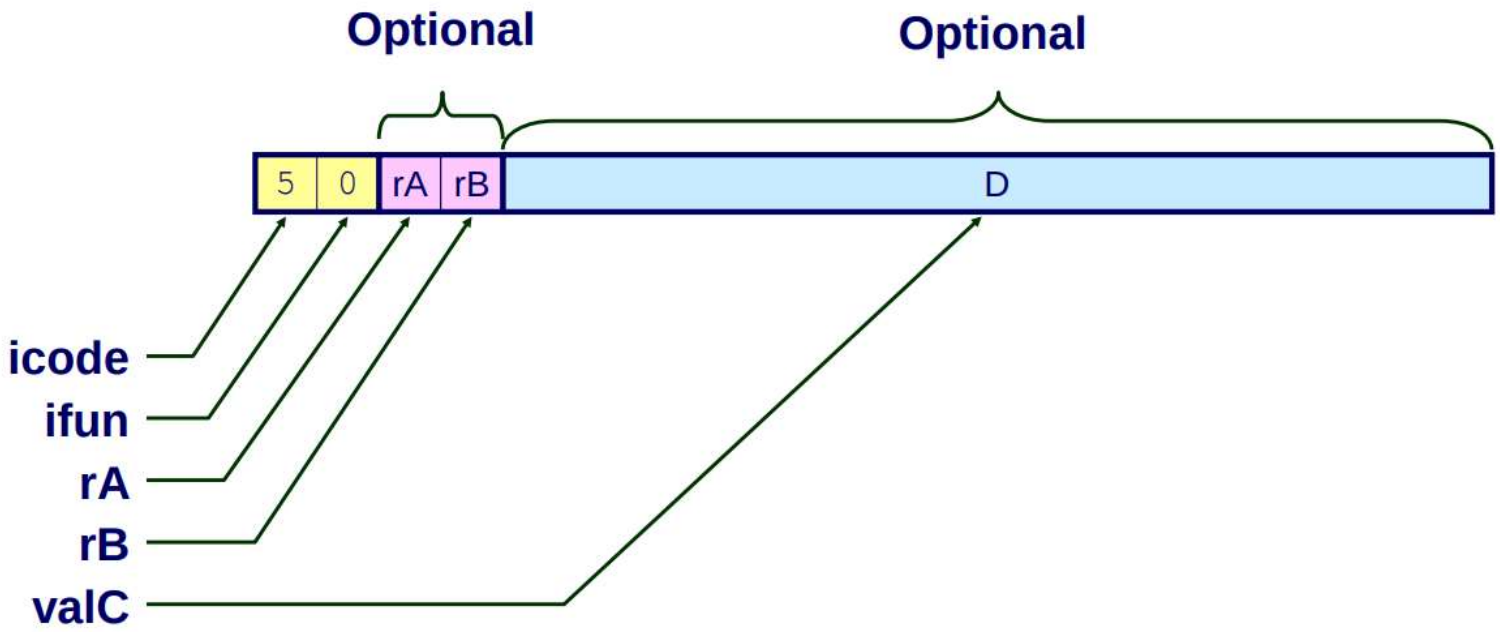
PC

- Update program counter



ITU CPH

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	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 ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation 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] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[%rsp] valB ← R[%rsp]	Read stack pointer Read stack pointer
Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read from stack
Write back	R[%rsp] ← valE R[rA] ← valM	Update stack pointer 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

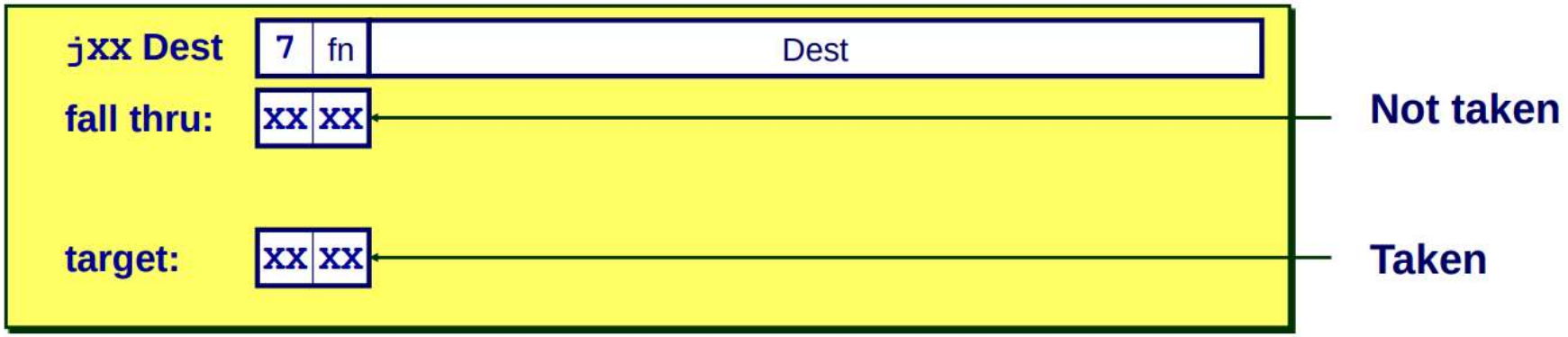
- Increment PC by 2

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

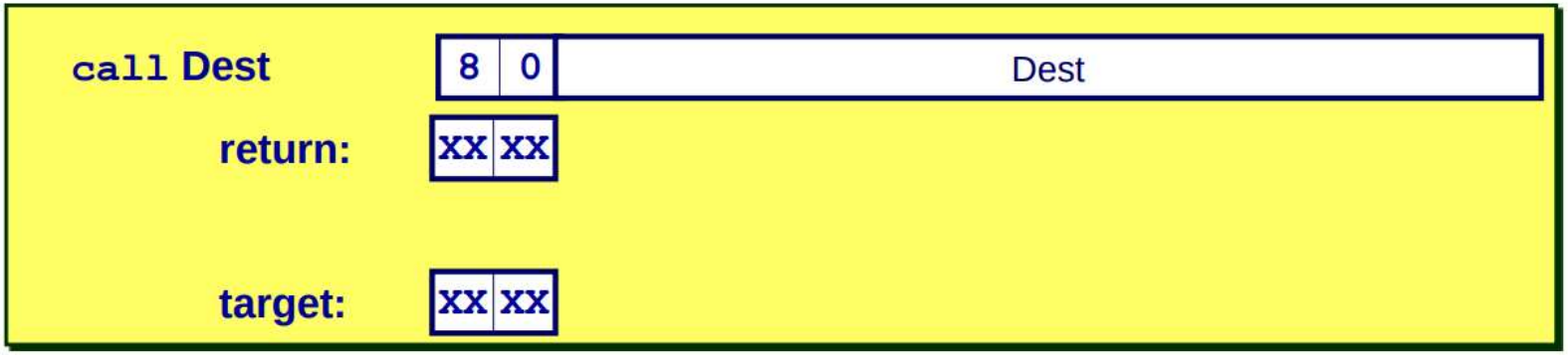
- Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_8[PC+1]	Read destination address
	valP ← PC+9	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

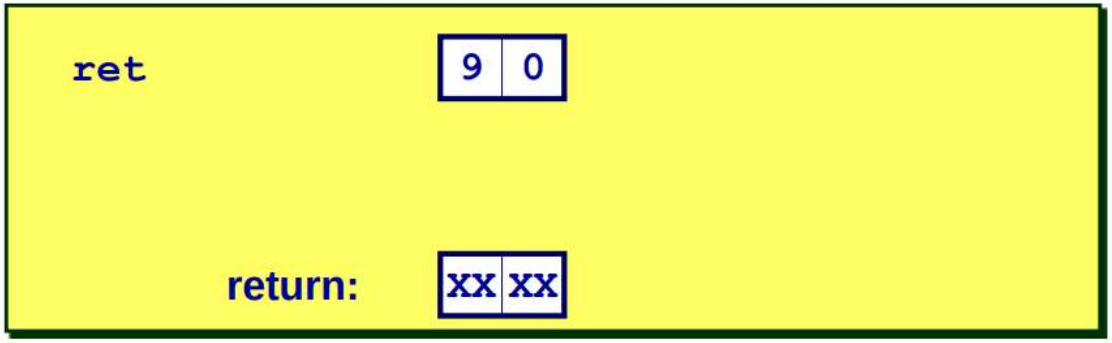
- Set PC to Dest

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

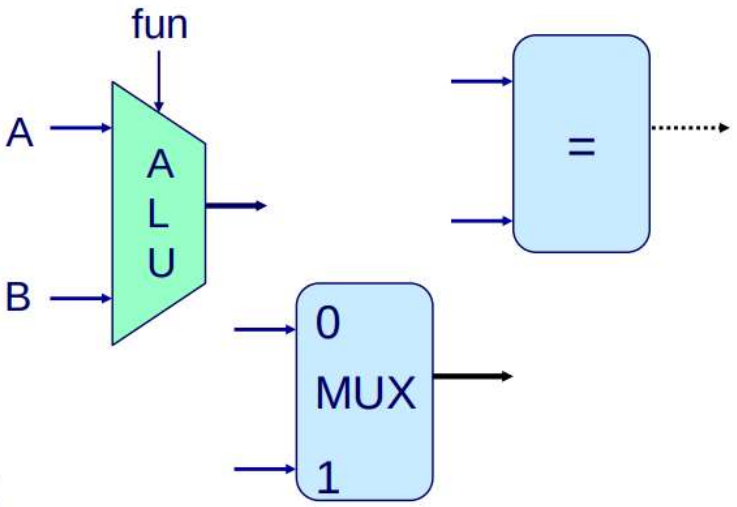
A Processor: Y86-64 Implementation (SEQ)

ITU CPH

Building Blocks

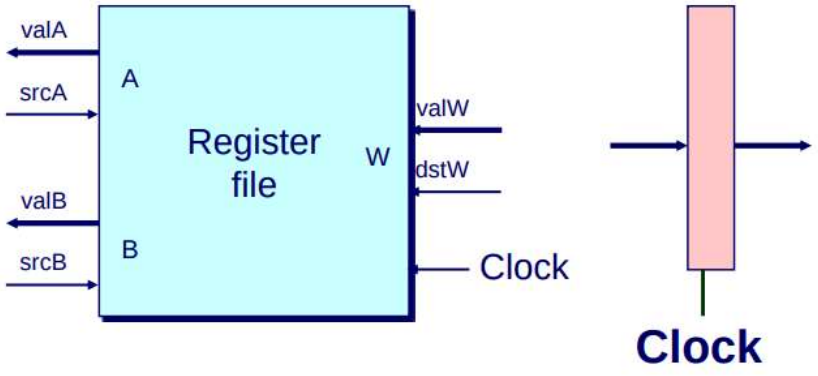
Combinational Logic

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



Storage Elements

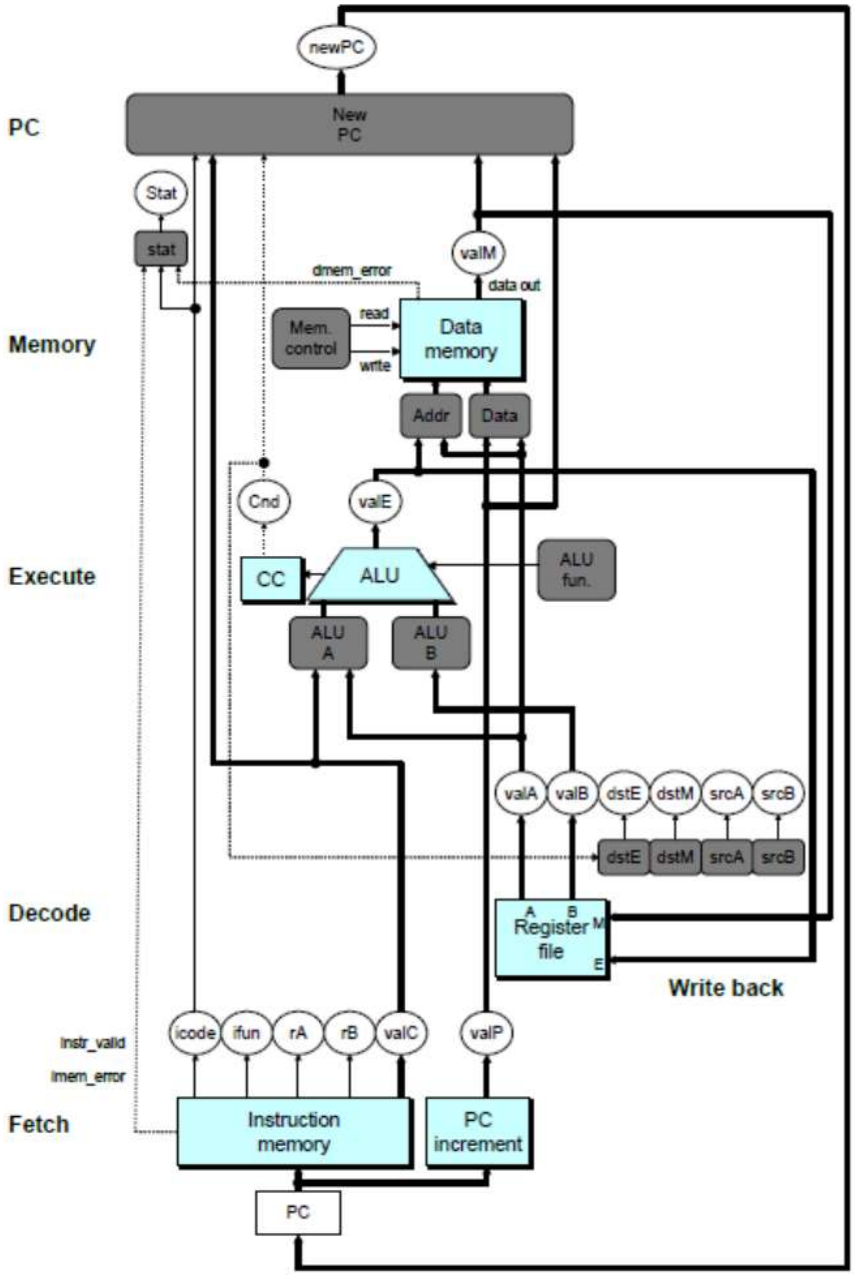
- Store bits
- Addressable memories
- Non-addressable registers
- Loaded only as clock rises



