# Intro to Microarchitecture: Single-Cycle

**CS 3330**

Samira Khan
University of Virginia
Feb 14, 2017

## AGENDA

- Review from last lecture
- Single-cycle microarchitecture
- Evaluation of single-cycle microarchitecture

2

## Recap of Last Week

- ISA basics and tradeoffs
  - Instruction length
  - Uniform vs. non-uniform decode
  - Number of registers
  - Addressing modes
  - RISC vs. CISC properties

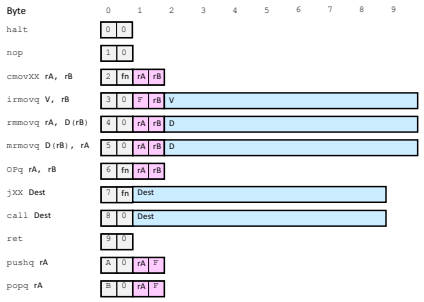
3

## A Note on RISC vs. CISC

- Usually, …
- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes
- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes

4

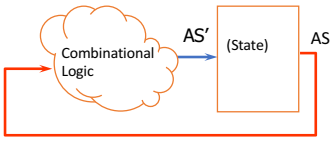
## Y86-64 Instruction Set #1




5

## A Very Basic Instruction Processing Engine

- Single-cycle machine




- What is the *clock cycle time* determined by?
- What is the *critical path* of the combinational logic determined by?

6

## Instruction Processing "Stage"

- Instructions are processed under the direction of a "control unit" step by step.
- Instruction stage: Sequence of steps to process an instruction
- Fundamentally, there are five phases:

- Fetch
- Decode
- Execute
- Memory
- Store Result

- Not all instructions require all stages

7

## Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as the one during which data signals are operated on
  - Everything related to an instruction happens in one clock cycle (serialized processing)

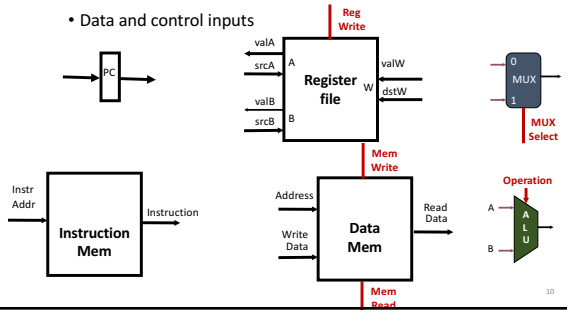
- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the current cycle
  - Latency of control processing can be overlapped with latency of datapath operation (more parallelism)

8

# A Single-Cycle Microarchitecture
*A Closer Look*

## Let's Start with the State Elements

- Data and control inputs

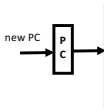



10

## Instruction Processing

- 6 (5) generic steps
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)
  - PC Update

IF

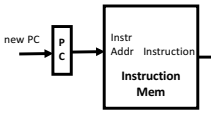



11

## Instruction Processing

- 6 (5) generic steps
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)
  - PC Update

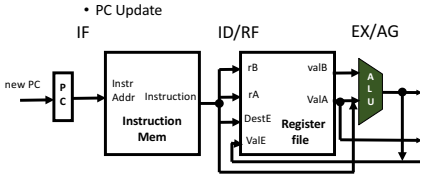
IF




12

## Instruction Processing

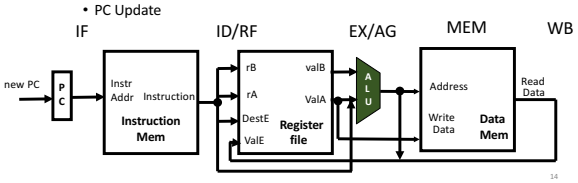
- 6 (5) generic steps
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)
  - PC Update




13

## Instruction Processing

- 6 (5) generic steps
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)
  - PC Update




