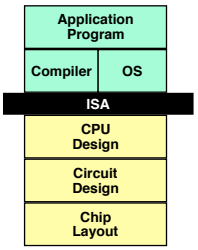
Chapter 4: Processor Architecture

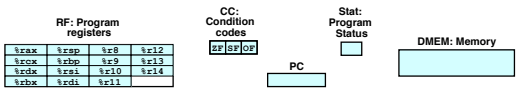
- **Focus: How does the hardware execute the instructions?**
- **We'll see by studying an example system**
  - Based on simple instruction set devised for this purpose: Y86-64
    - New one needed because x86 is too complex to use as example ISA
  - Y86-64 inspired by x86-64, instructions appear similar
    - Fewer data types, instructions, addressing modes
    - Simpler encodings
    - Reasonably complete for integer programs
  - We'll design hardware to implement Y86-64 ISA
    - Basic building blocks
    - Starting point: sequential implementation
    - Ultimate objective: pipelined implementation

Instruction Set Architecture

- **Defines interface between hardware and software**
  - Software spec is assembly language
    - State: registers, memory
    - Instructions, encodings
  - Hardware must execute instructions correctly
    - May use variety of transparent tricks to make execution fast.
    - Critical constraint: ***results must match sequential execution***.
- **ISA is a layer of abstraction**
  - Above: how hardware appears to software
  - Below: what actually gets built



Y86-64 Processor and System State

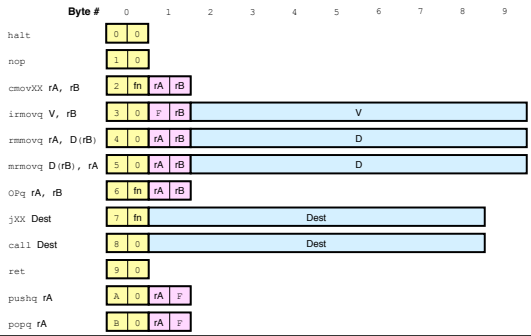


- Program registers
  - 15 64-bit registers (`%r15` is omitted)
- Condition codes
  - Single-bit flags: ZF (Zero), SF (Negative), OF (Overflow)
- Program counter
  - Contains address of next instruction
- Program status
  - Indicates exceptional outcomes (bad opcode, bad address, halt)
- Memory
  - Byte-addressable storage, words in little-endian byte order

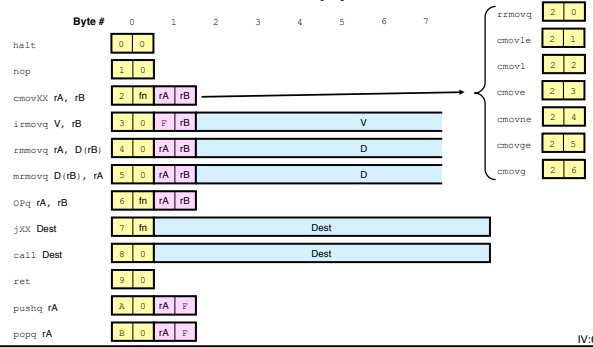
Y86 Instructions

- **Format**
  - 1 to 10 bytes of information read from memory
    - Can determine instruction length from first byte
    - Fewer instruction types and simpler encoding than x86-64
  - Each accesses and modifies some portion of the CPU and system state
    - Program registers
    - Condition codes
    - Program counter
    - Memory contents

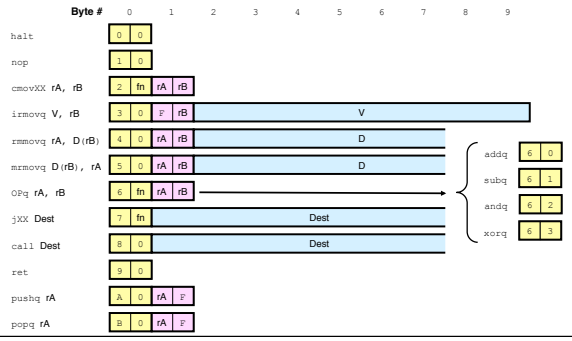
Y86-64 Instruction Set (1)



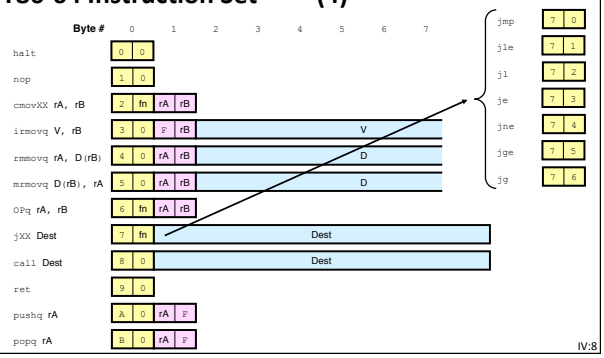
Y86-64 Instruction Set (2)



Y86-64 Instruction Set (3)



Y86-64 Instruction Set (4)



Encoding Registers

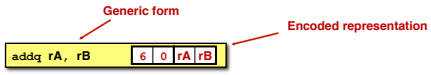
- **Each register has 4-bit ID**
  - Similar encoding used in x86-64
  - But we never had to concern ourselves with it
- **Register ID 0xF indicates "no register"**
  - Used in our hardware design in multiple places
  - Simplifies decoding of instructions