SEQ Hardware

Key

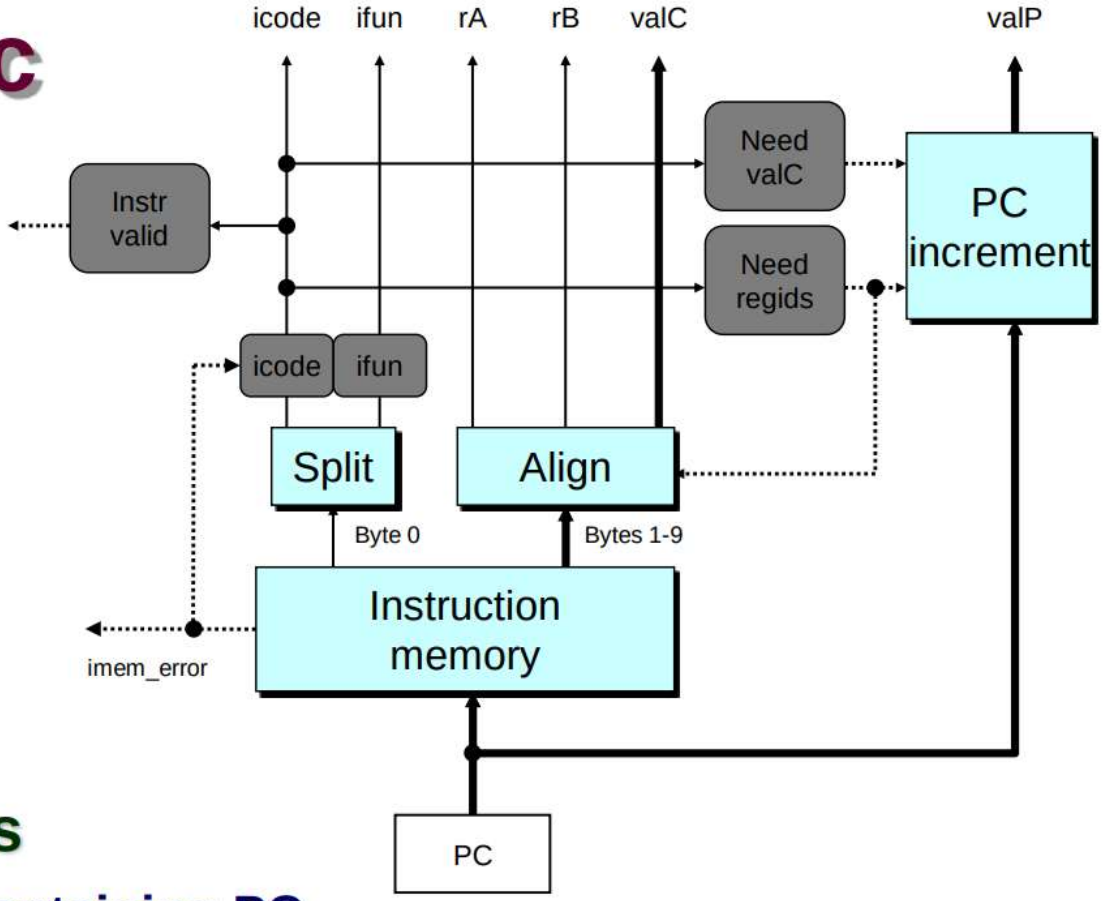
- Blue boxes: predesigned hardware blocks
  - E.g., memories, ALU
- Gray boxes: control logic
  - Describe in HCL
- White ovals: labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



CS:APP3e

ITU CPH

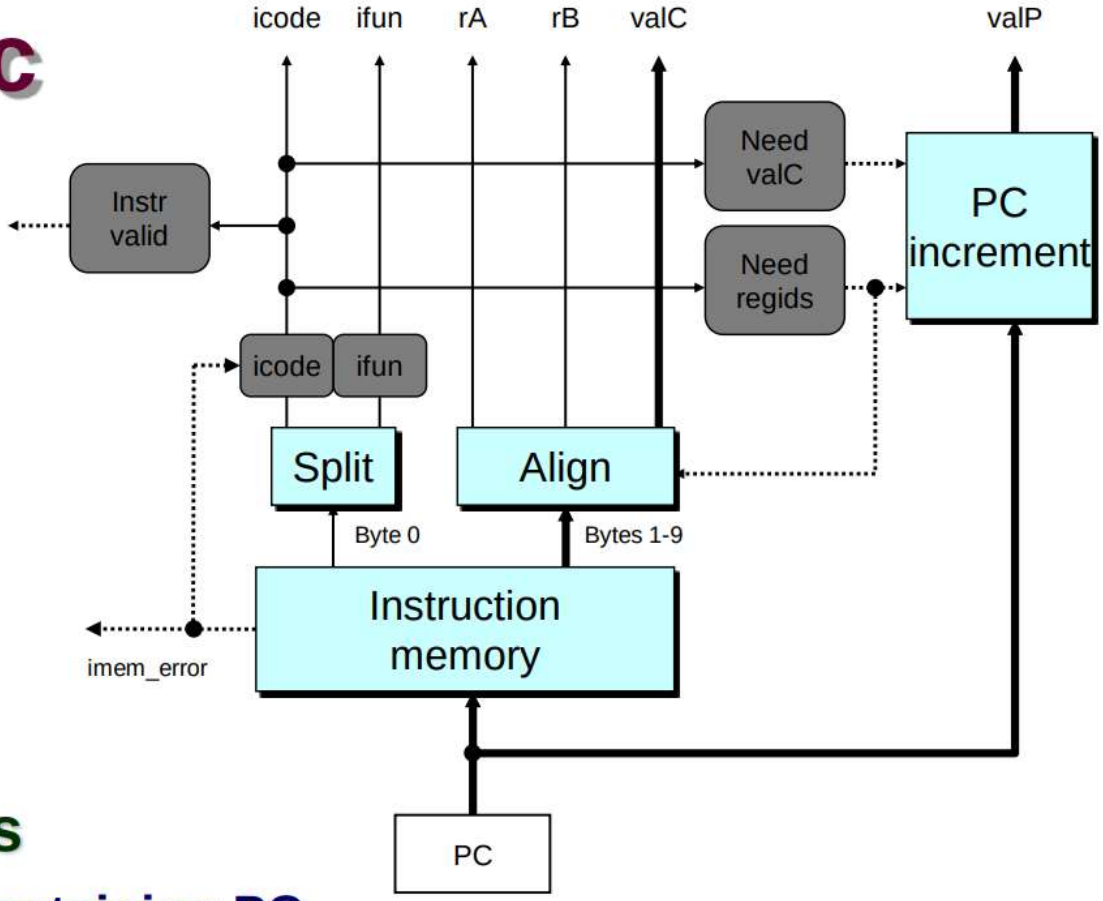
Fetch Logic



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

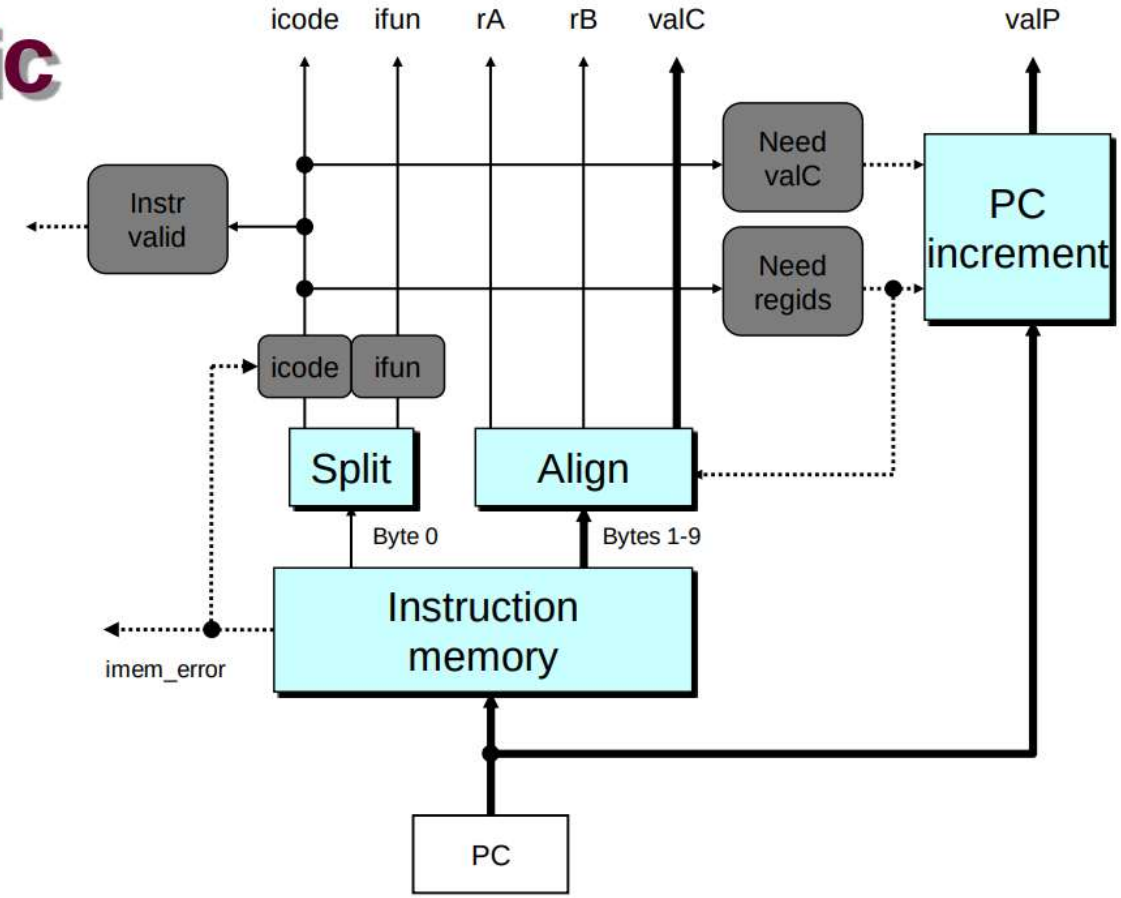
Fetch Logic



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?

Decode Logic

Register File

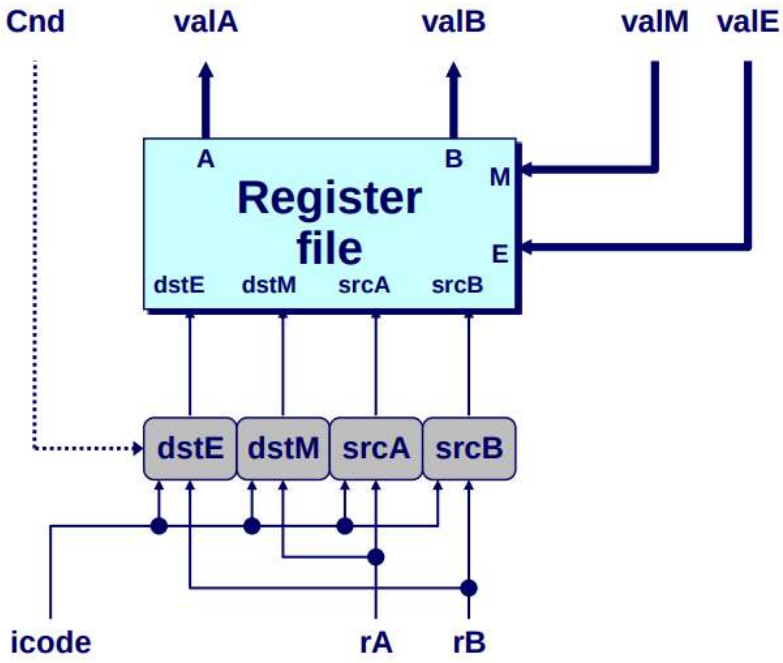
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

Control Logic

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

Signals

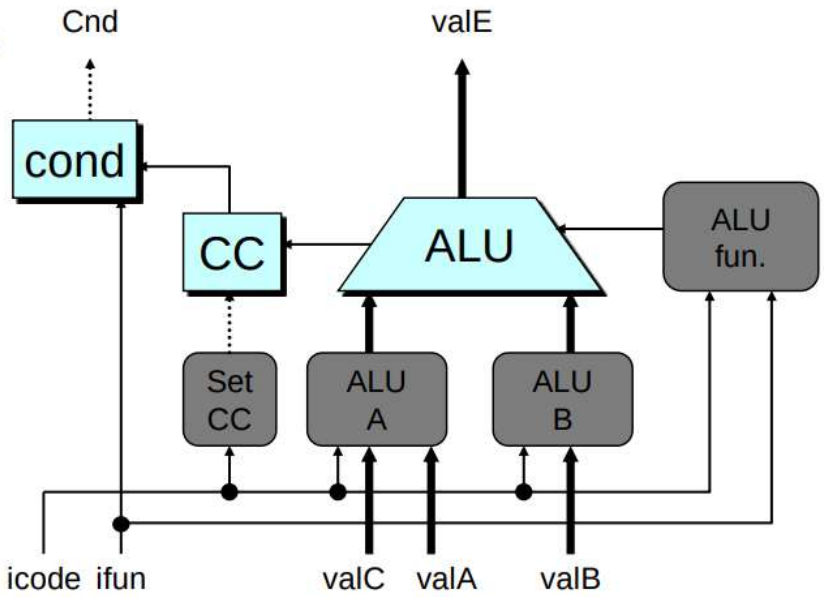
- Cnd: Indicate whether or not to perform conditional move
  - Computed in Execute stage



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?

CS:APP3e

ITU CPH

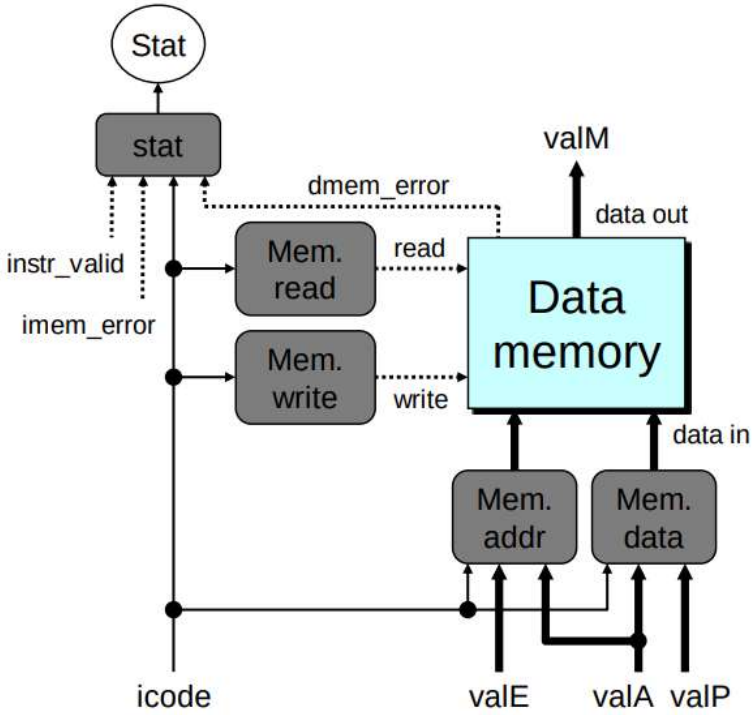
Memory Logic

Memory

- Reads or writes memory word

Control Logic

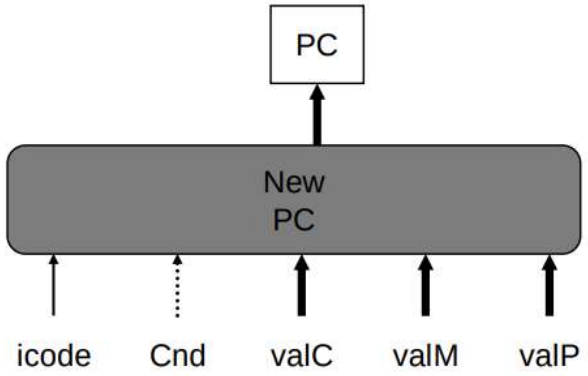
- stat: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



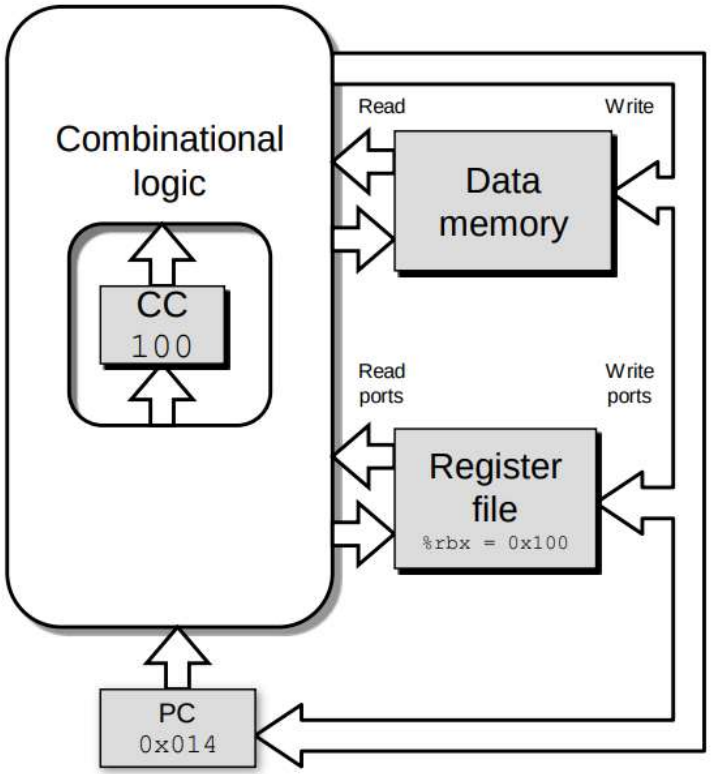
PC Update Logic

New PC

- Select next value of PC



SEQ Operation



State

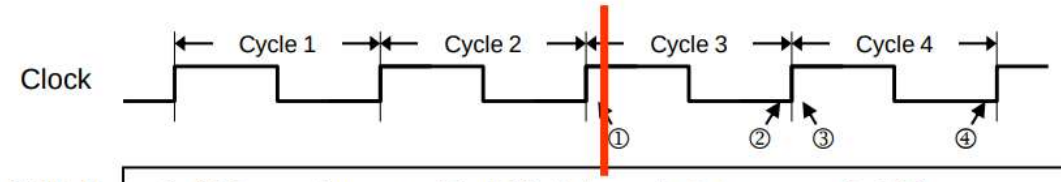
- PC register
- Cond. Code register
- Data memory
- Register file