14

## Single-Cycle Datapath for *Arithmetic and Logical Instructions*

## Executing Arith./Logical Operation

`OPq rA, rB`

| 6 | fn | rA | rB |
|---|---|---|---|

- 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

**if MEM[PC] == `OPq` rA, rB**
**R[rB] ← R[rB] op R[rA]**
**PC ← PC + 2**

16

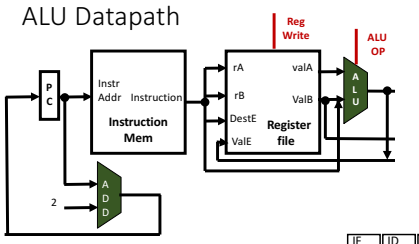
## Stage Computation: Arith/Log. Ops

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

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

17

## ALU Datapath



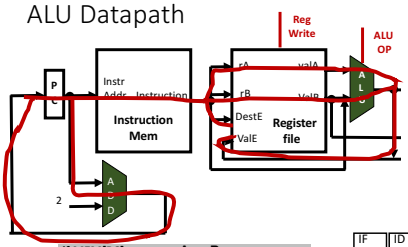

**if MEM[PC] == OPq rA, rB**
**R[rB] ← R[rB] op R[rA]**
**PC ← PC + 2**

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

18

## ALU Datapath




**if MEM[PC] == OPq rA, rB**
**R[rB] ← R[rB] op R[rA]**
**PC ← PC + 2**

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

19

## Single-Cycle Datapath for *Data Movement Instructions*

## Executing `mrmovq` (Load from Mem to Reg)

`mrmovq D (rB) ,rA`

| 6 | fn | rA | rB | D |
|---|---|---|---|---|

- Fetch
  - Read 10 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Read from memory
- Write back
  - Write to Register
- PC Update
  - Increment PC by 10

```
if MEM[PC]== mrmovq Disp (rB), rA
    EA = Disp + R[rB]
    R[rA] ← MEM[EA]
    PC ← PC + 10
```

21

## Stage Computation: `mrmovq`

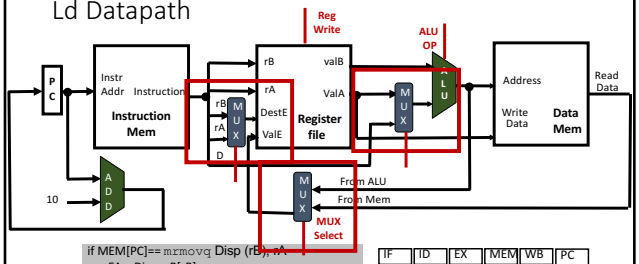
| | `mrmovq` D(rB), rA | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC] | Read instruction byte |
| | rA:rB ← $M_1$[PC+1] | Read register byte |
| | valC ← $M_8$[PC+2] | Read displacement D |
| | valP ← PC+10 | Compute next PC |
| Decode | valB ← R[rB] | Read operand B |
| Execute | valE ← valB + valC | Compute effective address |
| Memory | valM ← $M_8$[valE] | Write value to memory |
| Write back | R[rA] ← valM | |
| PC update | PC ← valP | Update PC |

- Use ALU for address computation

```
if MEM[PC]== mrmovq Disp (rB), rA
    EA = Disp + R[rB]
    R[rA] ← MEM[EA]
    PC ← PC + 10
```

22

## Ld Datapath



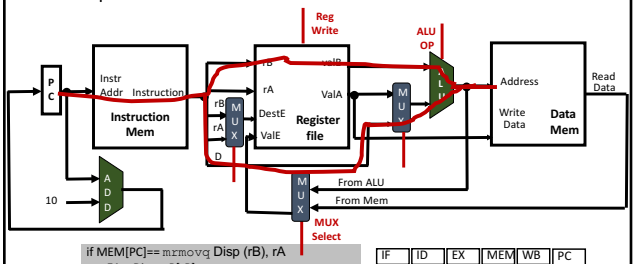

```
if MEM[PC]== mrmovq Disp (rB), rA
    EA = Disp + R[rB]
    R[rA] ← MEM[EA]
    PC ← PC + 10
```

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

23

## Ld Datapath: Calculate the Effective Address



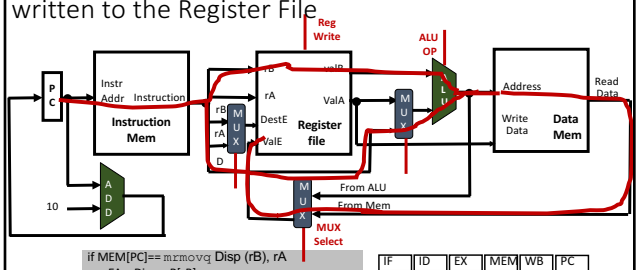

```
if MEM[PC]== mrmovq Disp (rB), rA
    EA = Disp + R[rB]
    R[rA] ← MEM[EA]
    PC ← PC + 10
```