Register + Encoding

%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7
%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Reg	F

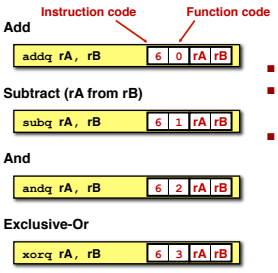
Instruction Example

- **Addition instruction**



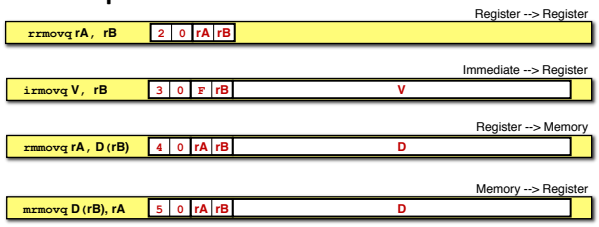
- Add value in register rA to value in register rB
  - Store result in register rB
  - Set condition codes based on result
  - Note: Y86-64 allows ALU operations on register operands only
- Two-byte encoding
  - First byte indicates instruction type
  - Second gives source and destination registers
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

Arithmetic and Logical Operations



- Referred to generically as "OPq"
- Encodings differ only by "function code"
  - Low-order 4 bits in first byte
- All set condition codes as side effect

Move Operations



- Similar to the x86-64 `movq` instruction
- Simpler format for memory addresses
- Separated into different instructions to simplify hardware implementation

Move Instruction Examples

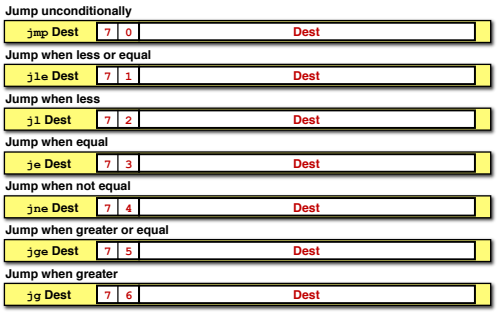
X86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx` Encoding: `30 82 cd ab 00 00 00 00 00 00`
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx` Encoding: `20 43`
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx` Encoding: `50 15 f4 ff ff ff ff ff ff ff`
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)` Encoding: `40 64 1c 04 00 00 00 00 00 00`
`movl $0xabcd, (%eax)`	?
`movl %eax, 12(%eax,%edx)`	?
`movl (%ebp,%eax,4),%ecx`	?

Jump Instructions

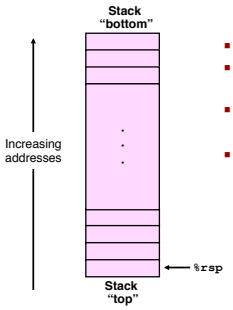
Jump conditionally/unconditionally

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

Jump Instructions



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

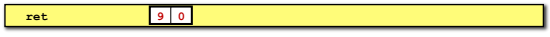
Stack Operations

- Decrement `%rsp` by 8
- Store word from rA to memory at `%rsp`

- Read word from memory at `%rsp`
- Save in rA
- Increment `%rsp` by 8

Subroutine Call and Return

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



- Pop value from stack
- Use popped value as address for next instruction

Miscellaneous Instructions

- Don't do anything

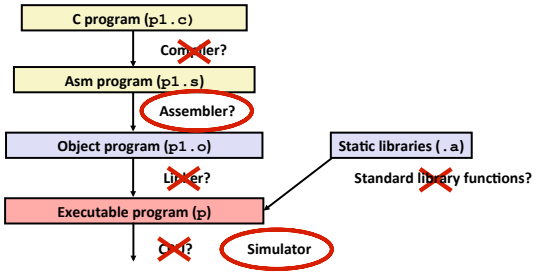
- Stop executing instructions
- x86-64 has comparable instruction, but it can't be executed in user mode
- This instruction is used to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

Status Conditions

Mnemonic	Code	Meaning
AOK	1	Normal operation
HLT	2	Halt instruction encountered
ADR	3	Bad address encountered (instr. or data)
INS	4	Invalid instruction encountered

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

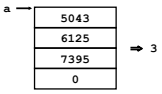
Y86-64: What Tools Exist?



Easiest Way to Get Y86-64 Code?

- **Use C compiler as much as possible**
  - Write code in C
  - Compile for x86-64 with `gcc -S`
  - Hand translate into Y86-64
  - Modern compilers make this more difficult; try different optimization levels
- **Coding example**
  - Find number of elements in null-terminated list

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



Y86-64 Code Generation Example

- **First try**
  - Write typical array code
  - Compile with `gcc -O2 -S`

```
/* Find number of elements in
   null-terminated list */
long len1(long a[])
{
  long len;
  for (len = 0; a[len]; len++)
       ;
  return len;
}
```

- **Result**
  - Hard to translate array indexing on Y86 w/o scaled addressing modes

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


Y86-64 Code Generation Example (2)

- **Second try**
  - Write C code that mimics desired Y86-64 code

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

- **Result**
  - Compiler generates exact same code as before!