All updated as clock rises

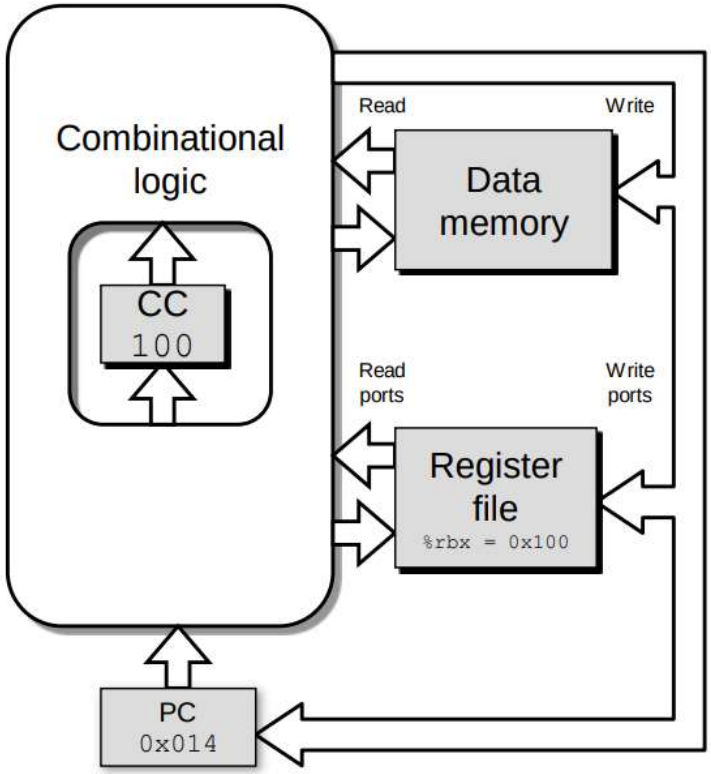
Combinational Logic

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

SEQ Operation #2

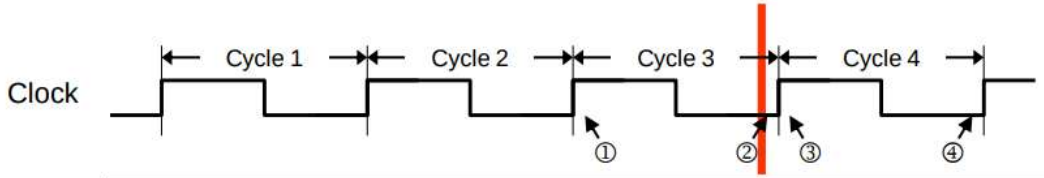


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

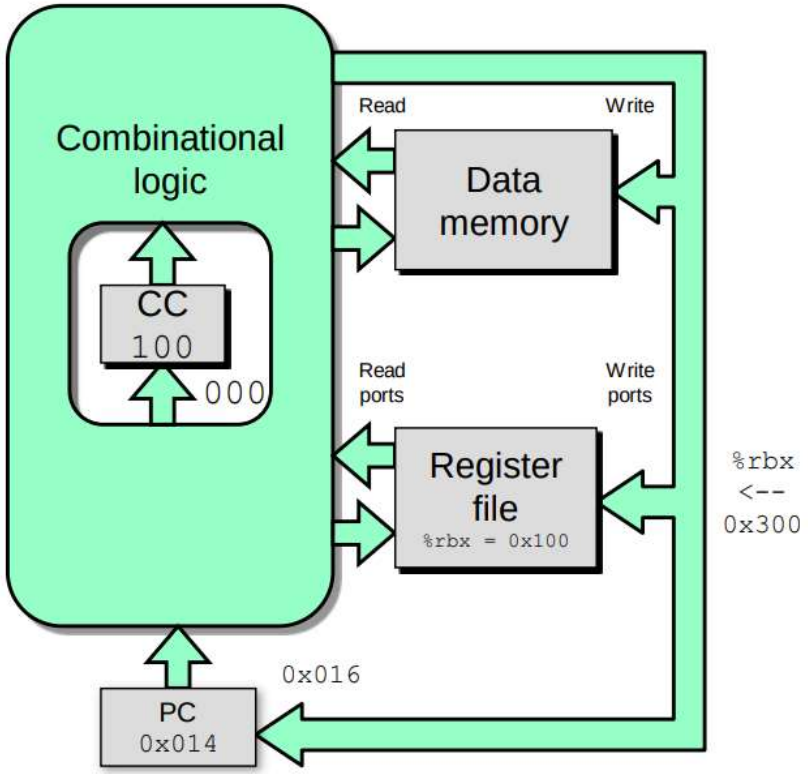


- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

SEQ Operation #3

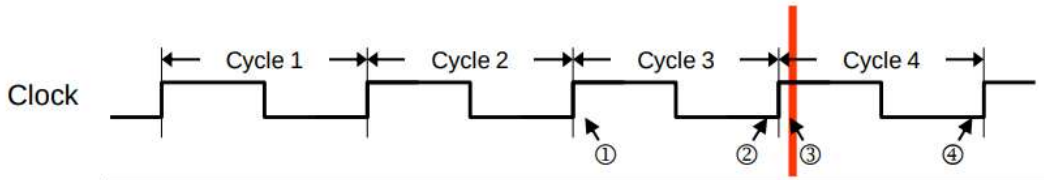


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

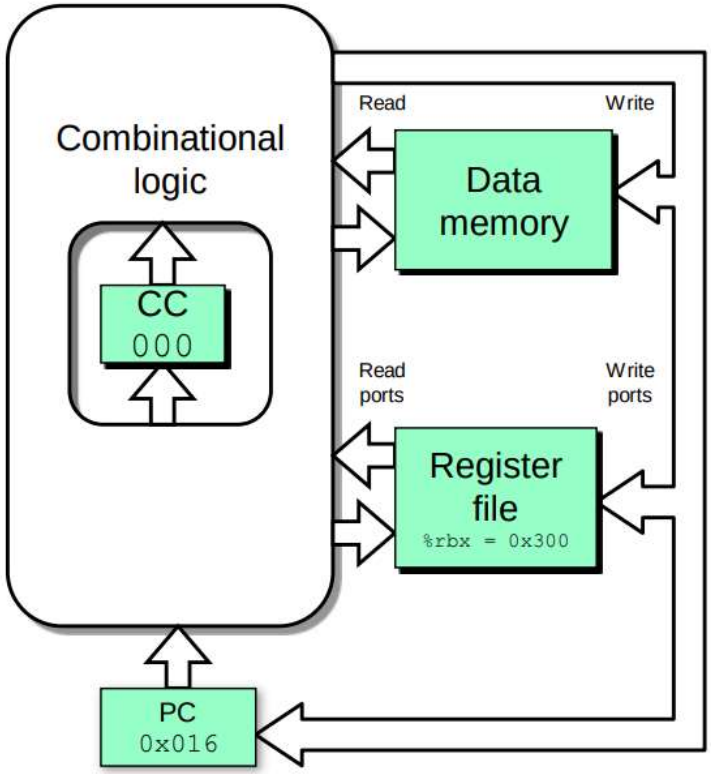


- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction

SEQ Operation #4

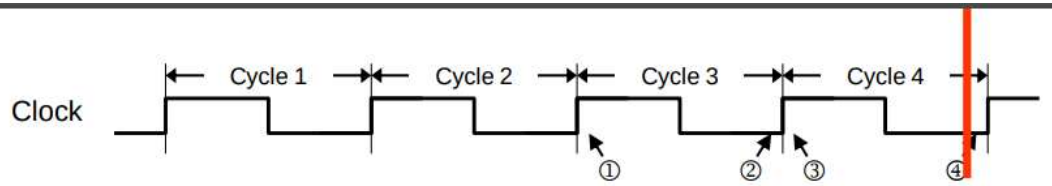


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

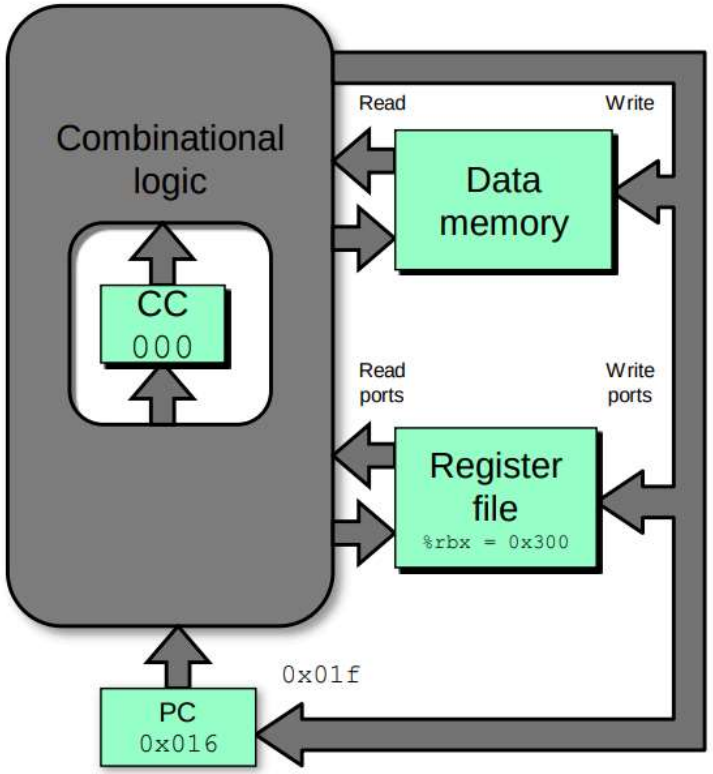


- state set according to `addq` instruction
- combinational logic starting to react to state changes

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

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

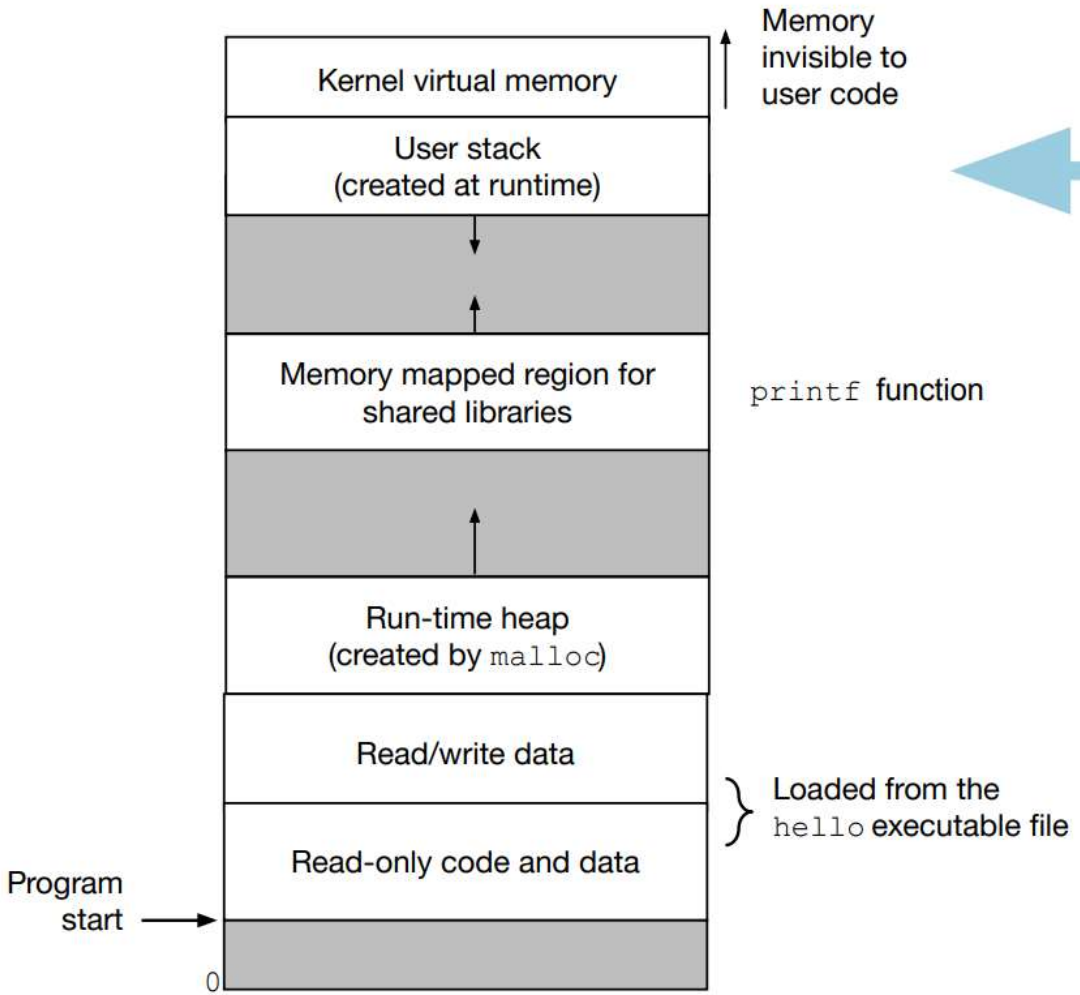
- Too slow to be practical
- In one cycle, must propagate through instruction memory, register file, ALU, and data memory
- Would need to run clock very slowly
- Hardware units only active for fraction of clock cycle

Stack, Procedure Calls

ITU CPH

Virtual Memory

What is the stack?