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

24

## Ld Datapath: Loaded memory needs to be written to the Register File



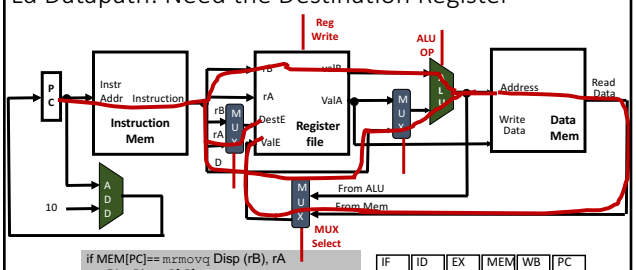

if MEM[PC]== `mrmovq` Disp (rB), rA
EA = Disp + R[rB]
R[rA] ← MEM[EA]
PC ← PC + 10

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

25

## Ld Datapath: Need the Destination Register



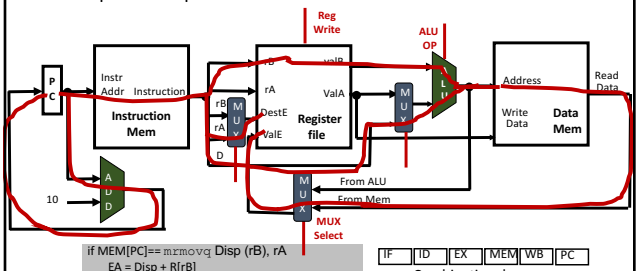

if MEM[PC]== `mrmovq` Disp (rB), rA
EA = Disp + R[rB]
R[rA] ← MEM[EA]
PC ← PC + 10

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

26

## Ld Datapath: Update PC




if MEM[PC]== `mrmovq` Disp (rB), rA
EA = Disp + R[rB]
R[rA] ← MEM[EA]
PC ← PC + 10

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

27

## Executing `rmmovq` (St from reg to Memory)

`rmmovq` rA, D (rB)

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

- Fetch
  - Read 10 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
- Write back
  - Do nothing
- PC Update
  - Increment PC by 10

if MEM[PC]== `rmmovq` rA, Disp (rB)
EA = Disp + R[rB]
MEM[EA] ← R[rA]
PC ← PC + 10

28

## Stage Computation: `rmmovq`

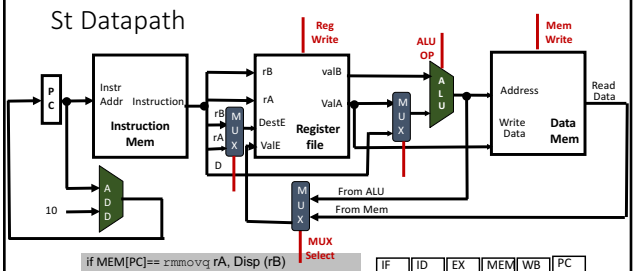
| | `rmmovq` rA, D(rB) | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC] | Read instruction byte |
| | rA:rB ← $M_1$[PC+1] | Read register byte |
| | valC ← $M_8$[PC+2] | Read displacement D |
| | valP ← PC+10 | Compute next PC |
| Decode | valA ← R[rA] | Read operand A |
| | valB ← R[rB] | Read operand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

- Use ALU for address computation

if MEM[PC]== `rmmovq` rA, Disp (rB)
EA = Disp + R[rB]
MEM[EA] ← R[rA]
PC ← PC + 10

29

## St Datapath



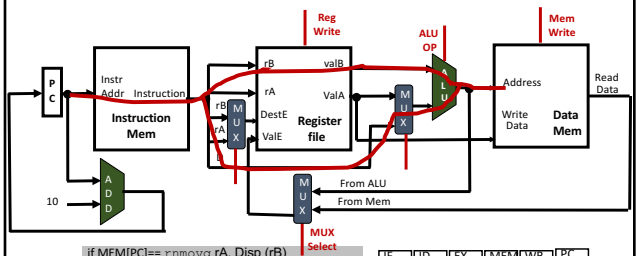

if MEM[PC]== `rmmovq` rA, Disp (rB)
EA = Disp + R[rB]
MEM[EA] ← R[rA]
PC ← PC + 10

IF | ID | EX | MEM | WB | PC

Combinational state update logic

30

## St Datapath: Calculate the effective address



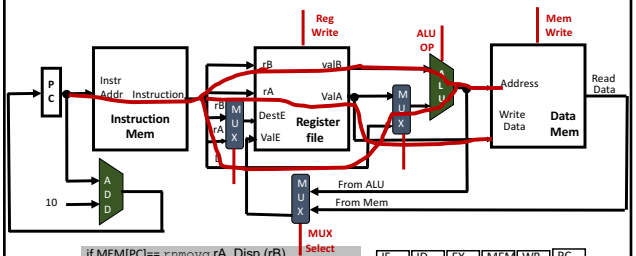

if MEM[PC]== `rnmovq` rA, Disp (rB)
EA = Disp + R[rB]
MEM[EA] ← R[rA]
PC ← PC + 10