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

Y86-64 Code Generation Example (3)

- **Third try**
  - Borrow what you can, carefully hand code the rest

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

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

Y86-64 Program Structure: Example (1)

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

    .align 8        # Program data
array:
    . . .
Main:               # Main function
    . . .
    call len
    . . .
len:                # Length function
    . . .
    .pos 0x100      # Placement of stack
Stack:
```

By convention, file extension is .ys

- Programmer must do more work
  - No compiler, linker, or run-time system
- Stack initialization must be explicit (here: addr. 0x100)
  - Must make sure code is not overwritten!
- Execution starts at address 0
- Data must be initialized
- Can use symbolic names

Y86-64 Program Structure: Example (2)

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

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

- Note:
  - Start of program execution
  - Stack initialization
  - Initialization of array data
  - Use of symbolic names

Y86-64 Program Structure: Example (3)

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

- Note:
  - Set up call to len
  - Follow x86-64 procedure conventions
  - Put array address into %rdi as argument

Assembling Y86-64 Program

```
linux> yas eg.ys
```

- Input is .ys file (as on a previous slide); output is "object code" file `eg.yo`
  - Looks like disassembler output: ASCII file that is easy for you to read

```
0x054:                      | len:
0x054: 30f80100000000000000 |    irmovq $1, %r8          # Constant 1
0x05e: 30f90800000000000000 |    irmovq $8, %r9          # Constant 8
0x068: 30f00000000000000000 |    irmovq $0, %rax         # len = 0
0x072: 502700000000000000   |    mrmovq (%rdi), %rdx     # val = *a
0x07c: 6222                 |    andq %rdx, %rdx         # Test val
0x07e: 73a000000000000000   |    je Done                 # If zero, goto Done
0x087:                      | Loop:
0x087: 6080                 |    addq %r8, %rax          # len++
0x089: 6097                 |    addq %r9, %rdi          # a++
0x08b: 502700000000000000   |    mrmovq (%rdi), %rdx     # val = *a
0x095: 6222                 |    andq %rdx, %rdx         # Test val
0x097: 74870000000000000    |    jne Loop                # If !0, goto Loop
0x0a0:                      | Done:
0x0a0: 90                   |    ret
```


Simulating Y86-64 Program

```
linux> yis eg.yo
```

- Instruction set simulator loads and executes .yo "object file"
  - Computes effect of each instruction on processor state, in sequence
  - At end, prints changes in state from original
  - No library functions, so no print functions allowed in C src

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

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

A Little History: CISC

- **CISC: Complex Instruction Set Computer**
  - Dominant style of machines designed prior to ~1980
- **Stack-oriented instruction set**
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- **Arithmetic instructions can access memory**
  - `addl %eax, 12(%ebx,%ecx,4)`
  - Requires memory read and write + complex address calculation
- **Condition codes**
  - Set as side effect of arithmetic and logical instructions
- **Philosophy**
  - Ideally instructions should perform "typical" programming tasks

RISC

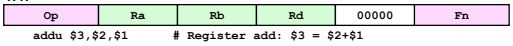
- **Reduced Instruction Set Computer**
  - Early projects at IBM, Stanford (Hennessy), and Berkeley (Patterson)
- **Simpler instructions in ISA (and fewer, at least initially)**
  - Takes more instructions to perform same operations (relative to CISC)
  - But each instruction can execute faster on simpler hardware
- **Register-oriented instruction set**
  - Many more registers (typically ≥ 32)
  - Used for arguments, return value, return address, temporaries
- **Only load and store instructions can access memory**
  - Similar to Y86-64 `mrmovq` and `rmmovq`
- **No condition codes**
  - Compare/test instructions return 0/1 in general purpose register

Example: MIPS Registers

$0	$0	Constant 0
$1	$at	Reserved Temp.
$2	$v0	Return Values
$3	$v1	
$4	$a0	Procedure arguments
$5	$a1	
$6	$a2	
$7	$a3	
$8	$t0	Caller Save Temporaries: May be overwritten by called procedures
$9	$t1	
$10	$t2	
$11	$t3	
$12	$t4	
$13	$t5	
$14	$t6	
$15	$t7	
$16	$s0	Callee Save Temporaries: May not be overwritten by called procedures
$17	$s1	
$18	$s2	
$19	$s3	
$20	$s4	
$21	$s5	
$22	$s6	
$23	$s7	
$24	$t8	Caller Save Temp
$25	$t9	
$26	$k0	Reserved for Operating Sys
$27	$k1	
$28	$gp	Global Pointer
$29	$sp	Stack Pointer
$30	$s8	Callee Save Temp
$31	$ra	Return Address

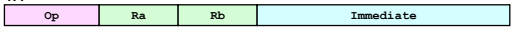
Example: MIPS Instructions

R-R



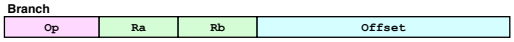

```
addu $3,$2,$1      # Register add: $3 = $2+$1
```

R-I



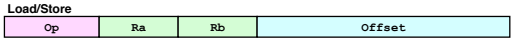