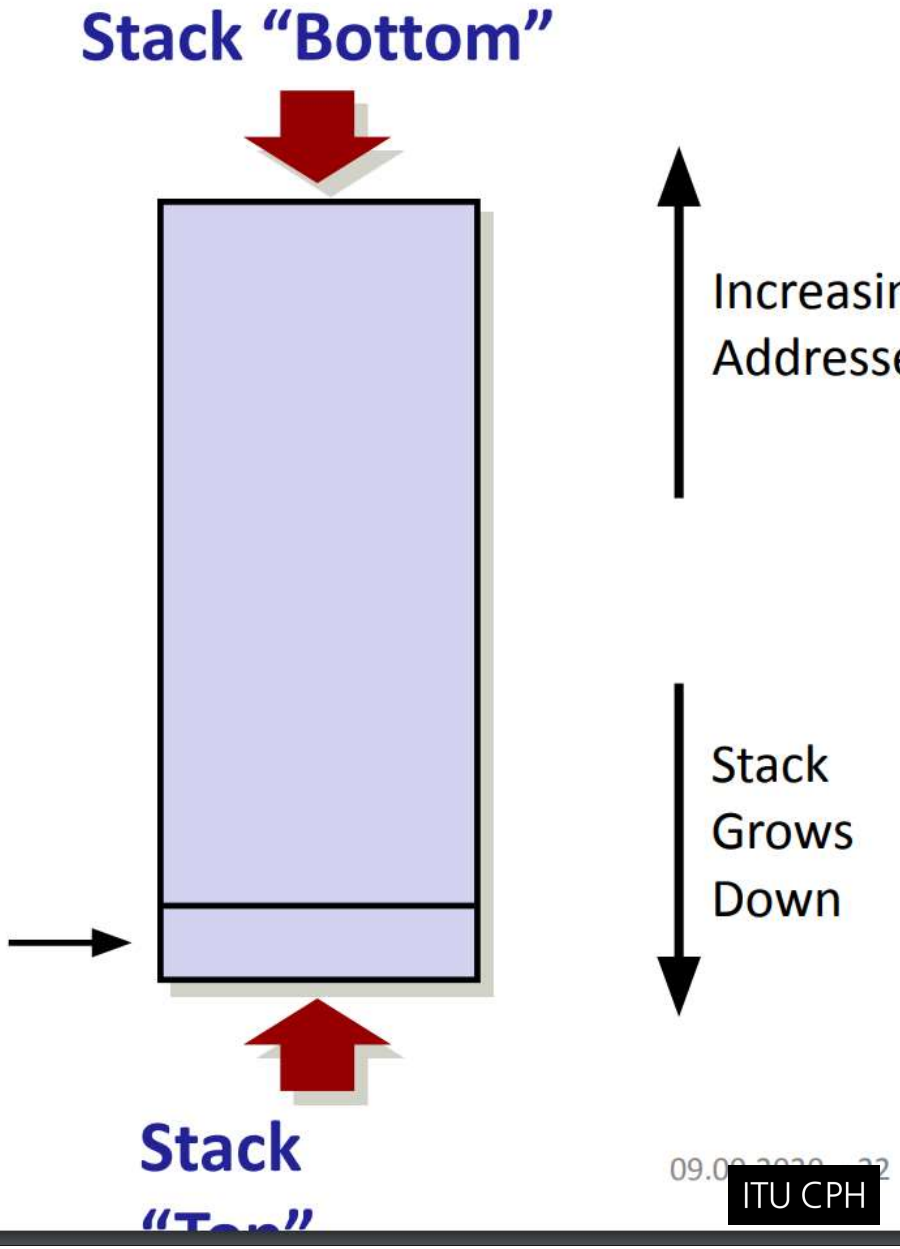
UNIVERSITY OF COPENHAGEN

09.0[illegible] 21

ITU CPH

The Stack

- Region of memory managed with stack discipline
- Grows towards lower addresses
- Register %rbp points to the bottom of the stack
- Register %rsp points to the top of the stack, it is the **stack pointer**

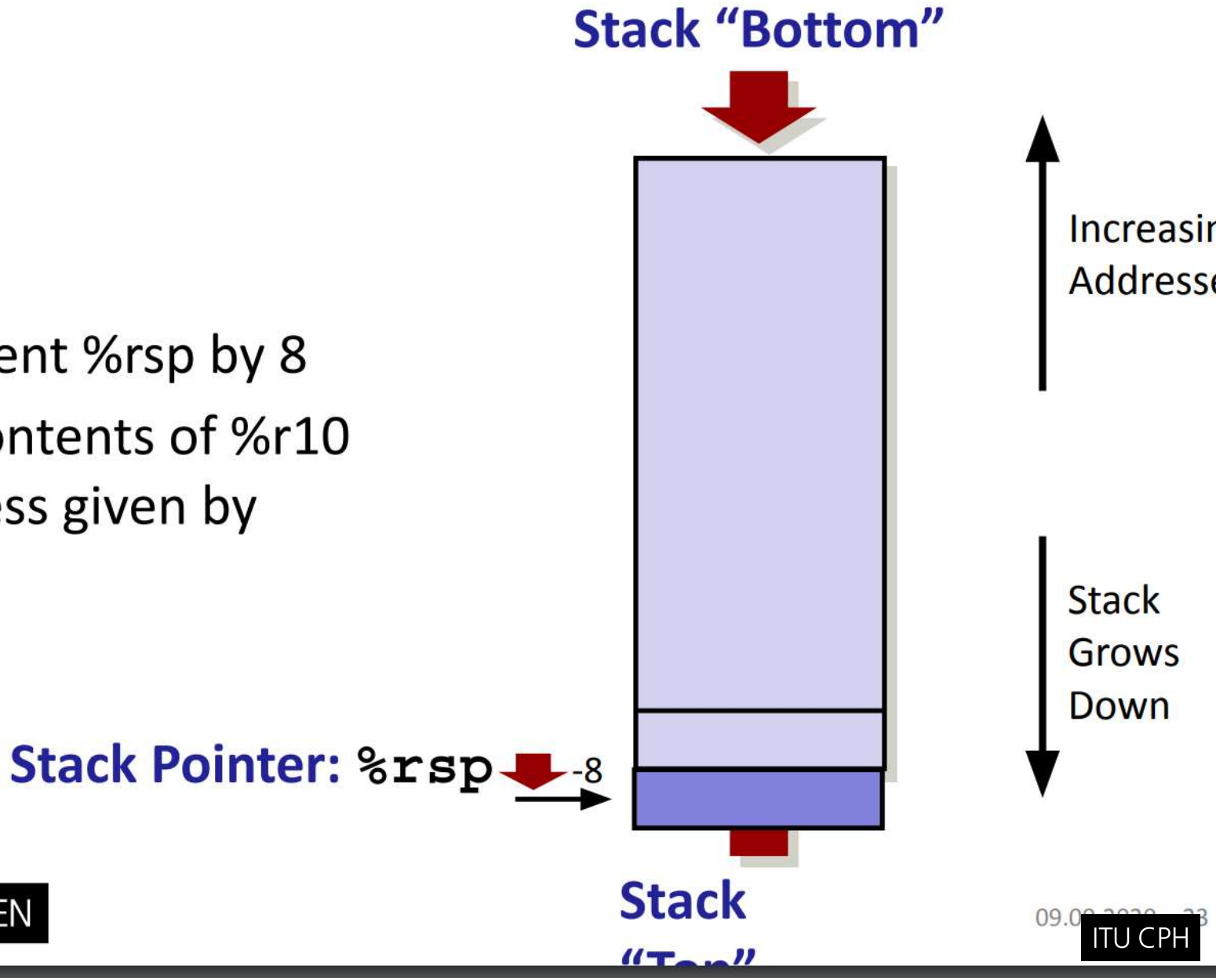


Stack - Push

Operator push

pushq %r10

1. Decrement %rsp by 8
2. Write contents of %r10 at address given by %rsp

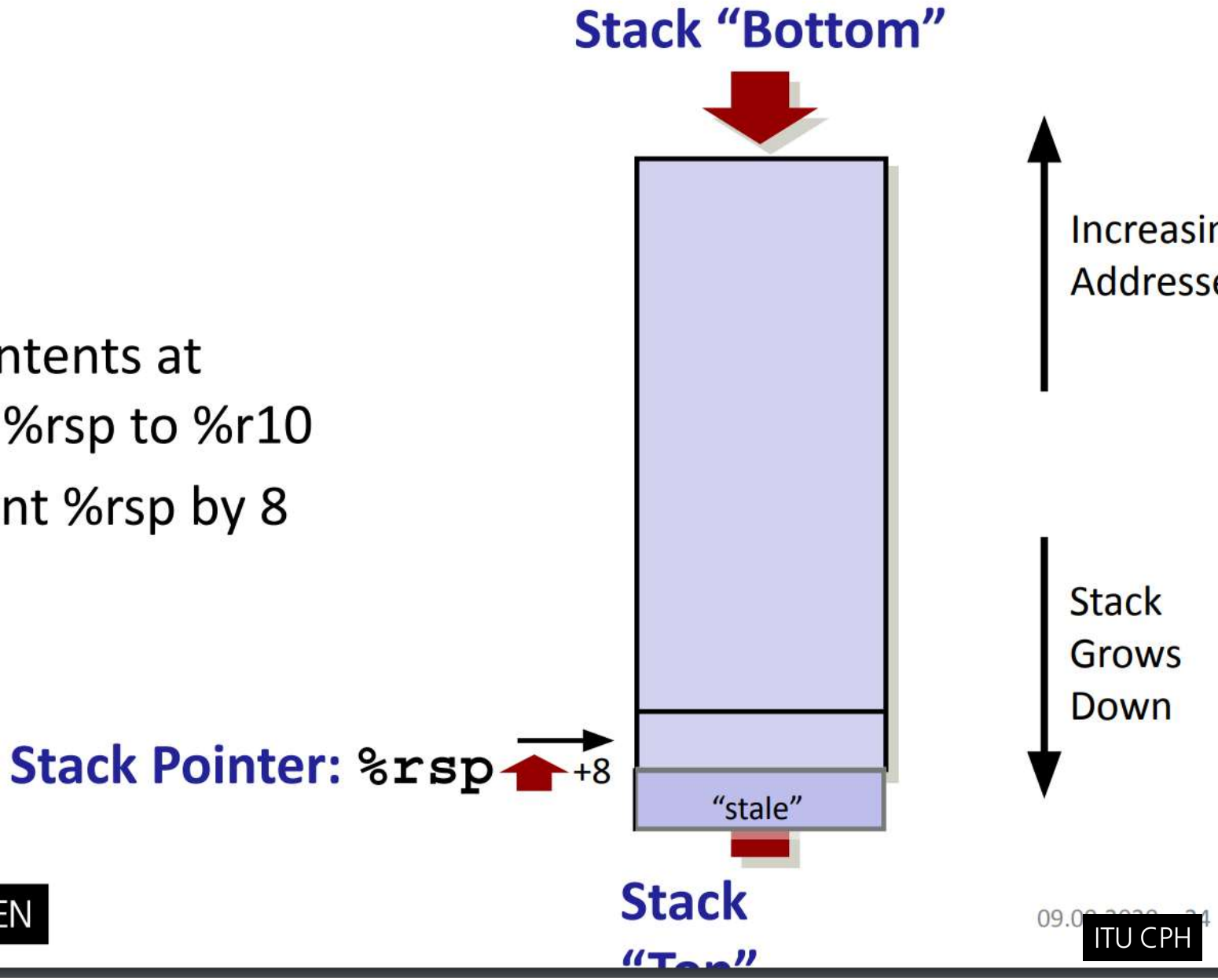


Stack - Pop

Operator pop

popq %r10

1. Copy contents at address %rsp to %r10
2. Increment %rsp by 8



Procedure Call

Instructions:

- **call**: push **return address** on stack; jump to label/address
  - Return address is address of instruction right after call instruction
- **ret**: pop address from stack; jump to address

Example

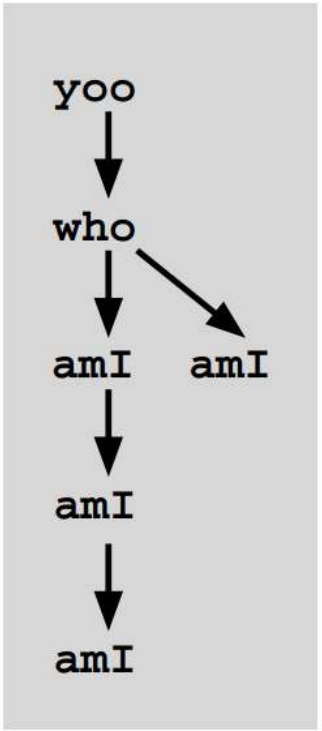
why/how is the stack relevant when talking about procedure calls?

```
yoo(...)
{
   •
   •
   who();
   •
   •
}
```

```
who(...)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(...)
{
   •
   •
   amI();
   •
   •
}
```

Example Call Chain



Procedure `amI()` is recursive

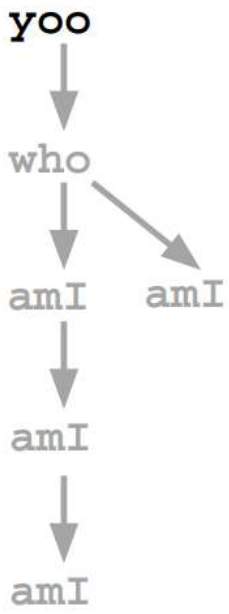
we use the stack to keep track of deep (possibly recursive) procedu