IF | ID | EX | MEM | WB | PC

Combinational state update logic

31

## St Datapath: Store data needs to reach memory



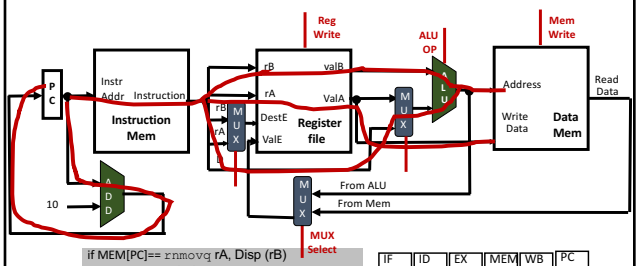

if MEM[PC]== `rnmovq` rA, Disp (rB)
EA = Disp + R[rB]
MEM[EA] ← R[rA]
PC ← PC + 10

IF | ID | EX | MEM | WB | PC

Combinational state update logic

32

## St Datapath: Update PC




if MEM[PC]== `rnmovq` rA, Disp (rB)
EA = Disp + R[rB]
MEM[EA] ← R[rA]
PC ← PC + 10

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

33

## Executing `irmovq` (Move imm to Reg)

`irmovq` V, rB

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

- Fetch
  - Read 10 bytes
- Decode
  - Read operand registers
- Execute
  - Add 0 to V
- Memory
  - Do nothing
- Write back
  - Write V to rB
- PC Update
  - Increment PC by 10

if MEM[PC]== `irmovq` V, rB
R[rB] ← V + 0
PC ← PC + 10

34

## Stage Computation: `irmovq`

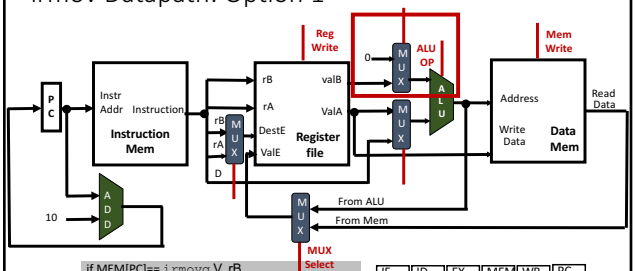
| | `irmovq` V, rB | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC] | Read instruction byte |
| | rA:rB ← $M_1$[PC+1] | Read register byte |
| | valC ← $M_8$[PC+2] | Read displacement D |
| | valP ← PC+10 | Compute next PC |
| Decode | | |
| Execute | valE ← 0 + valC | Compute effective address |
| | R[rB] ← valA | |
| Write back | | |
| PC update | PC ← valP | Update PC |

