

## Computer Architecture: Y86-64 Sequential Implementation

CSci 2021: Machine Architecture and Organization  
March 23rd-25th, 2020

Your instructor: Stephen McCamant

Based on slides originally by:  
Randy Bryant and Dave O'Hallaron

- 1 -

CS:APP3e

## Y86-64 Instruction Set

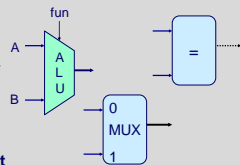
Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	fn	rB						V
rmmovq rA, D(rB)	4	0	rA	rB						D
mrmovq D(rB), rA	5	0	rA	rB						D
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn								Dest
call Dest	8	0								Dest
ret	9	0								
pushq rA	A	0	rA	fn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

CS:APP3e

## Building Blocks

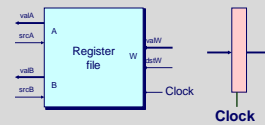
### Combinational Logic

- 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



- 6 -

CS:APP3e

## Hardware Control Language

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

### Data Types

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

### Statements

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

- 7 -

CS:APP3e

## HCL Operations

- Classify by type of value returned

### Boolean Expressions

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

### Word Expressions

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

- 8 -

CS:APP3e

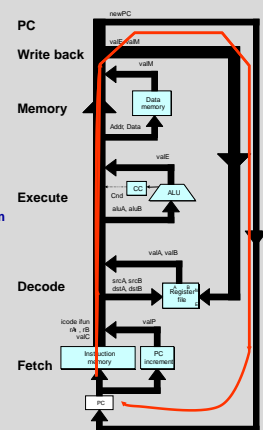
## SEQ Hardware Structure

### 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

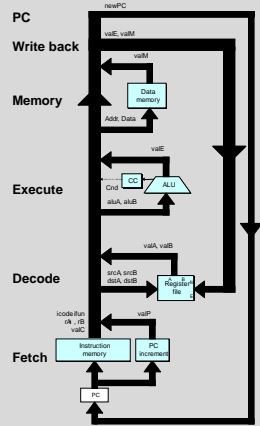
- Read instruction at address specified by PC
- Process through stages
- Update program counter



- 9 -

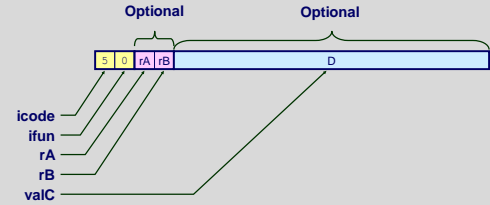
## SEQ Stages

- 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



- 10 -

## Instruction Decoding



### Instruction Format

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

- 11 -

CS:APP3e

## 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

- 12 -

CS:APP3e

## Stage Computation: Arith/Log. Ops

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

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

- 13 -

CS:APP3e

## 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

- 14 -

CS:APP3e

## Stage Computation: rmmovq

	rmmovq rA, D(rB)	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_6[PC+2]$ $valP \leftarrow PC+10$	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	Read operand A Read operand B
Execute	$valE \leftarrow valB + valC$	Compute effective address
Memory	$M_8[valE] \leftarrow valA$	Write value to memory
Write back		
PC update	$PC \leftarrow valP$	Update PC

- Use ALU for address computation

- 15 -

CS:APP3e

## Executing popq



- |                                |                               |
|--------------------------------|-------------------------------|
| <b>Fetch</b>                   | <b>Memory</b>                 |
| ■ Read 2 bytes                 | ■ Read from old stack pointer |
| <b>Decode</b>                  | <b>Write back</b>             |
| ■ Read stack pointer           | ■ Update stack pointer        |
| <b>Execute</b>                 | ■ Write result to register    |
| ■ Increment stack pointer by 8 | <b>PC Update</b>              |
|                                | ■ Increment PC by 2           |