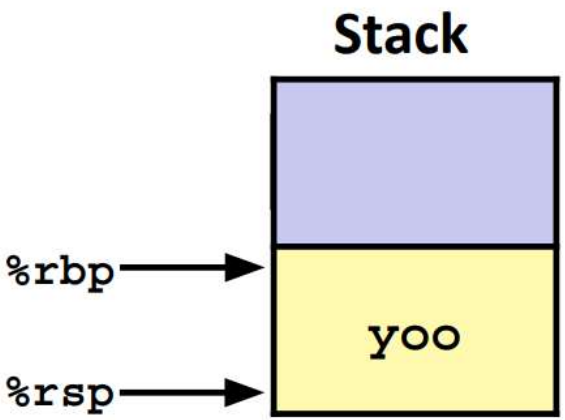
UNIVERSITY OF COPENHAGEN

ITU CPH

Example

```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```



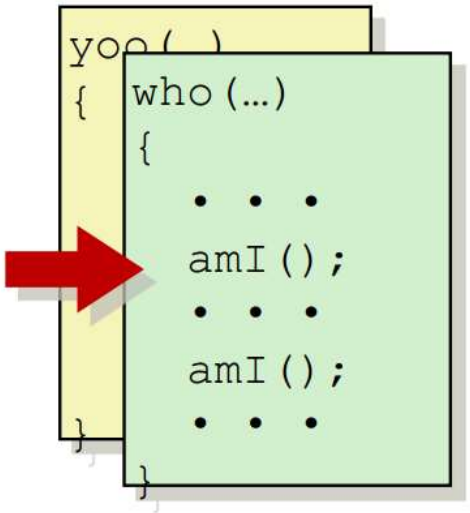


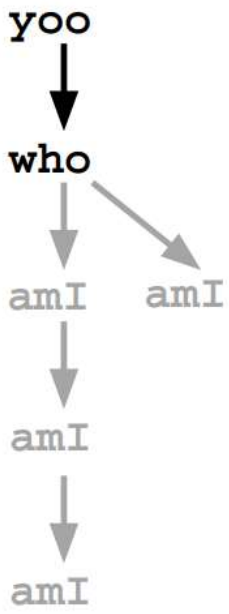
stack frame

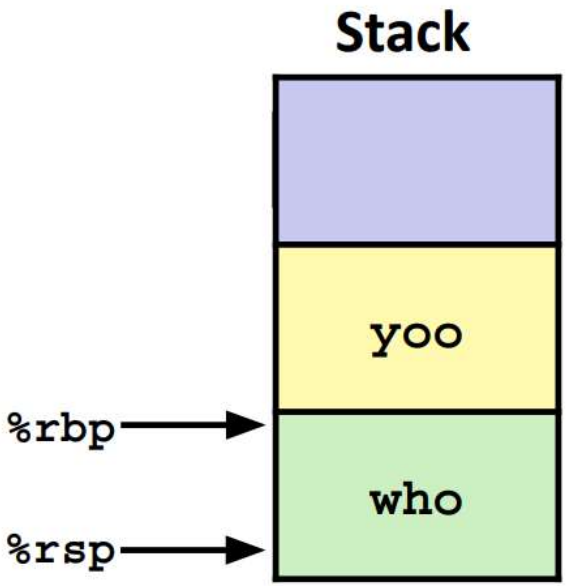
- caller's saved registers[1]
- local vars
- args
- return address[2]

[1] : to restore caller state on return (bottom frame has no caller).
[2] : pushes it when it calls (top frame does not need this)

Example



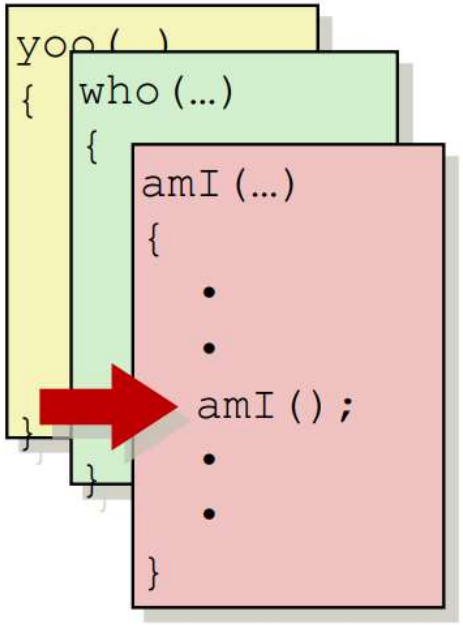


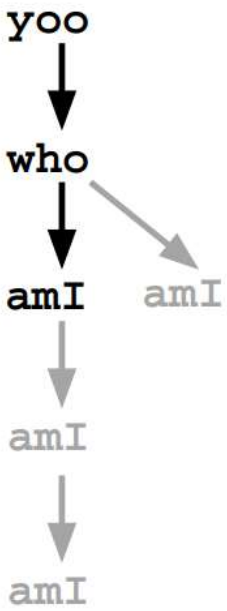


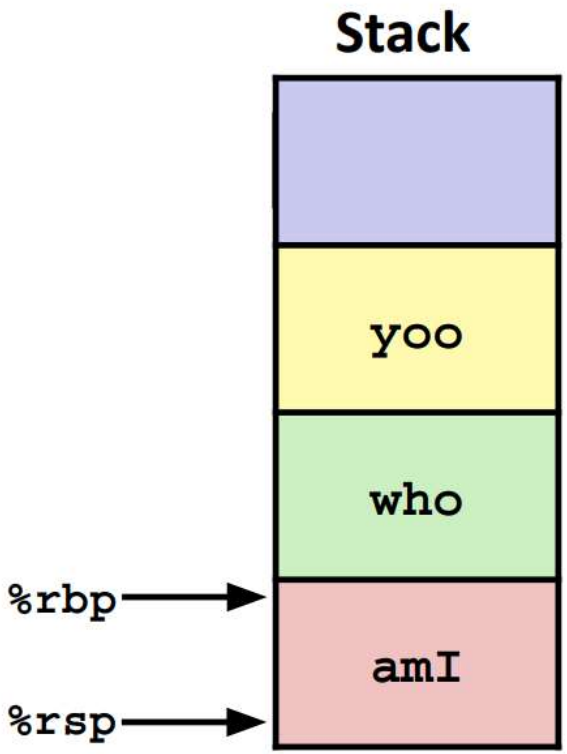
UNIVERSITY OF COPENHAGEN

ITU CPH

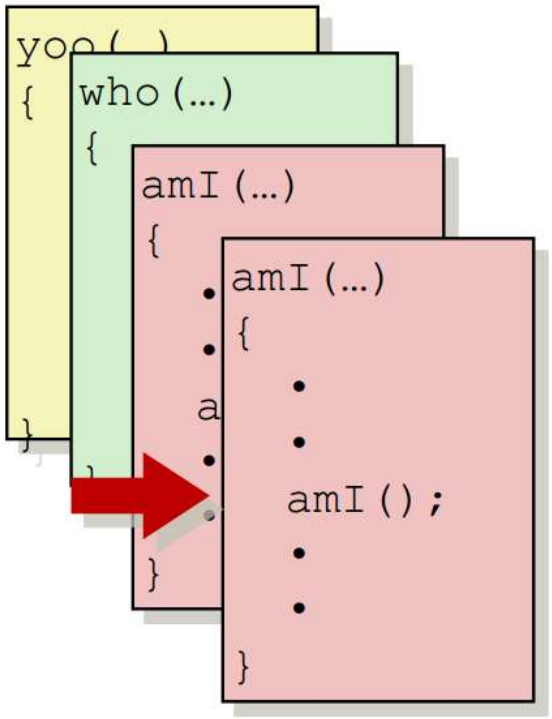
Example

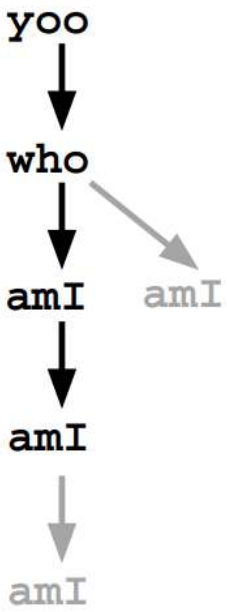

yoo ( )
{
who (...)
{
amI (...)
{
amI ();
}
}
}

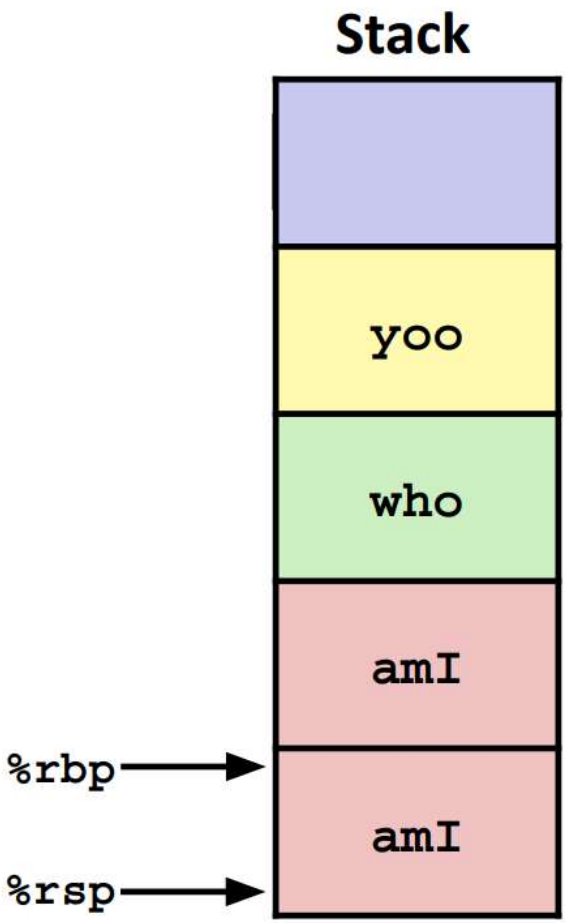

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

Example



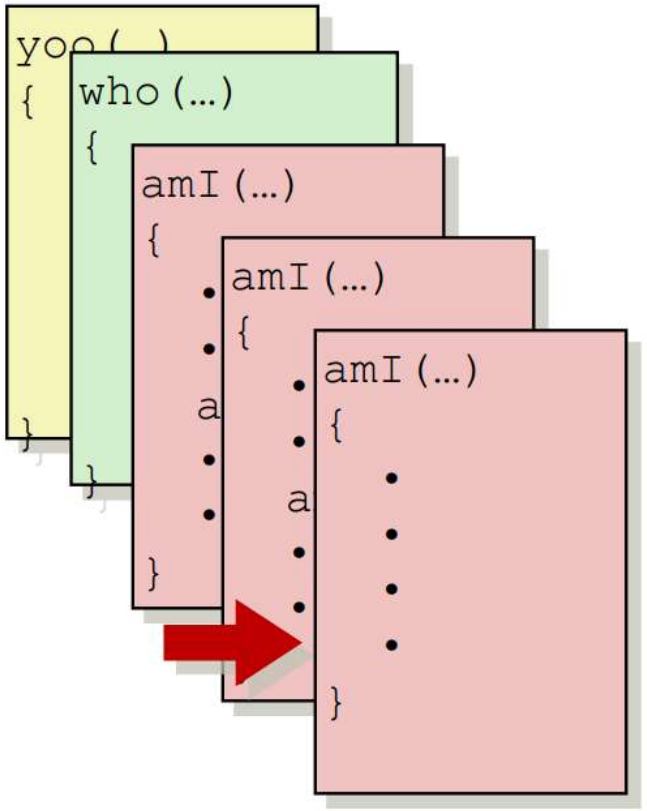


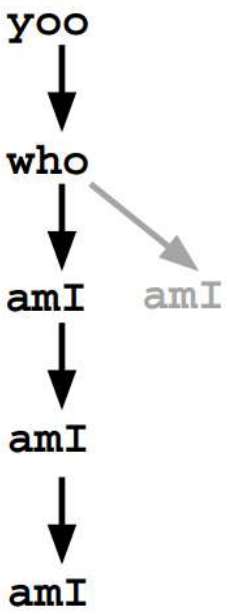


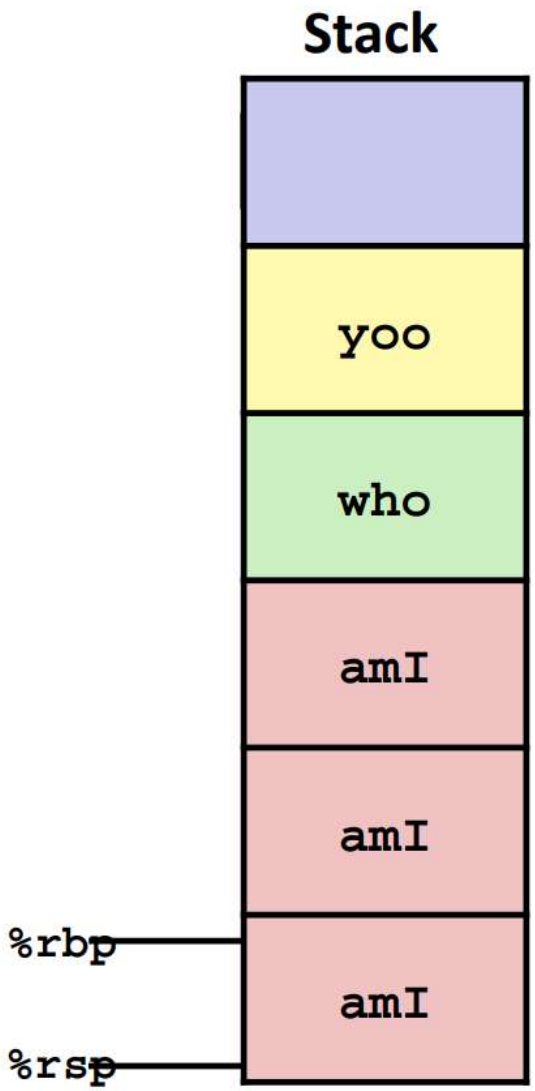
UNIVERSITY OF COPENHAGEN

ITU CPH

Example



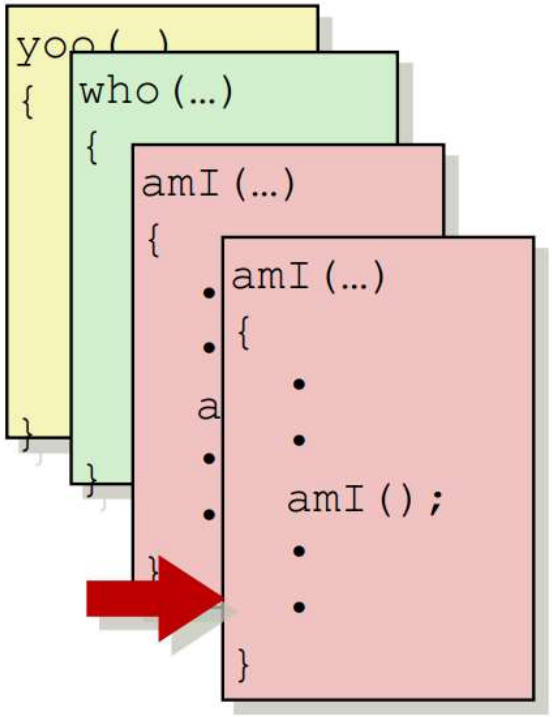


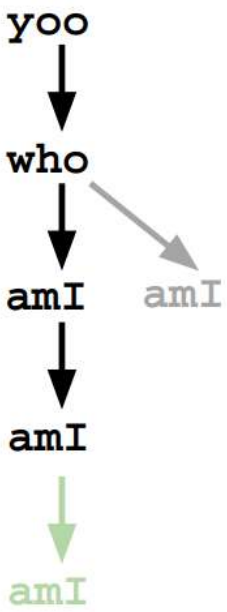


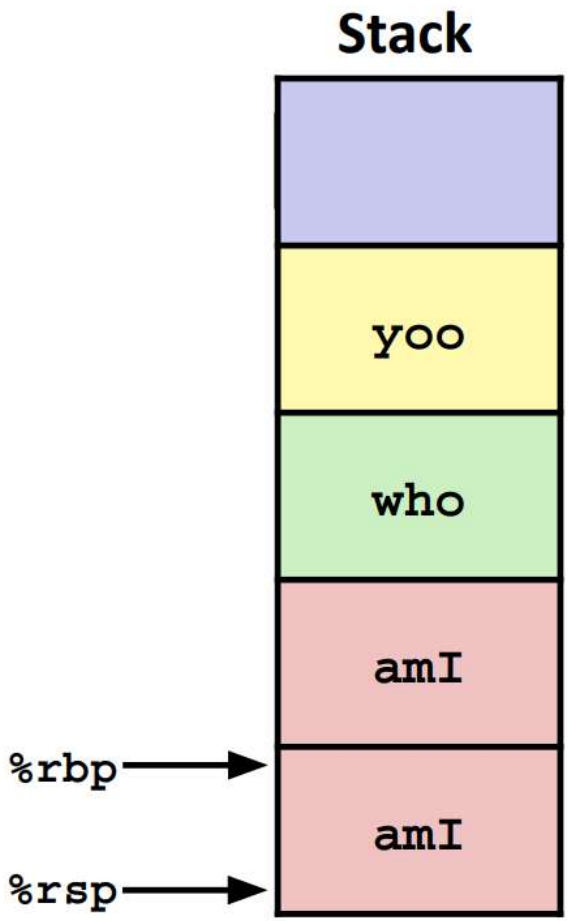
UNIVERSITY OF COPENHAGEN

ITU CPH

Example



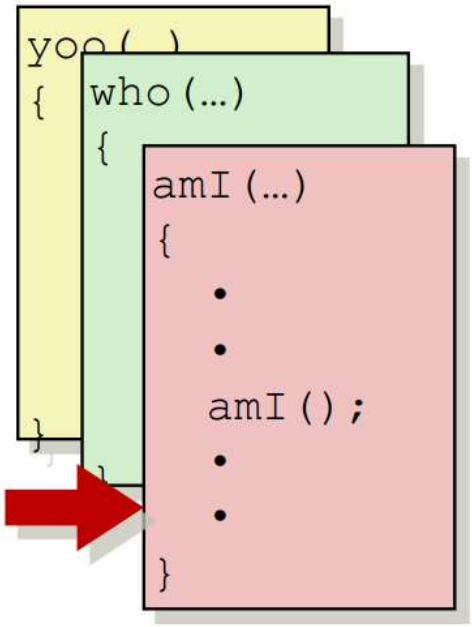


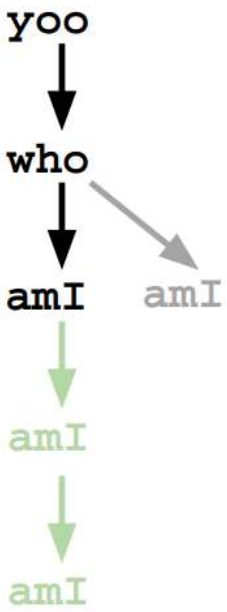


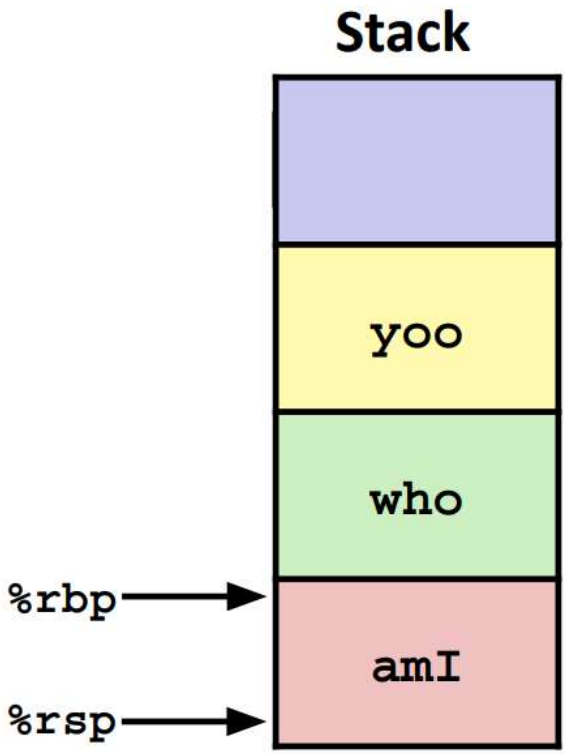
UNIVERSITY OF COPENHAGEN

ITU CPH

Example



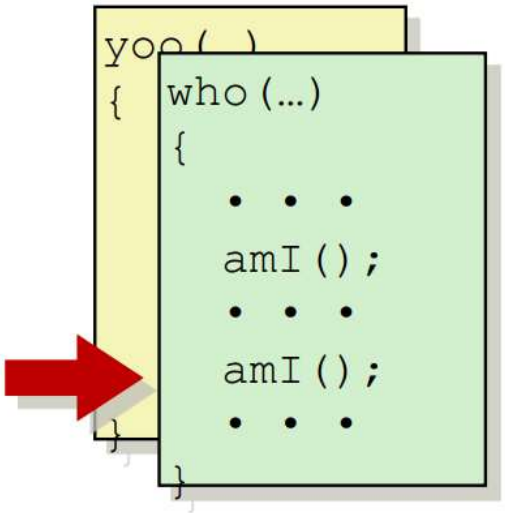


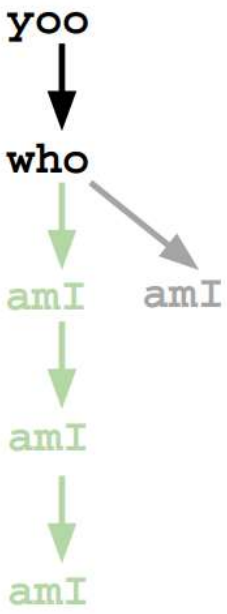


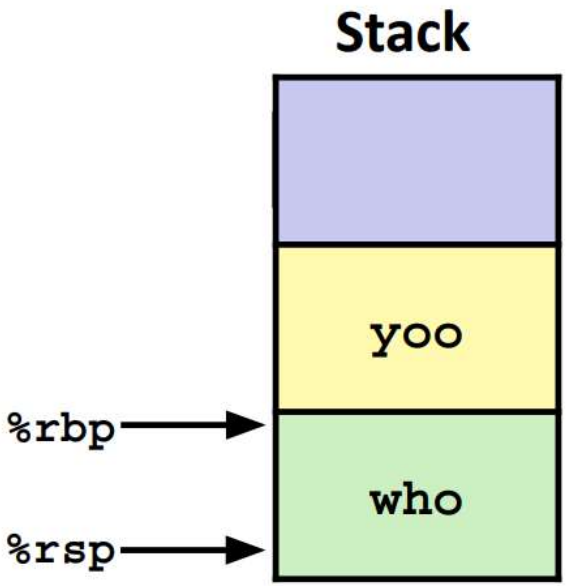
UNIVERSITY OF COPENHAGEN

ITU CPH

Example



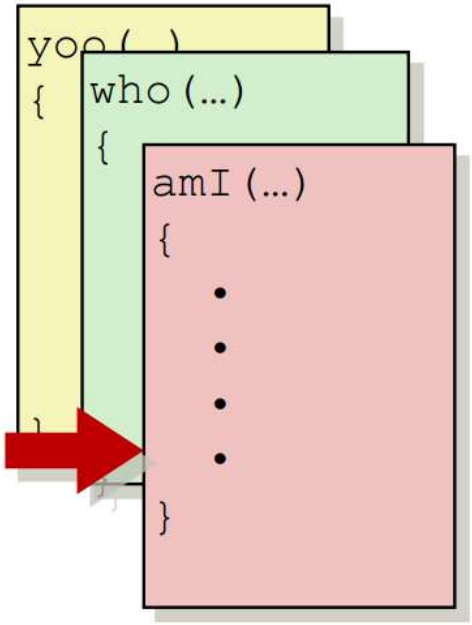


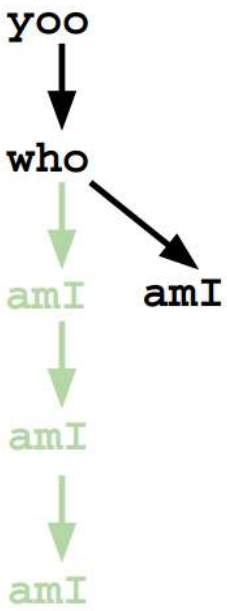


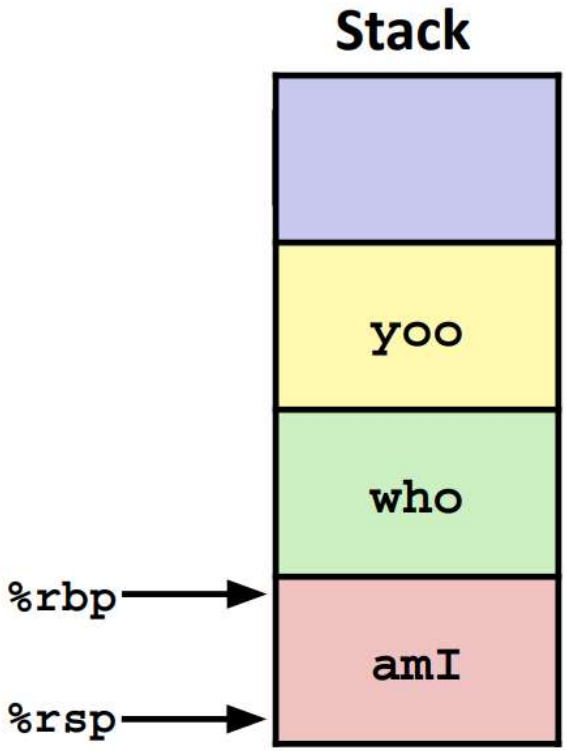
UNIVERSITY OF COPENHAGEN

ITU CPH

Example



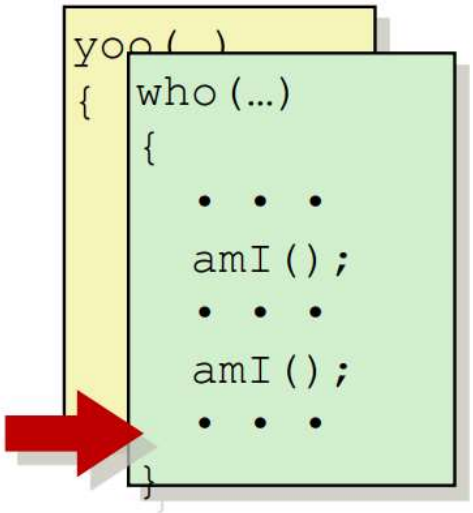


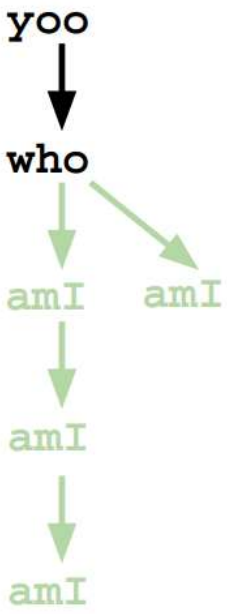


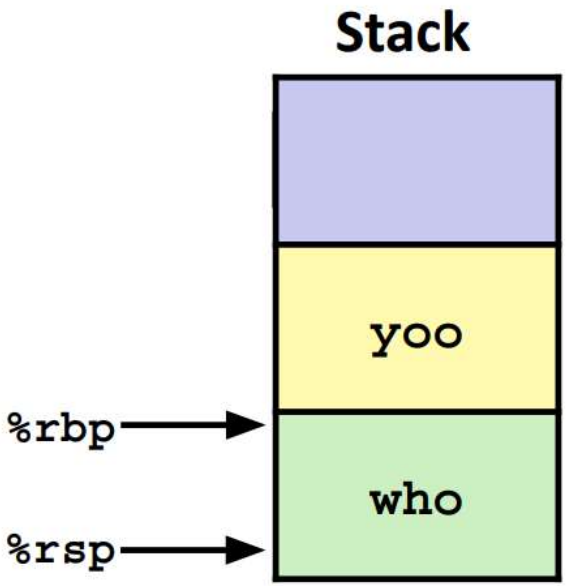
UNIVERSITY OF COPENHAGEN

ITU CPH

Example



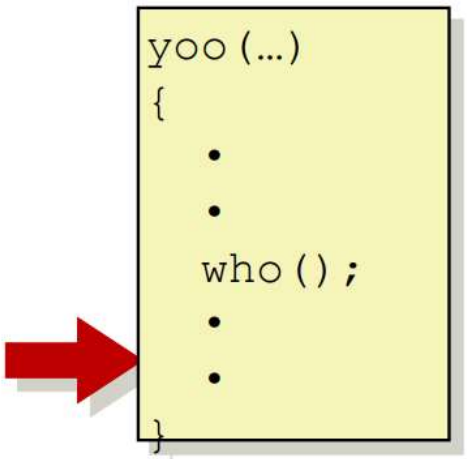


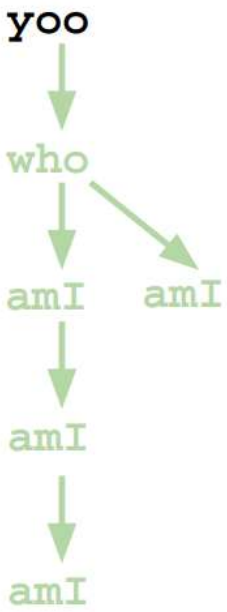


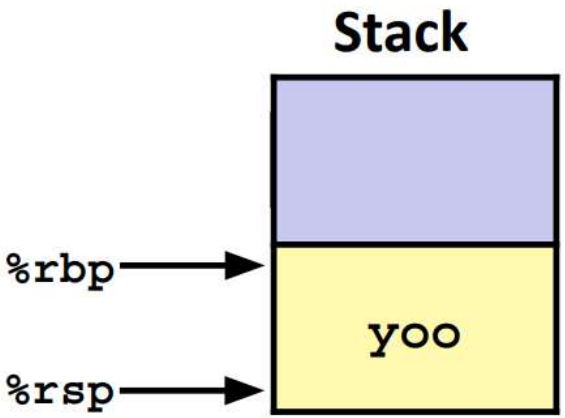
UNIVERSITY OF COPENHAGEN

ITU CPH

Example





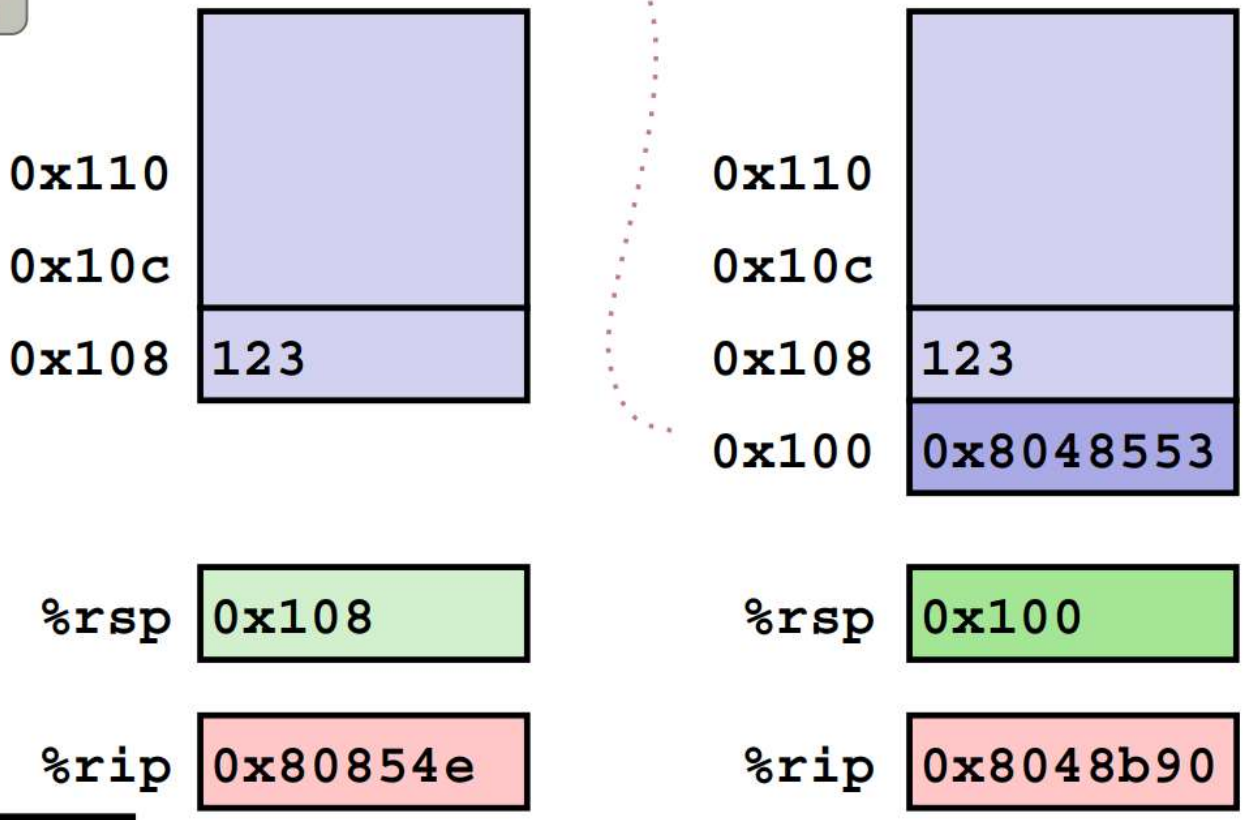


Procedure Call Example