```
addu $3,$2,3145    # Immediate add: $3 = $2+3145
sll $3,$2,2        # Shift left: $3 = $2 << 2
```

Branch




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

Load/Store




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


CISC vs. RISC Debate

- **Strong opinions at the time!**
  - CISC arguments:
    - Easier for compiler (bridges semantic gap)
    - Concise object code (memory was expensive)
  - RISC arguments:
    - Simpler target is better for optimizing compilers
    - Simpler CPU can be made to run faster
- **Current status**
  - For desktop processors, choice of ISA not a limiting factor
    - With enough hardware, anything can be made to run fast
    - x86 has adopted many RISC features (many internal, transparent)
    - Code compatibility is more important
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power

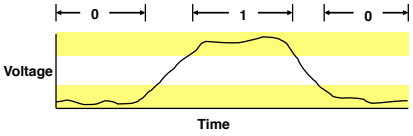
4.1 Summary

- **Y86-64 instruction set architecture**
  - Similar state and instructions as x86-64
  - Simpler encodings
  - Small instruction set
  - Somewhere between CISC and RISC

4.2: Logic Design: A Brief Review

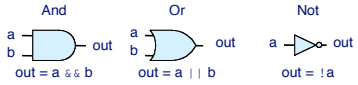
- **Fundamental hardware requirements**
  - Communication
    - How to move values from one place to another
  - Computation
  - Storage
- **All are simplified by restricting to 0s and 1s**
  - Communication
    - Low or high voltage on wire
  - Computation
    - Compute Boolean functions
  - Storage
    - Store bits of information

Communication: Digital Signals

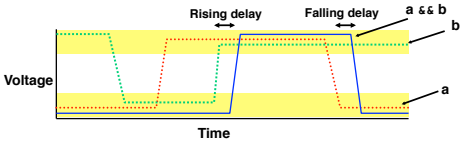


- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low quality circuit elements
  - Can make circuits simple, small, and fast

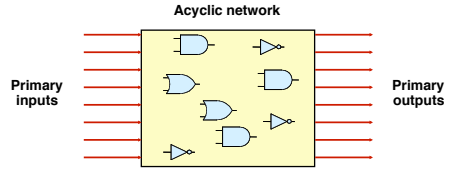
Computation: Logic Gates



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

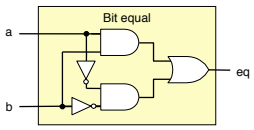


Combinational Circuits



- **Acyclic network of logic gates**
  - Continuously responds to changes on primary inputs
  - Primary outputs become (after some delay) Boolean functions of primary inputs

Computation Example: Bit Equality



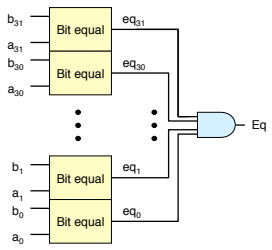
HCL expression

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

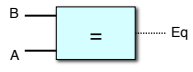
- Generate 1 if a and b are equal

- **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
    - Much more convenient than drawing gates
    - It is assumed that HCL compiler can generate gate equivalent

Word Equality



Word-level representation

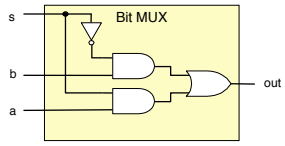


HCL representation

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

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

Bit-Level Multiplexer

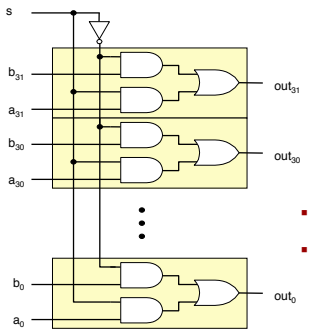


HCL expression

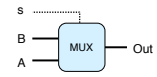
```
bool out = (s&&a)||(!s&&b)
```

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

Word Multiplexer



Word-level representation



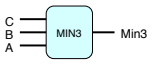
HCL representation

```
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
  - Result value determined by first successful test

HCL Word-Level Examples

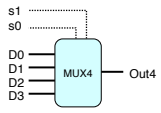
Minimum of 3 words




```
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 multiplexer

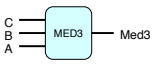



```
int Out4 = [
  !s1&&!s0: D0;
  !s1     : D1;
  !s0     : D2;
  1       : D3;
];
```

- Select one of 4 inputs based on two control bits
- HCL case expression
- Sequential matching can simplify test expressions

Quiz

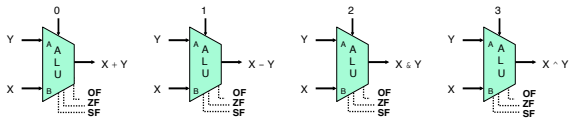
Median of 3 words (?)