– 16 –

CS:APP3e

## Stage Computation: popq

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

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

– 17 –

CS:APP3e

## Executing Conditional Moves



- |   |                            |
|---|----------------------------|
| <b>Fetch</b>                                    | <b>Memory</b>              |
| ■ Read 2 bytes                                  | ■ Do nothing               |
| <b>Decode</b>                                   | <b>Write back</b>          |
| ■ Read operand registers                        | ■ Update register (or not) |
| <b>Execute</b>                                  | <b>PC Update</b>           |
| ■ If !cnd, then set destination register to 0xF | ■ Increment PC by 2        |

– 18 –

CS:APP3e

## Stage Computation: Cond. Move

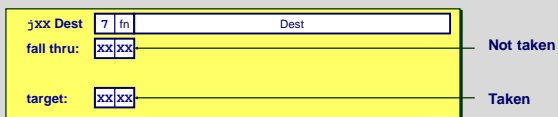
	cmovXX rA, rB		
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_2[PC+1]$	Read instruction byte Read register byte	
	valP $\leftarrow PC+2$	Compute next PC	
Decode	valA $\leftarrow R[rA]$ valB $\leftarrow 0$	Read operand A	
Execute	valE $\leftarrow$ valB + valA if ! Cond(CC,ifun) rB $\leftarrow$ 0xF	Pass valA through ALU (Disable register update)	
Memory			
Write back	R[rB] $\leftarrow$ valE	Write back result	
PC update	PC $\leftarrow$ 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

– 19 –

CS:APP3e

## Executing Jumps



- |  |   |
|--|---|
| <b>Fetch</b>   | <b>Memory</b>   |
| ■ Read 9 bytes   | ■ Do nothing  |
| ■ Increment PC by 9  | <b>Write back</b>   |
| <b>Decode</b>  | ■ Do nothing  |
| ■ Do nothing   | <b>PC Update</b>  |
| <b>Execute</b>   | ■ Set PC to Dest if branch taken or to incremented PC if not branch |
| ■ Determine whether to take branch based on jump condition and condition codes |   |

– 20 –

CS:APP3e

## Stage Computation: Jumps

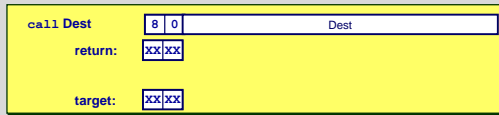
	JXX Dest	
Fetch	icode:ifun ← M <sub>1</sub> [PC]	Read instruction byte
	valC ← M <sub>8</sub> [PC+1]	Read destination address
	valP ← PC+9	Fall through address
Decode		
Execute	Cnd ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Cnd ? valC : valP	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

– 21 –

CS:APP3e

## 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

– 22 –

CS:APP3e

## Stage Computation: call

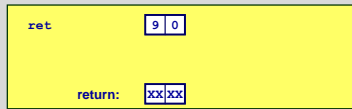
call Dest		
Fetch	icode:ifun $\leftarrow M_1[PC]$  valC $\leftarrow M_6[PC+1]$ valP $\leftarrow PC+9$	Read instruction byte  Read destination address Compute return point
Decode	valB $\leftarrow R[\%rsp]$	Read stack pointer
Execute	valE $\leftarrow valB + -8$	Decrement stack pointer
Memory	$M_8[valE] \leftarrow valP$	Write return addr. on stack
Write back	$R[\%rsp] \leftarrow valE$	Update stack pointer
PC update	$PC \leftarrow valC$	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

– 23 –

CS:APP3e

## 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

– 24 –

CS:APP3e

## Stage Computation: ret

ret		
Fetch	icode:ifun $\leftarrow M_1[PC]$	Read instruction byte
Decode	valA $\leftarrow R[\%rsp]$ valB $\leftarrow R[\%rsp]$	Read operand stack pointer Read operand stack pointer
Execute	valE $\leftarrow valB + 8$	Increment stack pointer
Memory	valM $\leftarrow M_8[valA]$	Read return address
Write back	$R[\%rsp] \leftarrow valE$	Update stack pointer
PC update	$PC \leftarrow valM$	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

– 25 –

CS:APP3e

## Computation Steps

OPq rA, rB		
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$ valC valP	Read instruction byte Read register byte [Read constant word] Compute next PC
Decode	valA, srcA valB, srcB	Read operand A Read operand B
Execute	valE Cond code	Perform ALU operation Set/use cond. code reg
Memory	valM	[Memory read/write]
Write back	dstE dstM	Write back ALU result [Write back memory result]
PC update	PC	Update PC