```
804854e:  e8 3d 06 00 00   call   8048b90 <main>
8048553:  50               pushl  %eax
```

return address
= address of next instruction

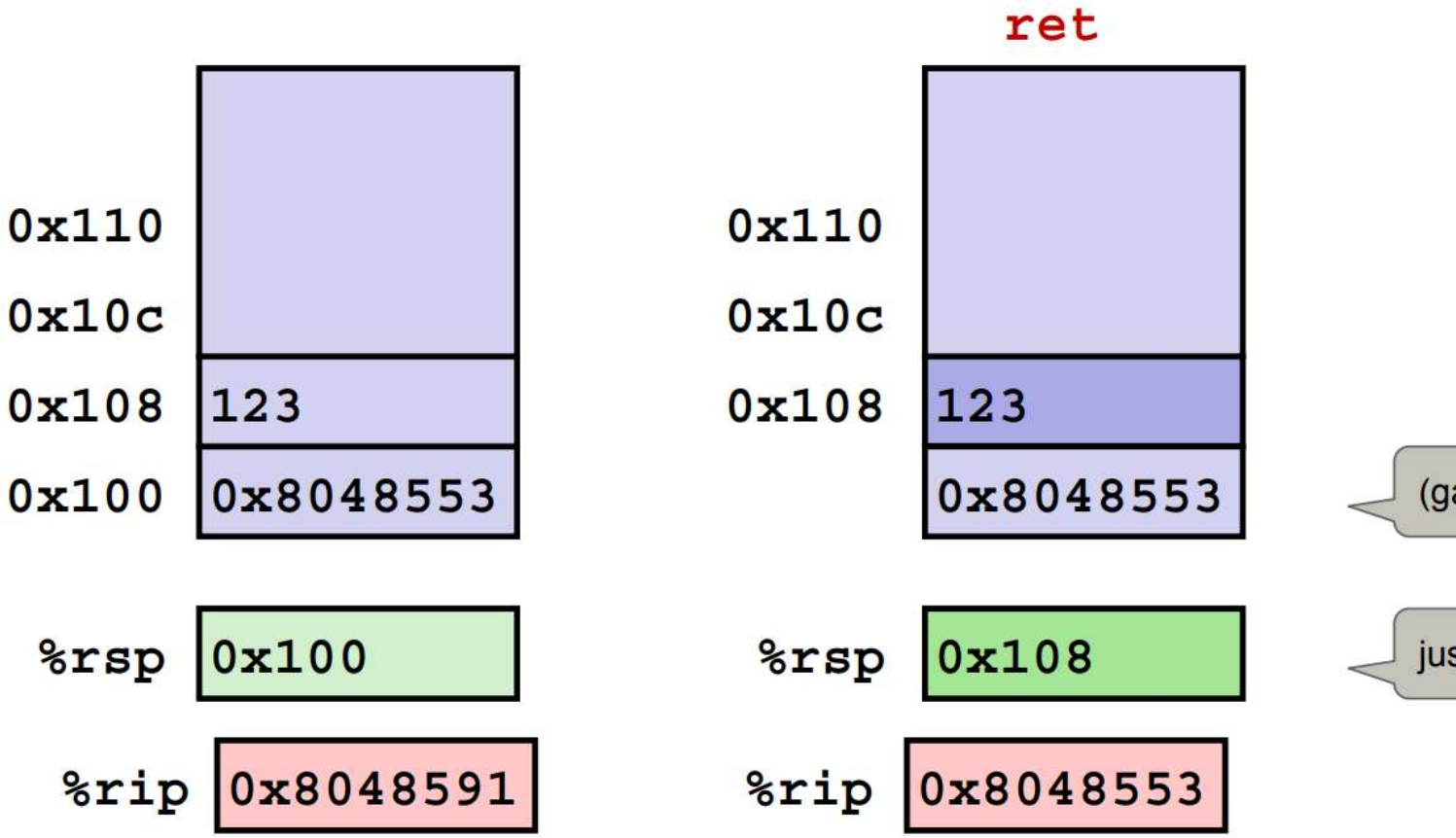
call 8048b90



instruction pointer
updated to callee

Procedure Call Example

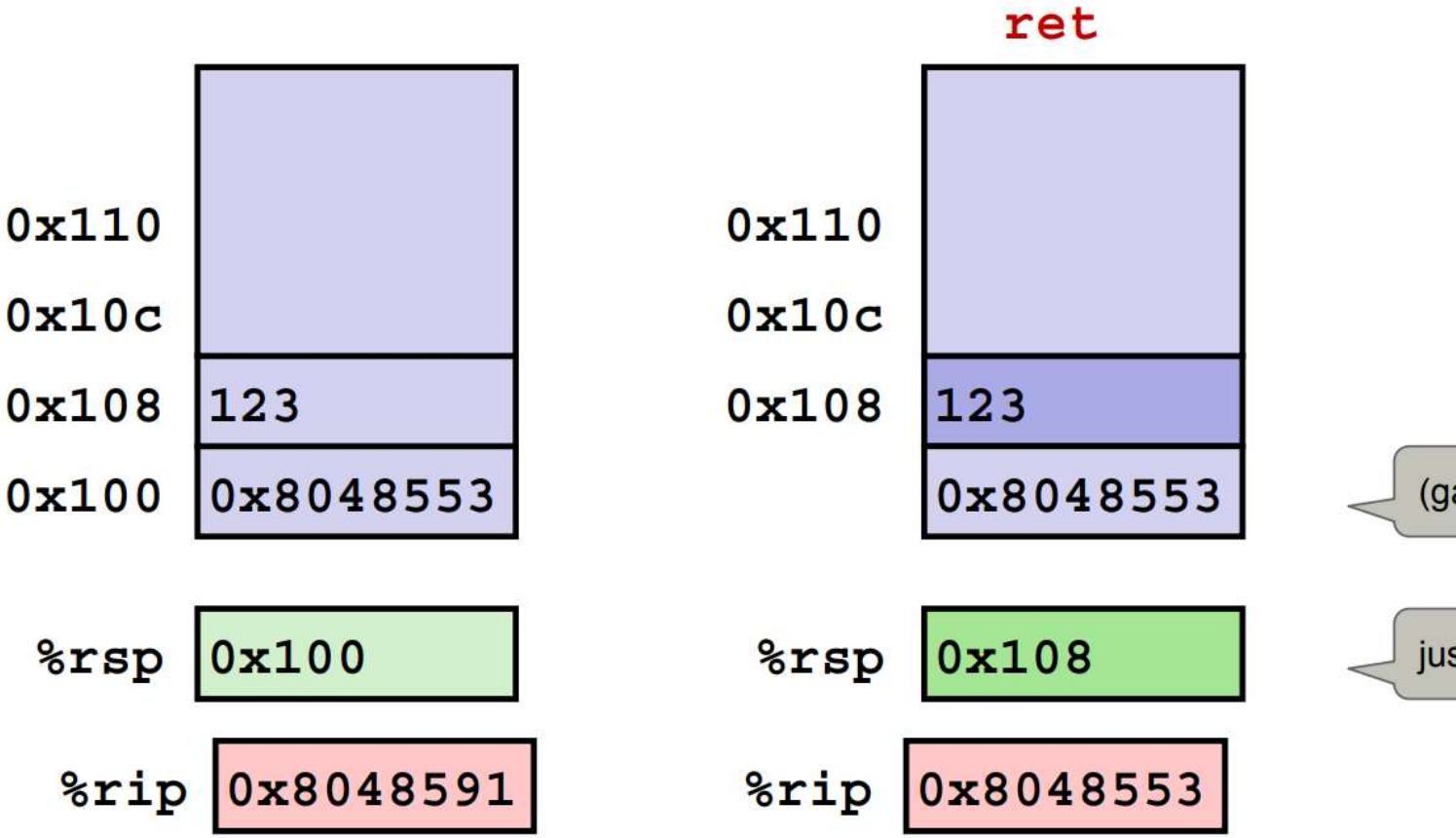
```
8048591:  c3                    ret
```



Q: what happens if user input overwrites stack pointer, or return address?

Procedure Call Example