```
int Med3 = [
  A <= B && B <= C : B;
  B <= A && A <= C : A;
  1                : C;
];
```

- Intent of the designer: find the median of three input words
- Is the code correct?

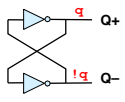
Components: Arithmetic Logic Unit



- Example of higher-level constructs we can use in our designs
- Ultimately, ALU consists of 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

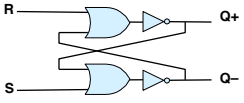
Storage: Latches (A Brief Review)

Bistable Element

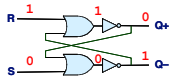


q = 0 or 1

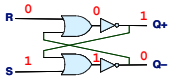
R-S Latch



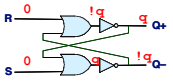
Resetting



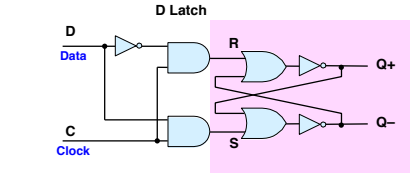
Setting



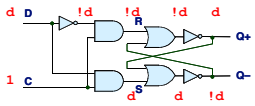
Storing



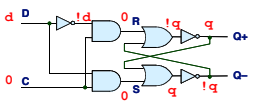
1-Bit Latch



Latching

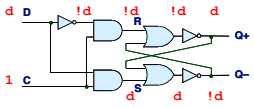


Storing

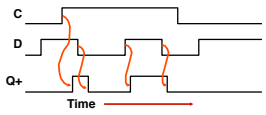


1-Bit Latch

Latching



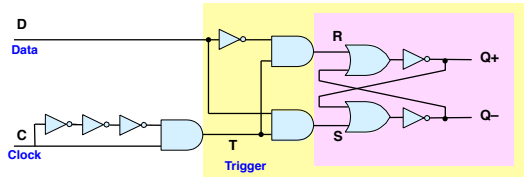
Changing D

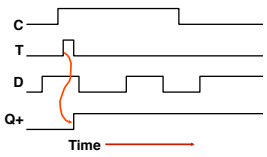


- When C asserted, Q+ follows D (with some delay)
- When C falls, value of D is "latched"

Latch is *transparent*: output follows input when clock input is asserted.

Edge-Triggered Latch

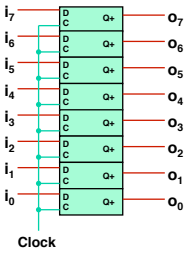




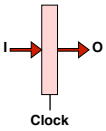
- Only in latching mode for brief period
  - On rising clock edge
- Value latched depends on data as clock rises
- Output remains stable at all other times

Storage: Registers

Structure

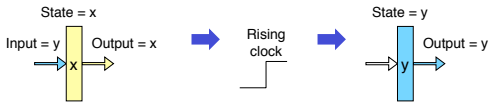


Representation



- Each register stores *word* of data
  - Register at left stores one byte
  - Input loaded on rising edge of clock
- Register is collection of edge-triggered latches

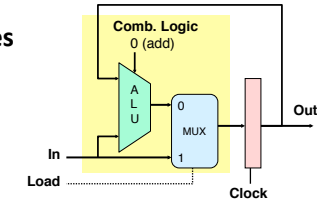
Register Operation



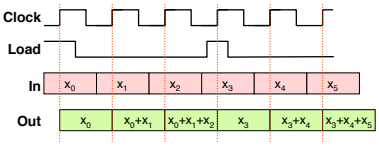
- Stores data bits
- For most of time register acts as barrier between input and output
- As clock rises, register loads input

- Used in our processor for PC, condition codes, Stat, pipeline registers
  - *Not* used for register file (general purpose registers)

Quiz: What Does This Do?

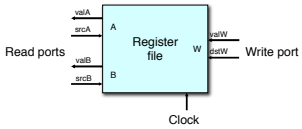


- Accumulator circuit
- Load or accumulate on each cycle

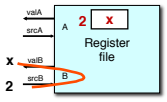


Storage: Random-Access Memory

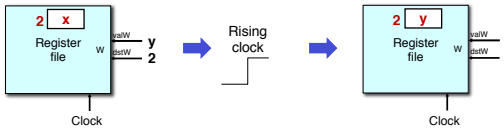
- **Stores multiple words of memory**
  - Address input specifies which word to read or write
- **Example: register file that holds program registers**
  - In Y86-64: `%rax` through `%r14`
  - Register identifier serves as address (ID `0xF` implies no access)
- **Multiple ports, each with separate address and data input/output**
  - Each port allows distinct access in same cycle (below: 2 reads, 1 write)



Register File Timing



- **Reading**
  - Like combinational logic
  - Output data generated based on input address, after some delay
- **Writing**
  - Like edge-triggered latch
  - Updated only as clock rises

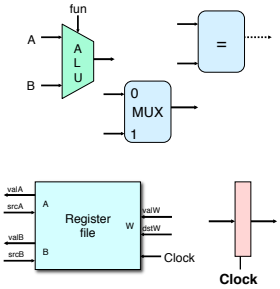


4.2 Summary

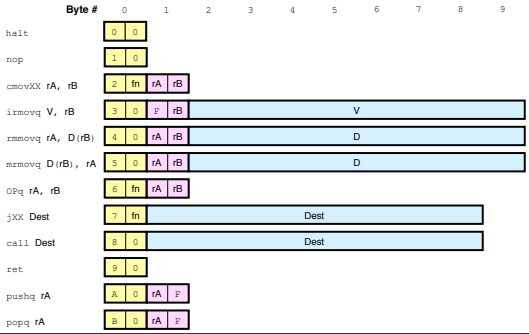
- **Computation**
  - Performed by combinational logic
  - Computes Boolean functions
  - Continuously reacts to input changes
- **Storage**
  - Registers
    - Hold single words
    - Loaded as clock rises
  - Random-access memories
    - Hold multiple words
    - Multiple read and write ports possible
    - Read word anytime address input changes
    - Write word only on rising clock edge

Building Blocks

- **Combinational logic**
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control
- **Storage elements**
  - Addressable memories
  - Non-addressable registers
  - Loaded only as clock rises

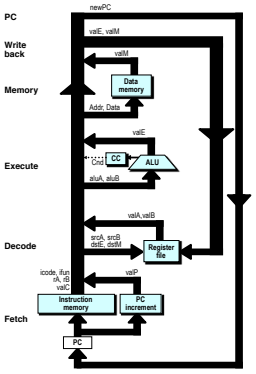


What Might a Y86-64 Datapath Look Like?



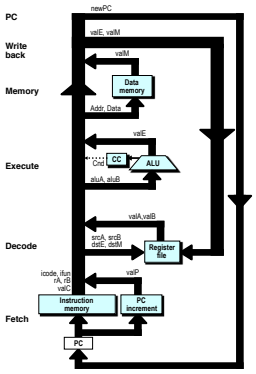