- Use ALU for address computation

if MEM[PC]== `irmovq` V, rB
R[rB] ← V + 0
PC ← PC + 10

35

## irmov Datapath: Option 1



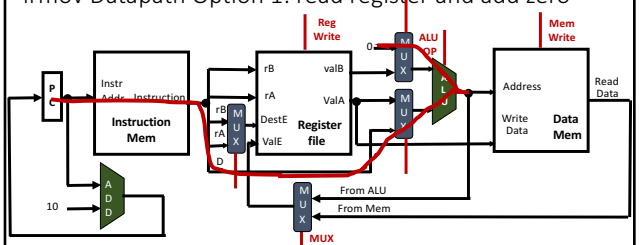

if MEM[PC]== `irmovq` V, rB
R[rB] ← V + 0
PC ← PC + 10

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

36

## irmov Datapath Option 1: read register and add zero



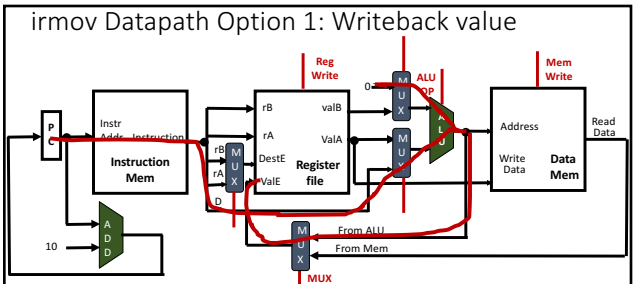

if MEM[PC]== `irmovq` V, rB
R[rB] ← V + 0
PC ← PC + 10

IF | ID | EX | MEM | WB | PC

Combinational state update logic

37

## irmov Datapath Option 1: Writeback value



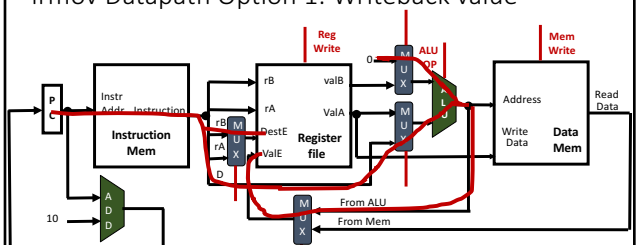

if MEM[PC]== `irmovq` V, rB
R[rB] ← V + 0
PC ← PC + 10

IF | ID | EX | MEM | WB | PC

Combinational state update logic

38

## irmov Datapath Option 1: Writeback value



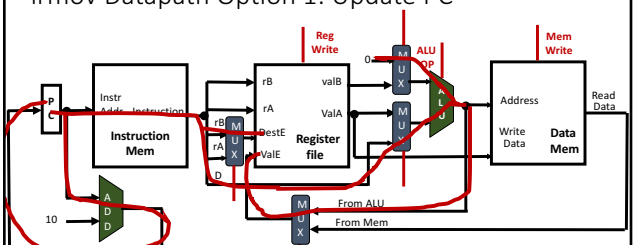

if MEM[PC]== `irmovq` V, rB
R[rB] ← V + 0
PC ← PC + 10

IF | ID | EX | MEM | WB | PC

Combinational state update logic

39

## irmov Datapath Option 1: Update PC



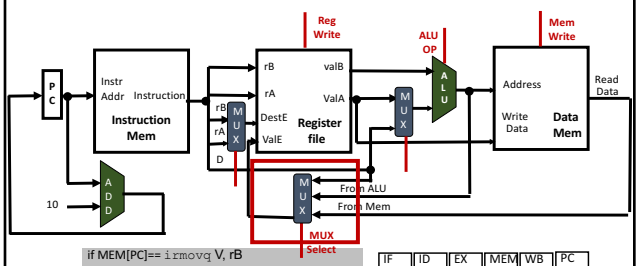

if MEM[PC]== `irmovq` V, rB
R[rB] ← V + 0
PC ← PC + 10

IF | ID | EX | MEM | WB | PC

Combinational state update logic

40

## irmov Datapath Option 2



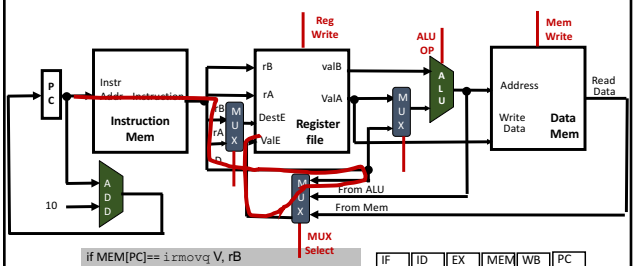

if MEM[PC]== `irmovq` V, rB
R[rB] ← V
PC ← PC + 10

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

41

## irmov Datapath: Option 2




if MEM[PC]== `irmovq` V, rB
R[rB] ← V
PC ← PC + 10

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

42

- Tradeoffs between option 1 and option 2?
  - Extra 2-to-1 MUX?
  - 3-to-1 MUX? What is the cost for n-to-1 MUX?
  - Wire length?
  - Generality?

43

## Executing `rrmovq` (Move from Reg to Reg)

`rrmovq` rA, rB

| 2 | 0 | rA | rB |
|---|---|---|---|

- Fetch
  - Read 2 bytes
- Decode
  - Read operand register rA
- Execute
  - Add 0 to val rA
- Memory
  - Do nothing
- Write back
  - Write val rA to rB
- PC Update
  - Increment PC by 2

if MEM[PC]== `rrmovq` rA, rB
R[rB] ← R[rA] + 0
PC ← PC + 2

44

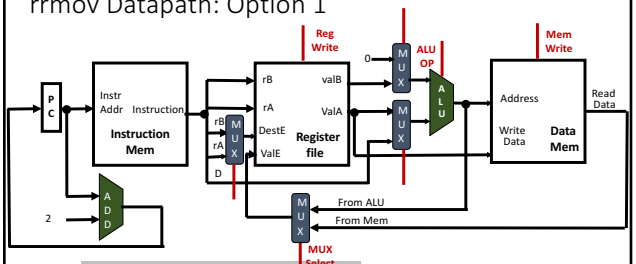
## Stage Computation: `rrmovq`

| | `rrmovq` rA, rB | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC] | Read instruction byte |
| | rA:rB ← $M_1$[PC+1] | Read register byte |
| | | Read displacement D |
| | valP ← PC+2 | Compute next PC |
| Decode | ValA ← R[rA] | |
| Execute | valE ← 0 + valA | Compute effective address |
| Write back | R[rB] ← valE | |
| PC update | PC ← valP | Update PC |