```
8048591:  c3                    ret
```



Q: what happens if user input overwrites stack pointer, or return address?
A: user has full control over control flow. (exploit)

Calling a function (x86-64)

To call a function, a program:

1. **Places the first six integer or pointer parameters in %rdi, %rsi, %rdx, %rcx, %r8 and %r9**
2. Pushes onto the stack subsequent parameters and parameters larger than 8B (in order).
3. Executes the call instruction, which:
   - Pushes the return address onto the stack
   - Jumps to the start of the specified function

GP Register Usage during function calls

First six arguments of a function stored in
    di, si, dx, cx, r8, r9
Remaining arguments are on the
    stack (more later)

Return value is in rax

What is the stack?
How is the stack used?

Calling a function preserves rbp, rbx, r12-15. The other registers **might be overwritten**.

Executing a function

The C run-time system introduces instruction to set-up and clean-up the stack in each procedure.

Set-up consists in allocation and initialization of a stack-frame. Clean-up: deallocating a stack frame.

A stack-frame is the space needed on the stack by a procedure for storing:

- The return address
- (some) parameters
- Local variables

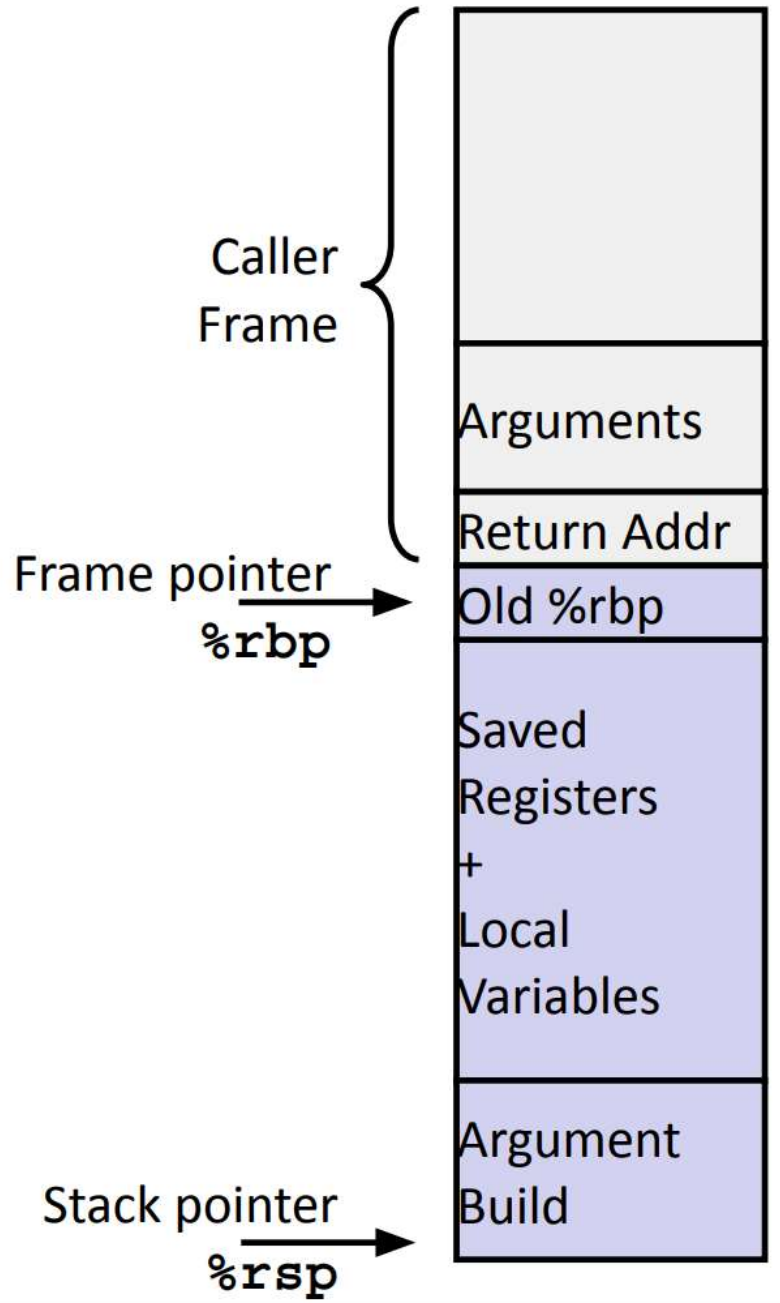
Stack Frame

Caller:

- Arguments
  - pushed by program (if needed)
- Return address
  - pushed by call

Callee:

- Previous frame pointer (%rbp)
- Other callee-save registers (%rbx, %r12-15)
- Space for local variables
- Arguments for next function (when about to call another function)



Example

arith.s arith.c

```c
#include <stdio.h>
#include <stdlib.h>

int logical(int x, int y)
{
        int t1 = x^y;
        int t2 = t1 >> 17;
        int mask = (1<<13) - 7;
        int rval = t2 & mask;
        return rval;
}

int main(int argc, char* argv[])
{
        if (argc != 3) {
                printf("Usage: arith x y\n");
                return 1;
        }

        int x = atoi(argv[1]);
        int y = atoi(argv[2]);
        printf("Arguments x: %d, y: %d\n", x, y);
        printf("Logical returns: %d\n", logical(x,y));
        printf("\n");

        return 0;
}
```

Example

```
logical:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    -20(%rbp), %eax
        xorl    -24(%rbp), %eax
        movl    %eax, -16(%rbp)
        movl    -16(%rbp), %eax
        sarl    $17, %eax
        movl    %eax, -12(%rbp)
        movl    $8185, -8(%rbp)
        movl    -12(%rbp), %eax
        andl    -8(%rbp), %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Set-up:

- Previous stack frame base %rbp pushed on stac
- %rbp is the only callee save register
- Frame pointer re-initialised

Function:

- 4 local variables at positions relative to stack fram base %rbp
  - t1: -16(%rbp)
  - t2: -12(%rbp)
  - mask: -8(%rbp)
  - rval: -4(%rbp)
- %eax holds intermediate results
- %eax holds return value at the end of the function

remember: stack grows down

Clean-up:

- Previous stack frame base restored
- ret manipulates %rsp and %rip to return control to return address

UNIVERSITY OF COPENHAGEN

09.0 2020 46

ITU CPH

Buffer Overflows

ITU CPH

Vulnerable C code

example of vulnerable C code

echo.c

```c
#include <stdio.h>

void echo()
{
        char buf[4];
        gets(buf);
        puts(buf);

}

int main()
{
        echo();

        return 0;
}
```

disable stack protection
(more on that later today)
(so it's not really a problem
today; compilers prevent
problem by default)

```
% gcc -fno-stack-protector echo.c -o echonp
```

```
% ./echonp
12345678
12345678
% ./echonp
123456789
123456789
% ./echonp
1234567890123456789012
1234567890123456789012
% ./echonp
123456789012345678901234567890123
123456789012345678901234567890123
[1]    19108 segmentation fault (core dumped)  ./echonp
```

process tried I/O on
kernel virtual memory
(kernel-space).
what's happening?
let's investigate.

Vulnerable C code

```
objdump -D echonp >> echonp.d
```

The stack is 16B aligned
=>
rsp is decreased with 16B
sub $0x10, %rsp

tip: for assignment, always use objdump. with it, you get assembly, and the object, and the address where it is

"load effective address" (arith on addr, store in specific register)

push old base pointer on stack

new base pointer := old stack pointer

new stack pointer := old stack pointer - 16

= %rsp

address of buff into ra then into 1 arg

initialize return

evict stack frame

```
0000000000400566 <echo>:
  400566:       55                      push   %rbp
  400567:       48 89 e5                mov    %rsp,%rbp
  40056a:       48 83 ec 10             sub    $0x10,%rsp
  40056e:       48 8d 45 f0             lea    -0x10(%rbp),%rax
  400572:       48 89 c7                mov    %rax,%rdi
  400575:       b8 00 00 00 00          mov    $0x0,%eax
  40057a:       e8 d1 fe ff ff          callq  400450 <gets@plt>
  40057f:       48 8d 45 f0             lea    -0x10(%rbp),%rax
  400583:       48 89 c7                mov    %rax,%rdi
  400586:       e8 a5 fe ff ff          callq  400430 <puts@plt>
  40058b:       90                      nop
  40058c:       c9                      leaveq
  40058d:       c3                      retq

000000000040058e <main>:
  40058e:       55                      push   %rbp
  40058f:       48 89 e5                mov    %rsp,%rbp
  400592:       b8 00 00 00 00          mov    $0x0,%eax
  400597:       e8 ca ff ff ff          callq  400566 <echo>
  40059c:       b8 00 00 00 00          mov    $0x0,%eax
  4005a1:       5d                      pop    %rbp
  4005a2:       c3                      retq
  4005a3:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  4005aa:       00 00 00
  4005ad:       0f 1f 00                nopl   (%rax)
```

enough space for buf

UNIVERSITY OF COPENHAGEN

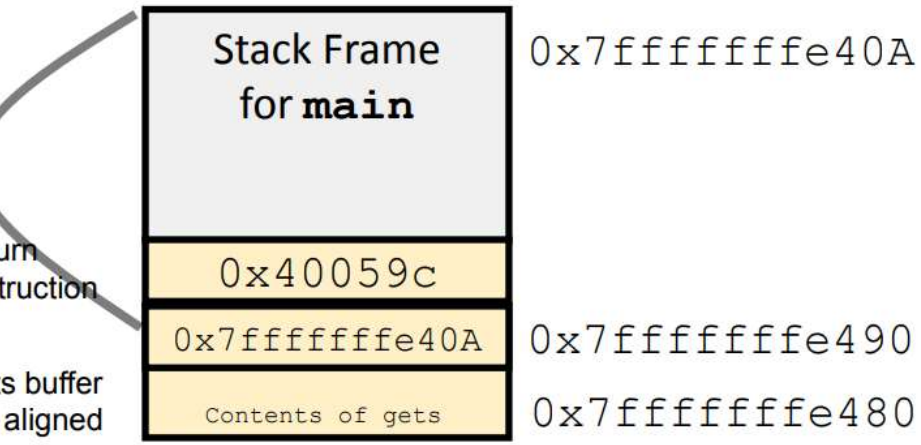
01.10.2020 67

ITU CPH

Vulnerable code

so, why vulnerable?