SEQ Hardware Structure

- **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

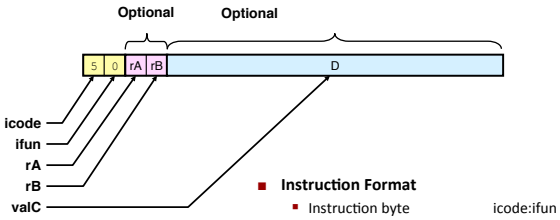


SEQ Stages

- **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	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] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A 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
- Use same general form for all instructions

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

Why twice?

- Use ALU to increment stack pointer
- Must update two registers (popped value + stack pointer)

Executing Conditional Moves

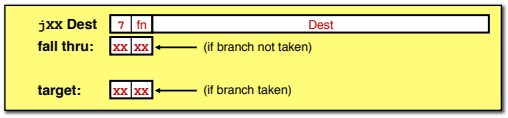
- **Fetch**
  - Read 2 bytes
- **Decode**
  - Read operand registers
- **Execute**
  - If !cnd, then set destination register to 0xF
- **Memory**
  - Do nothing
- **Write back**
  - Update register (unless 0xF)
- **PC update**
  - Increment PC by 2

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

- Store value is passed through ALU (added to 0)
- rB value determines if destination field written or not

Executing Jumps



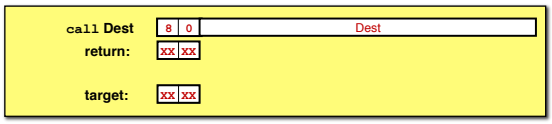
- **Fetch**
  - Read 9 bytes
  - Increment PC by 9
- **Decode**
  - Do nothing
- **Execute**
  - Determine whether to take branch based on jump condition and condition codes
- **Memory**
  - Do nothing
- **Write back**
  - Do nothing
- **PC update**
  - Set PC to Dest if branch taken or to incremented PC if branch not taken

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	Read instruction byte Read destination address Fall through address
Decode		
Execute	Cnd ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Cnd ? valC : valP	Update PC

- Compute both addresses
- Choose next PC based on condition codes and branch condition

Executing `call`



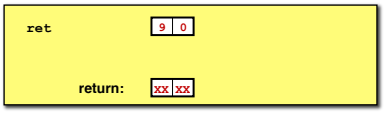
- **Fetch**
  - Read 9 bytes
  - Increment PC by 9
- **Decode**
  - Read stack pointer
- **Execute**
  - Decrement stack pointer by 8
- **Memory**
  - Write incremented PC to new value of stack pointer
- **Write back**
  - Update stack pointer
- **PC update**
  - Set PC to Dest

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`]	Read operand stack pointer
	valB ← R[`%rsp`]	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 condition code register
Memory	valM		[Memory read/write]
Write back	dstE	R[rB] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valP	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computation Steps

		`call Dest`	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_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 condition code]
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

Quiz

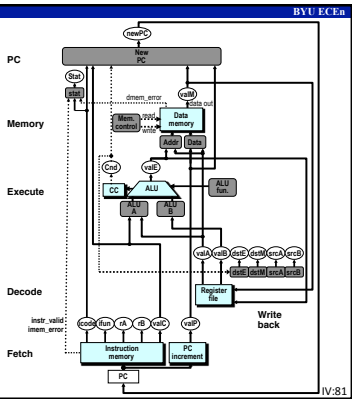
	??
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10
Decode	
Execute	valE ← 0 + valC
Memory	
Write back	R[rB] ← valE
PC update	PC ← valP

What Y86-64 instruction matches this sequence of steps?

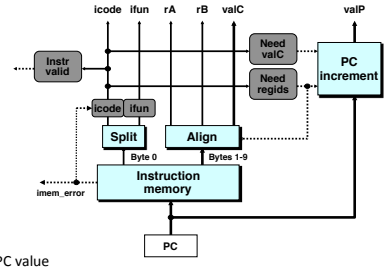
a. irmovq
b. addq
c. rmmovq
d. mrmovq
e. rrmovq
f. none of the above

SEQ Hardware

- **Key**
  - Blue boxes: predesigned hardware blocks
    - E.g., memories, ALU
  - White ovals: labels for signals
    - Thick lines: 64-bit values
    - Thin lines: 4-8 bit values
    - Dotted lines: 1-bit values
  - Gray boxes: control logic
    - Work that remains for us
    - We describe in HCL
- Not all details included here...

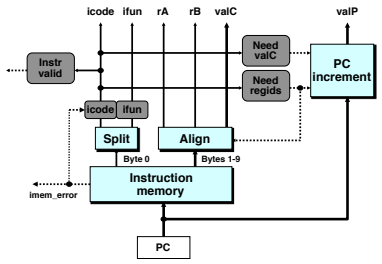


Fetch Logic



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

Fetch Logic

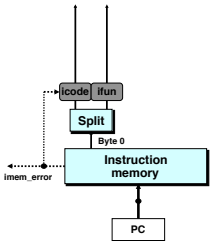


- **Control logic**
  - Instr. valid: Is this instruction valid?
  - Need regids: Does this instruction have a register ID byte?
  - Need valC: Does this instruction have a constant word?

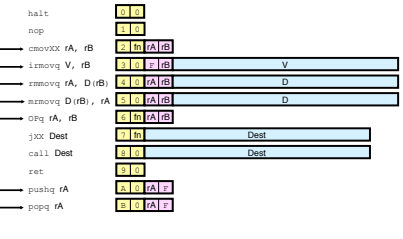
Fetch Control Logic in HCL