if MEM[PC]== `rrmovq` rA, rB
R[rB] ← R[rA] + 0
PC ← PC + 2

45

## rrmov Datapath: Option 1



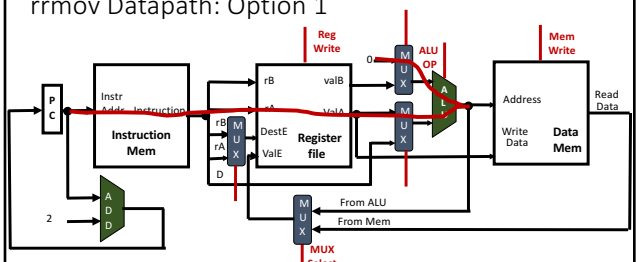

if MEM[PC]== `rrmovq` rA, rB
R[rB] ← R[rA] + 0
PC ← PC + 2

IF | ID | EX | MEM | WB | PC

Combinational state update logic

46

## rrmov Datapath: Option 1



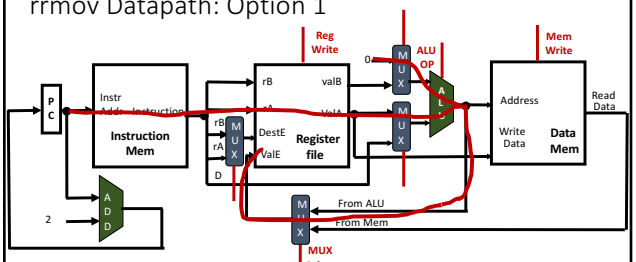

if MEM[PC]== `rrmovq` rA, rB
R[rB] ← R[rA] + 0
PC ← PC + 2

IF | ID | EX | MEM | WB | PC

Combinational state update logic

47

## rrmov Datapath: Option 1



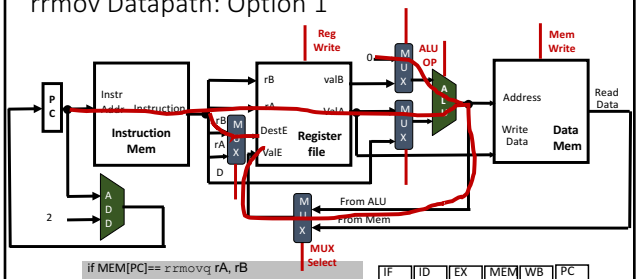

if MEM[PC]== `rrmovq` rA, rB
R[rB] ← R[rA] + 0
PC ← PC + 2

IF | ID | EX | MEM | WB | PC

Combinational state update logic

48

## rrmov Datapath: Option 1



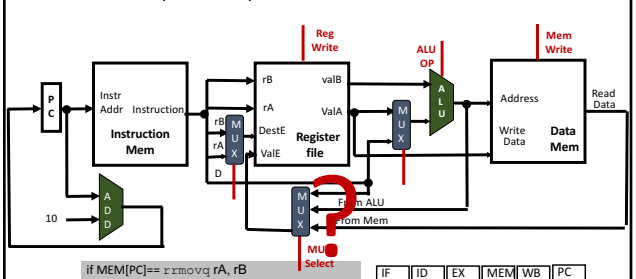

if MEM[PC]== `rrmovq` rA, rB
R[rB] ← R[rA] + 0
PC ← PC + 2

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational
state update logic

49

## rrmov Datapath: Option 2




if MEM[PC]== `rrmovq` rA, rB
R[rB] ← R[rA]
PC ← PC + 2

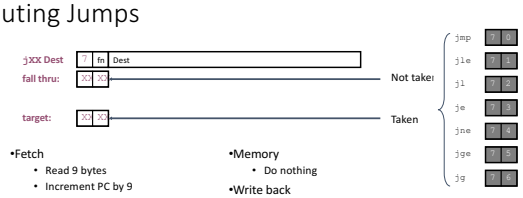
| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational
state update logic