Before call to gets




```
% gdb ./echonp
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./echonp...done.
(gdb) break echo
Breakpoint 1 at 0x40056e: file echo.c, line 6.
(gdb) r
Starting program: /home/phbo/Class/C/TMP/echonp

Breakpoint 1, echo () at echo.c:6
6               gets(buf);
(gdb) p/x $rbp
$1 = 0x7fffffffe490
(gdb) p/x *((unsigned long*)$rbp)
$2 = 0x7fffffffe4a0
(gdb) p/x *((unsigned long*)$rbp+1)
$3 = 0x40059c
```

%rbp+1 is return instruction

```
(gdb) p/x $rsp
$4 = 0x7fffffffe480
```

if I write more than 16 bytes,
then I overwrite return address.
I can take over the process
(or provoke segmentation fault)

```
342   400597:       e8 ca ff ff ff          callq  400566 <echo>
343   40059c:       b8 00 00 00 00          mov    $0x0,%eax
```

UNIVERSITY OF COPENHAGEN

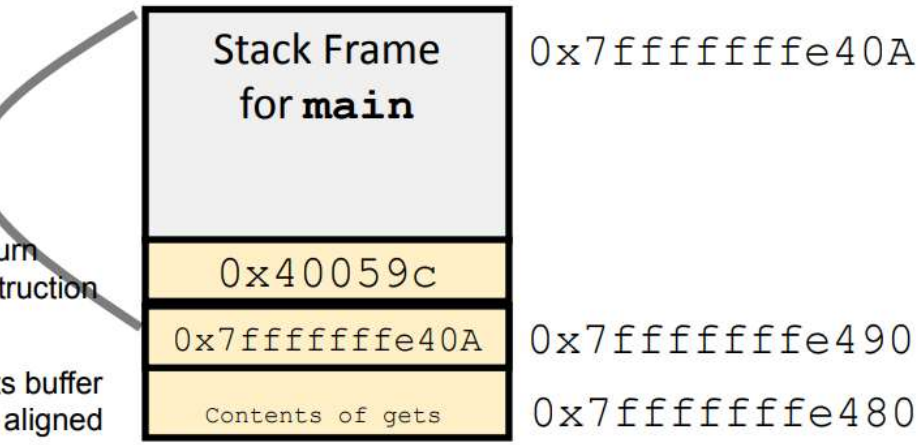
ITU CPH

Vulnerable code

so, why vulnerable?

Press Esc to exit full screen

Before call to gets




```
% gdb ./echonp
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./echonp...done.
(gdb) break echo
Breakpoint 1 at 0x40056e: file echo.c, line 6.
(gdb) r
Starting program: /home/phbo/Class/C/TMP/echonp

Breakpoint 1, echo () at echo.c:6
6               gets(buf);
(gdb) p/x $rbp
$1 = 0x7fffffffe490
(gdb) p/x *((unsigned long*)$rbp)
$2 = 0x7fffffffe4a0
(gdb) p/x *((unsigned long*)$rbp+1)
$3 = 0x40059c
```

%rbp+1 is return instruction

```
(gdb) p/x $rsp
$4 = 0x7fffffffe480
```

if I write more than 16 bytes,
then I overwrite return address.
I can take over the process
(or provoke segmentation fault)

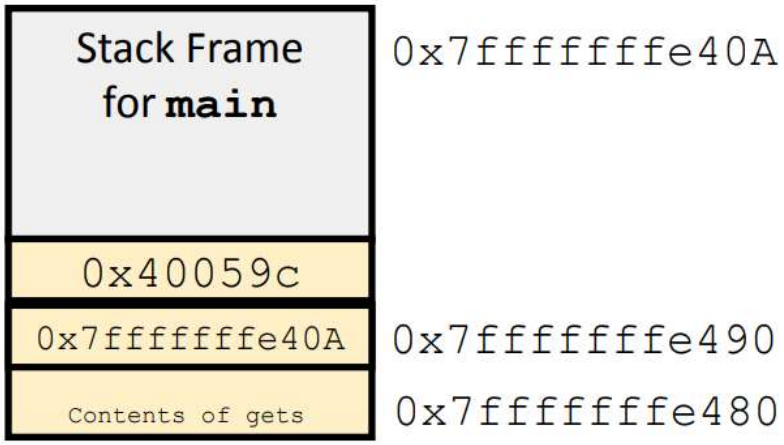
```
342   400597:       e8 ca ff ff ff          callq  400566 <echo>
343   40059c:       b8 00 00 00 00          mov    $0x0,%eax
```

UNIVERSITY OF COPENHAGEN

ITU CPH

Buffer Overflow Stack Example

Before call to gets



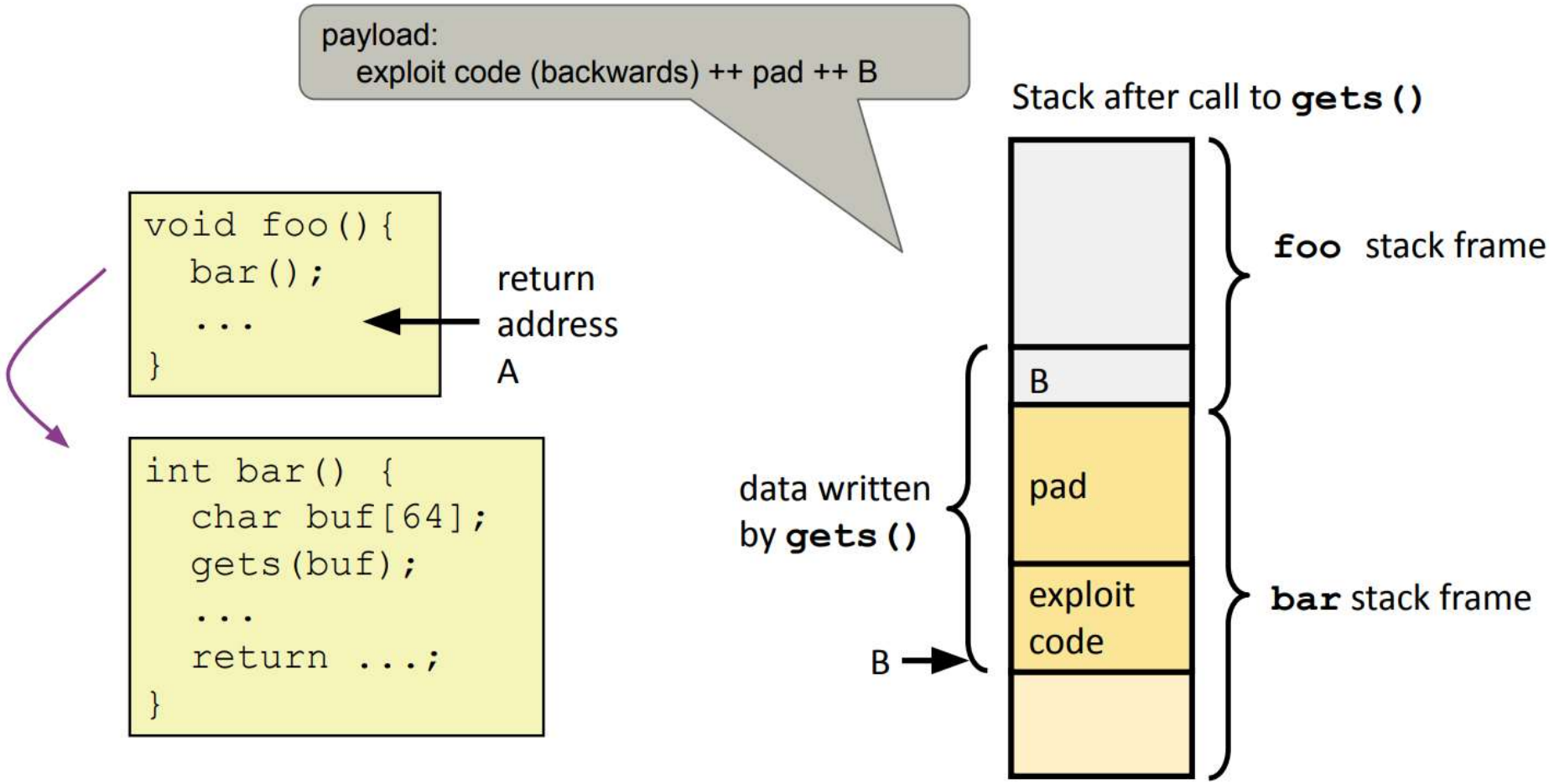
depending on input, different termination behavior.

```
 % ./echonp
12345678901234567890123
12345678901234567890123
[1]    23189 segmentation fault (core dumped)  ./echonp
 % ./echonp
1234567890123456789012345
1234567890123456789012345
[1]    23255 illegal hardware instruction (core dumped)  ./echonp
 % ./echonp
123456789012345678901234
123456789012345678901234
```

UNIVERSITY OF COPENHAGEN

ITU CPH

Malicious Use of Buffer Overflow



Input string contains byte representation of executable code
Overwrite return address with address of exploit code
When `bar()` executes `ret`, will jump to exploit code

UNIVERSITY OF COPENHAGEN

ITU CPH

Exploits Based on Buffer Overflows

Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines

Internet worm

Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:

- **`finger droh@cs.cmu.edu`**

Morris worm, 1988

Worm attacked fingerd server by sending phony argument:

- **`finger "exploit-code padding new-return-address"`**
- exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

how to stop this:

- want to make it impossible to execute code on the stack.
- want to make it hard to figure out where exploit code starts.
- want to protect return address.

Avoiding Overflow Vulnerability

use a function that checks how much you read.

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

Use library routines that limit string lengths

`fgets` instead of **`gets`**

`strncpy` instead of **`strcpy`**

Don't use **`scanf`** with **`%s`** conversion specification

- Use **`fgets`** to read the string
- Or use **`%ns`** where **`n`** is a suitable integer

System-Level Protections

Randomized stack offsets

At start of program, allocate random amount of space on stack

Makes it difficult for hacker to predict beginning of inserted code

Nonexecutable code segments

In traditional x86, can mark region of memory as either "read-only" or "writeable"

- Can execute anything readable

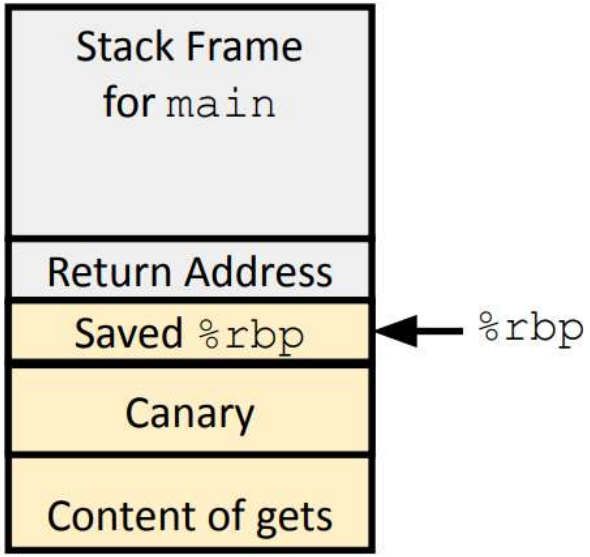
X86-64 added explicit "execute" permission

Stack Canaries

- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - `-fstack-protector`
  - `-fstack-protector-all (default)`

Setting Up Canary

Before call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

random part of memory into rax

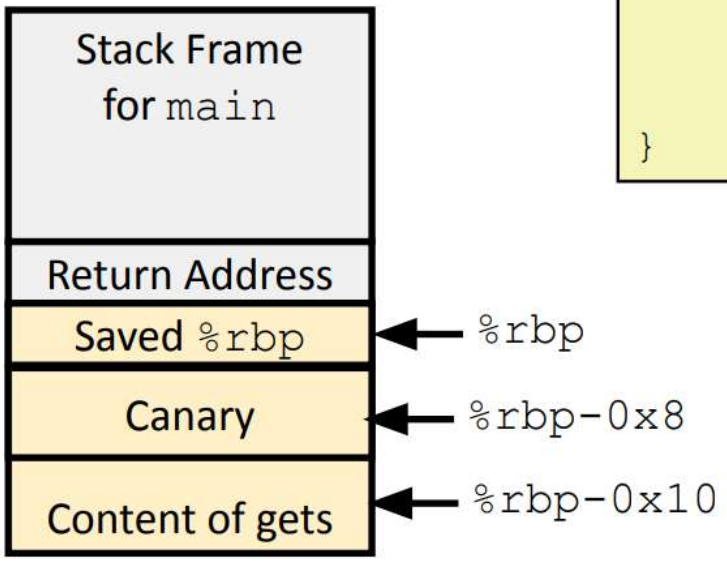
put canary just below base pointer

```
echo:
    . . .
    mov  %fs:20, %rax      # Get canary
    mov  %rax, -8(%rbp)    # Put on stack
    xor  %eax, %eax        # Erase canary
    . . .
```

must overwrite canary to overwrite return address

Checking Canary

Before call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    mov  -8(%rbp), %rax    # Retrieve from
stack
    xor  %fs:20, %rax      # Compare with
Canary
    je   .L24              # Same: skip ahead
    call __stack_chk_fail  # ERROR
.L24:
    . . .
```

Canary Example

```
0000000004005d6 <echo>:
  4005d6:       55                      push   %rbp
  4005d7:       48 89 e5                mov    %rsp,%rbp
  4005da:       48 83 ec 10             sub    $0x10,%rsp
  4005de:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4005e5:       00 00
  4005e7:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  4005eb:       31 c0                   xor    %eax,%eax
  4005ed:       48 8d 45 f0             lea    -0x10(%rbp),%rax
  4005f1:       48 89 c7                mov    %rax,%rdi
  4005f4:       b8 00 00 00 00          mov    $0x0,%eax
  4005f9:       e8 c2 fe ff ff          callq  4004c0 <gets@plt>
  4005fe:       48 8d 45 f0             lea    -0x10(%rbp),%rax
  400602:       48 89 c7                mov    %rax,%rdi
  400605:       e8 86 fe ff ff          callq  400490 <puts@plt>
  40060a:       90                      nop
  40060b:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  40060f:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  400616:       00 00
  400618:       74 05                   je     40061f <echo+0x49>
  40061a:       e8 81 fe ff ff          callq  4004a0 <__stack_chk_fail@plt>
  40061f:       c9                      leaveq
  400620:       c3                      retq

0000000000400621 <main>:
  400621:       55                      push   %rbp
  400622:       48 89 e5                mov    %rsp,%rbp
  400625:       b8 00 00 00 00          mov    $0x0,%eax
  40062a:       e8 a7 ff ff ff          callq  4005d6 <echo>
  40062f:       b8 00 00 00 00          mov    $0x0,%eax
  400634:       5d                      pop    %rbp
  400635:       c3                      retq
  400636:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  40063d:       00 00 00
```

UNIVERSITY OF COPENH

ITU CPH

Canary Example

```
1234567890123456789012345
 % ./echo
12345678
12345678
 % ./echo
123456789
123456789
*** stack smashing detected ***: ./echo terminated
[1]    23643 abort (core dumped)  ./echo
```

when compiled w/ protection

GCC protections

Compile-time

```
cc     echo.c   -o echo
echo.c: In function ‘echo’:
echo.c:6:2: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-declaration]
  gets(buf);
  ^
/tmp/cchUozYT.o: In function `echo':
echo.c:(.text+0x24): warning: the `gets' function is dangerous and should not be used.
```

Run-time

```
 % ./echo
123456789
123456789
*** stack smashing detected ***: ./echo terminated
[1]    17139 abort (core dumped)  ./echo
```

pretty clear that you shouldn't use **gets**.

UNIVERSITY OF COPENHAGEN

ITU CPH