```
# Determine instruction code
int icode = [
        imem_error: INOP;
        1: imem_icode;
];

# Determine instruction function
int ifun = [
        imem_error: FNONE;
        1: imem_ifun;
];
```



Fetch Control Logic in HCL

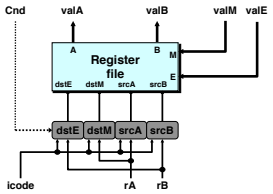



```
bool need_regids =
        icode in { IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
                   IOPQ, IPUSHQ, IPOPQ };

bool instr_valid = icode in
        { IHALT, INOP, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
          IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };
```


Decode Logic

- **Register file**
  - Read ports A, B
  - Write ports E, M
  - Address inputs are register IDs or 0xF (no access)
- **Control logic**
  - srcA, srcB: read port addresses
  - dstE, dstM: write port addresses
- **Signals**
  - Cnd: perform cond. move? Computed in Execute stage



A Source

OPq rA, rB		
Decode	valA ← R[rA]	Read operand A
cmovXX rA, rB		
Decode	valA ← R[rA]	Read operand A
rmmovq rA, D(rB)		
Decode	valA ← R[rA]	Read operand A
popq rA		
Decode	valA ← R[%rsp]	Read stack pointer
jXX Dest		
Decode		No operand
call Dest		
Decode		No operand
ret		
Decode	valA ← R[%rsp]	Read stack pointer

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

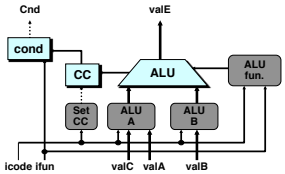

E Destination

OPq rA, rB		
Write-back	R[rB] ← valE	Write back result
cmovXX rA, rB		
Write-back	R[rB] ← valE	Conditionally write back result
rmmovq rA, D(rB)		
Write-back		None
popq rA		
Write-back	R[%rsp] ← valE	Update stack pointer
jXX Dest		
Write-back		None
call Dest		
Write-back	R[%rsp] ← valE	Update stack pointer
ret		
Write-back	R[%rsp] ← valE	Update stack pointer

```
int dstE = [
        icode in { IRRMOVQ } && Cnd : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```


Execute Logic

- **Units**
  - ALU
    - Implements 4 required functions
    - Generates valE and CC values
  - CC
    - Register with 3 condition code bits
  - cond
    - Computes branch/cond. 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.: Operation ALU is to compute



ALU A Input

OPq rA, rB		
Execute	valE ← valB OP valA	Perform ALU operation
cmovXX rA, rB		
Execute	valE ← 0 + valA	Pass valA through ALU
rmmovq rA, D(rB)		
Execute	valE ← valB + valC	Compute effective address
popq rA		
Execute	valE ← valB + 8	Increment stack pointer
jXX Dest		
Execute		No operation
call Dest		
Execute	valE ← valB + –8	Decrement stack pointer
ret		
Execute	valE ← valB + 8	Increment stack pointer

```
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
        icode in { ICALL, IPUSHL } : -8;
        icode in { IRET, IPOPL } : 8;
        # Other instructions don't need ALU
];
```

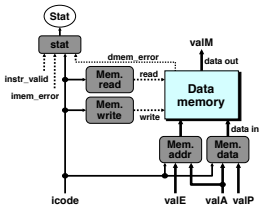
ALU Operation

	OPq rA, rB	
Execute	valE ← valB OP valA	Perform ALU operation
	cmovXX rA, rB	
Execute	valE ← 0 + valA	Pass valA through ALU
	rmmovq rA, D(rB)	
Execute	valE ← valB + valC	Compute effective address
	popq rA	
Execute	valE ← valB + 8	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	valE ← valB + –8	Decrement stack pointer
	ret	
Execute	valE ← valB + 8	Increment stack pointer

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

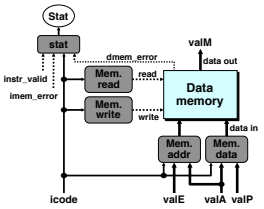

Memory Logic

- **Memory**
  - Reads or writes memory word
- **Control logic**
  - stat: what is instruction status?
  - Mem. read: should word be read?
  - Mem. write: should word be written?
  - Mem. addr.: location to access
  - Mem. data: value to be written



Instruction Status

- **Control logic**
  - stat: what is instruction status?