50

# Single-Cycle Datapath for *Control Flow Instructions*

## 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

if MEM[PC]== `jmp` Dest
if (cond) PC ← Dest else PC ← PC + 9

52

## Stage Computation: Jumps

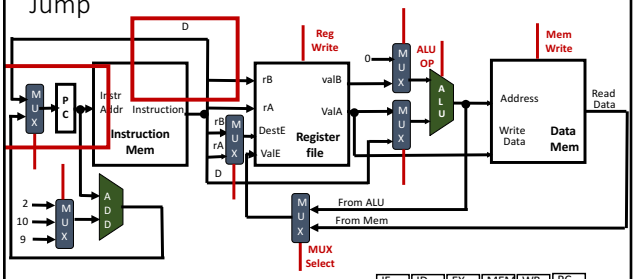
| | jXX Dest | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC]<br>valC ← $M_8$[PC+1]<br>valP ← PC+9 | Read instruction byte<br>Read destination address<br>Fall through address |
| Decode | | |
| Execute | Cnd ← Cond(CC,ifun) | Take branch? |
| Memory | | |
| Write back | | |
| PC update | PC ← Cnd ?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

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

if MEM[PC]== `jmp` Dest
if (cond) PC ← Dest else PC ← PC + 9

53

## Jump



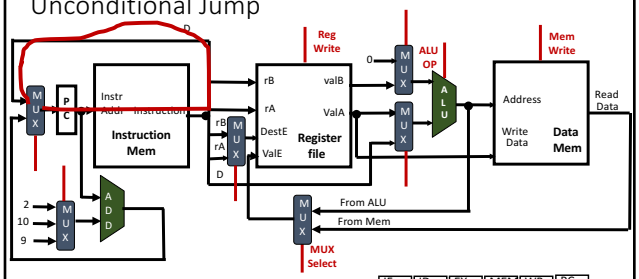

if MEM[PC]== `jmp` Dest
PC ← Dest

IF | ID | EX | MEM | WB | PC

Combinational state update logic

54

## Unconditional Jump



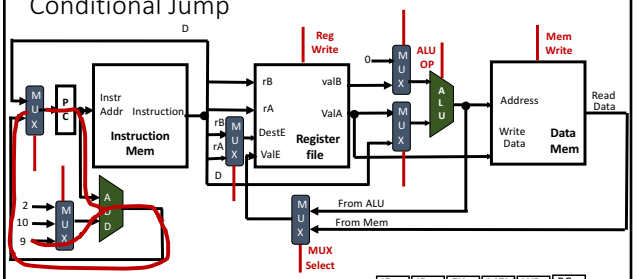

if MEM[PC]== `jmp` Dest
PC ← Dest

IF | ID | EX | MEM | WB | PC

Combinational state update logic

55

## Conditional Jump



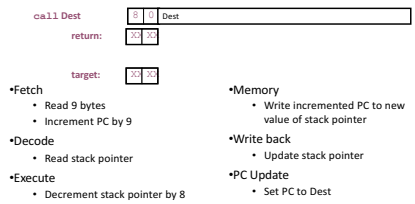

if MEM[PC]== `jmp` Dest
if (cond) PC ← Dest else PC ← PC + 9

IF | ID | EX | MEM | WB | PC

Combinational state update logic

56

## 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

if MEM[PC]== `call` Dest
MEM [R[rsp]-8] ← PC + 9
R[rsp] ← R[rsp] - 8
PC ← Dest

57

## Stage Computation: `call`

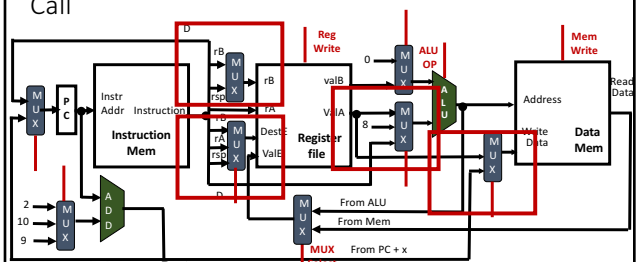
| | `call` Dest | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC] | Read instruction byte |
| | valC ← $M_8$[PC+1] | Read destination address |
| | valP ← PC+9 | Compute return point |
| Decode | valB ← R[%rsp] | Read stack pointer |
| Execute | valE ← valB + −8 | Decrement stack pointer |
| Memory | $M_8$[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