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

– 26 –

CS:APP3e

## Computation Steps

call Dest		
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB valC valP	Read instruction byte [Read register byte] Read constant word Compute next PC
Decode	valA, srcA valB, srcB	[Read operand A] Read operand B
Execute	valE Cond code	Perform ALU operation [Set /use cond. code reg]
Memory	valM	Memory read/write
Write back	dstE dstM	Write back ALU result [Write back memory result]
PC update	PC	Update PC

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

– 27 –

CS:APP3e

## Computed Values

### Fetch

icode Instruction code  
ifun Instruction function  
rA Instr. Register A  
rB Instr. Register B  
valC Instruction constant  
valP Incremented PC

### Execute

valE ALU result  
Cnd Branch/move flag

### Memory

valM Value from memory

### 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

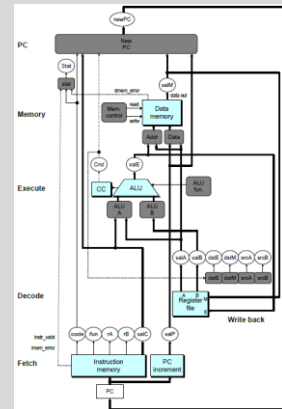
- 28 -

CS:APP3e

## SEQ Hardware

### Key

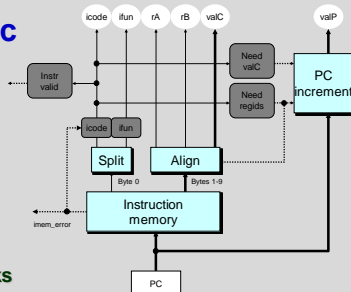
- Blue boxes: predefined hardware blocks
  - E.g., memories, ALU
- Gray boxes: control logic
  - Describe in HCL
- White ovals: labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



- 30 -

CS:APP3e

## Fetch Logic



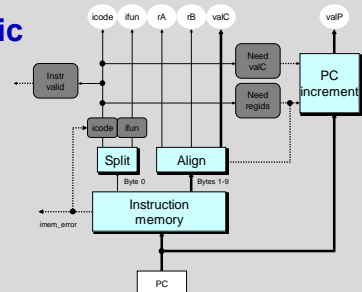
### Predefined Blocks

- 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

- 31 -

CS:APP3e

## Fetch Logic



### Control Logic

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

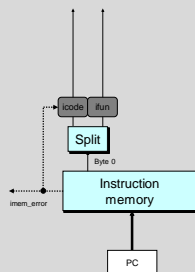
- 32 -

CS:APP3e

## Fetch Control Logic in HCL

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

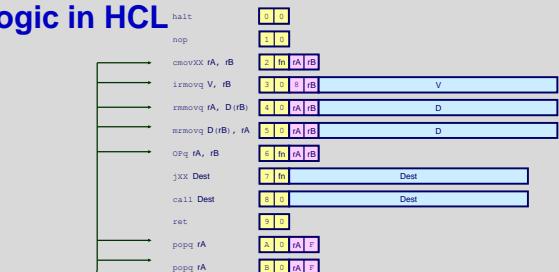
# Determine instruction function
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];
```



- 33 -

CS:APP3e

## Fetch Control Logic in HCL



```
bool need_regs =
    icode in { IRRMOVQ, IOPOQ, IPUSHQ, IPOPOQ,
               IIRMOVQ, IRMMOVQ, IMRMOVQ };
```

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

- 34 -

CS:APP3e

## Decode Logic

### Register File

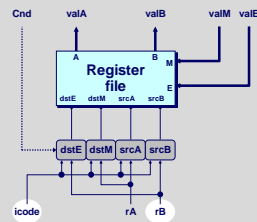
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

### 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



- 35 -

CS:APP3e

## A Source

	OPq rA, rB	
Decode	valA ← R[rA]	Read operand A
	cmovXX rA, rB	
Decode	valA ← R[rA]	Read operand A
	xmmovq 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, IOPOQ, IPUS HQ } : rA;
    icode in { IPOPOQ, IRET } : RRSP;
    1 : RNONE; # Don't need register
];
```