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


Memory Address

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

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

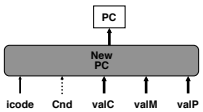

Memory Read

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

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


PC Update Logic

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



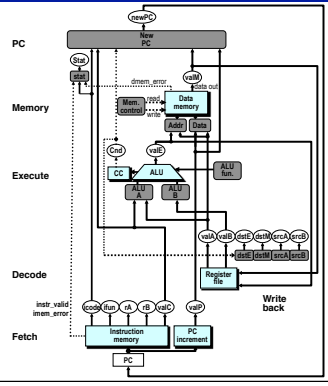
PC Update

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

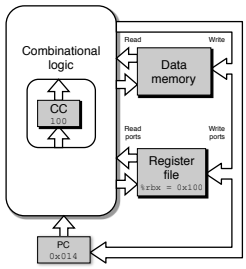
```
int new_pc = [
        icode == ICALL : valC;
        icode == IJXX && Cnd : valC;
        icode == IRET : valM;
        1 : valP;
];
```


SEQ Hardware

- **The story so far...**
  - We *derived* a datapath for the Y86-64 instruction set
  - We used basic building blocks (blue boxes)
  - We named each required register and value
  - We identified where control logic was required (gray boxes)
  - We devised control logic by writing HCL code
- **So, how well does it work?**
  - What happens on each rising clock edge?



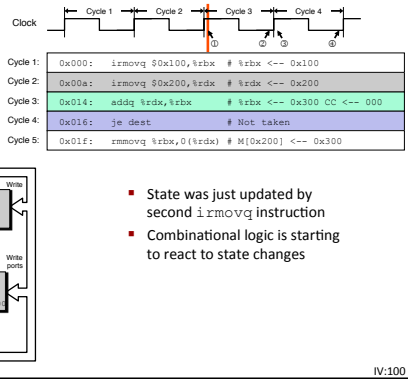
SEQ Operation: An Abstracted View

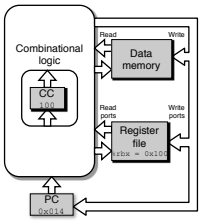


- **State**
  - PC register
  - Cond. code register
  - Data memory
  - Register file

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

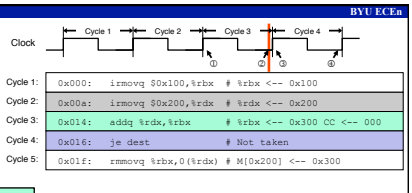
SEQ Operation: State at ①

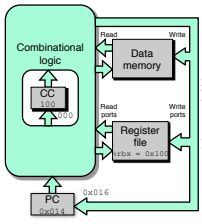




- State was just updated by second `irmovq` instruction
- Combinational logic is starting to react to state changes

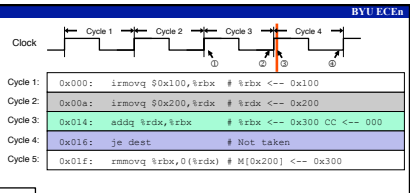
SEQ Operation: State at ②

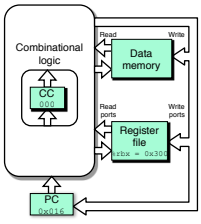




- State as yet unchanged by the current instruction
- Combinational logic has determined results for `addq` instruction
- What state changes will take place on next clock edge?

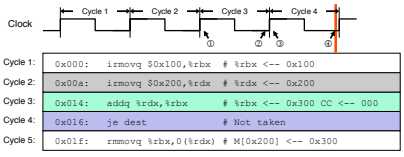
SEQ Operation: State at ③



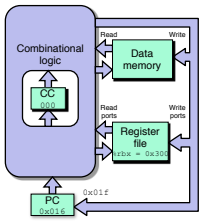


- State was just updated according to `addq` instruction
- Combinational logic is starting to react to state changes

SEQ Operation: State at 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 not yet changed by current instruction
- Combinational logic has determined results for `je` instruction

SEQ Summary

- **RTL (Register Transfer Level) implementation**
  - Every instruction was expressed as series of simple steps
  - The same general flow was followed for all instruction types
  - Datapath used registers, memories, predesigned combinational blocks
  - Control logic connected it all together
- **Limitations**
  - Too slow to be practical
  - In one cycle, signals must propagate through instruction memory, register file, ALU, and data memory
  - Would need to run clock *very slowly*
  - Hardware units active for fraction of clock cycle – utilization is low

Discussion

- **Complexity of data path reflects complexity of ISA**
- **Thought experiments to illustrate:**
  - What changes would be required in the data path if
    - ALU instructions could use input operands from memory?
    - ALU instructions could write results to memory?