- 36 -

CS:APP3e

## 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
	xmmovq 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, IOPOQ } : rB;
    icode in { IPUS HQ, IPOPOQ, ICALL, IRET } : RRSP;
    1 : RNONE; # Don't write any register
];
```

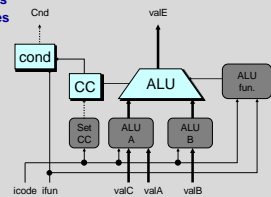
- 37 -

CS:APP3e

## 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?

- 38 -

CS:APP3e

## ALU A Input

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

```
int aluA = [
    icode in { IRRMOVQ, IOPOQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUS HQ } : -8;
    icode in { IRET, IPOPOQ } : 8;
    # Other instructions don't need ALU
];
```

- 39 -

CS:APP3e

## ALU Operation

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

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

- 40 -

CS:APP3e

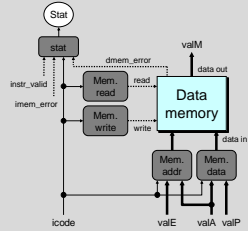
## Memory Logic

### 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



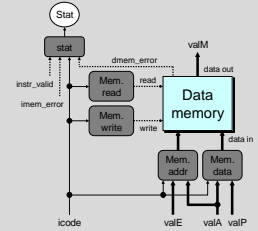
- 41 -

CS:APP3e

## 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;
];
```

- 42 -

CS:APP3e

## Memory Address

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

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

- 43 -

CS:APP3e

## Memory Read

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

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

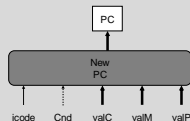
- 44 -

CS:APP3e

## PC Update Logic

### New PC

- Select next value of PC



- 45 -

CS:APP3e

## PC Update

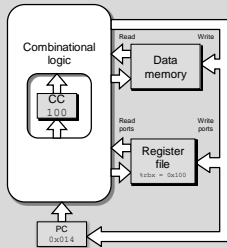
PC update	OPq rA, rB $PC \leftarrow valP$	Update PC
PC update	xmmovq rA, D(rB) $PC \leftarrow valP$	Update PC
PC update	popq rA $PC \leftarrow valP$	Update PC
PC update	jXX Dest $PC \leftarrow Cnd ? valC : valP$	Update PC
PC update	call Dest $PC \leftarrow valC$	Set PC to destination
PC update	ret $PC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

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

- 46 -

CS:APP3e

## SEQ Operation



### State

- PC register
  - Cond. Code register
  - Data memory
  - Register file
- All updated as clock rises*

### Combinational Logic

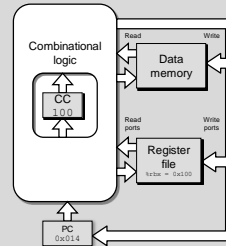
- ALU
- Control logic
- Memory reads
  - Instruction memory
  - Register file
  - Data memory

– 47 –

CS:APP3e

## 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



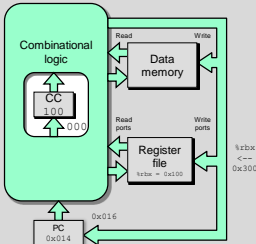
- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

– 48 –

CS:APP3e

## 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



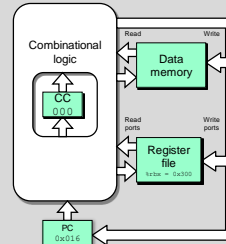
- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction

– 49 –

CS:APP3e

## 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



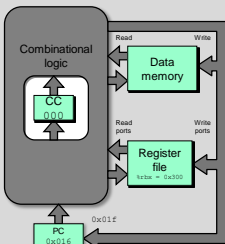
- state set according to `addq` instruction
- combinational logic starting to react to state changes

– 50 –

CS:APP3e

## 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

– 51 –

CS:APP3e

## 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

– 52 –

CS:APP3e