if MEM[PC]== `call` Dest
MEM [R[rsp]-8] ← PC + 9
R[rsp] ← R[rsp] - 8
PC ← Dest

58

## Call



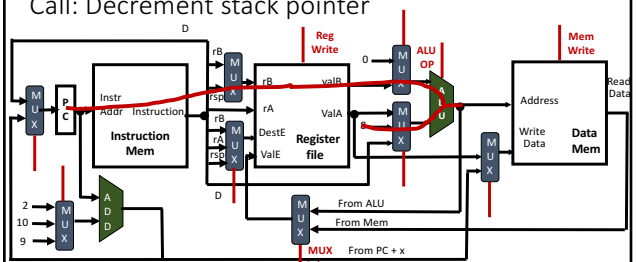

if MEM[PC]== `call` Dest
MEM [R[rsp]-8] ← PC + 9
R[rsp] ← R[rsp] - 8
PC ← Dest

IF | ID | EX | MEM | WB | PC

Combinational state update logic

59

## Call: Decrement stack pointer



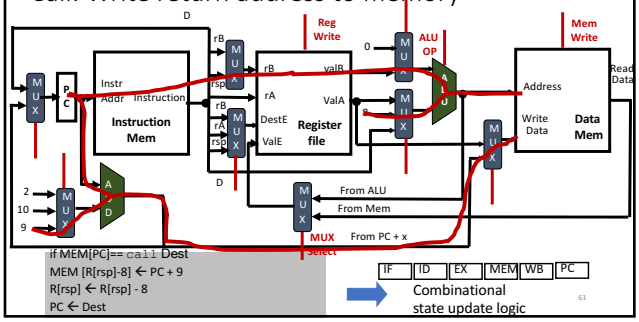

if MEM[PC]== `call` Dest
MEM [R[rsp]-8] ← PC + 9
R[rsp] ← R[rsp] - 8
PC ← Dest

IF | ID | EX | MEM | WB | PC

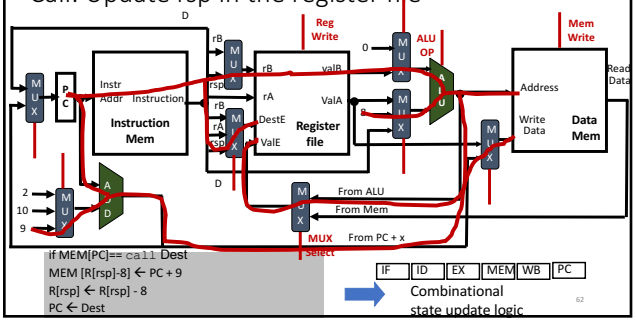
Combinational state update logic

60

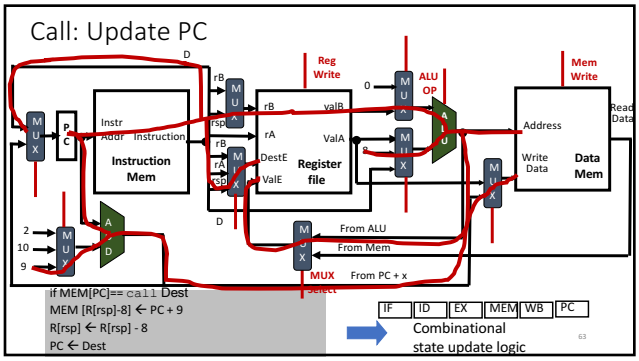
## Call: Write return address to memory




## Call: Update rsp in the register file




## Call: Update 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

if MEM[PC]== `ret`
PC ← MEM [R[rsp]]
R[rsp] ← R[rsp] + 8

64

## Stage Computation: `ret`

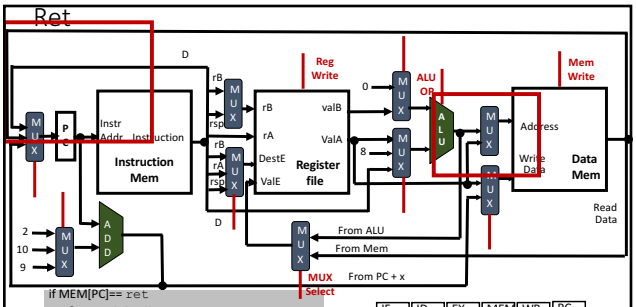
| | `ret` | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC] | Read instruction byte |
| Decode | valA ← R[%rsp]<br>valB ← R[%rsp] | Read operand stack pointer<br>Read operand stack pointer |
| Execute | valE ← valB + 8 | Increment stack pointer |
| Memory | valM ← $M_8$[valA] | Read return address |
| 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

if MEM[PC]== `ret`
PC ← MEM [R[rsp]]
R[rsp] ← R[rsp] + 8

65

## Ret



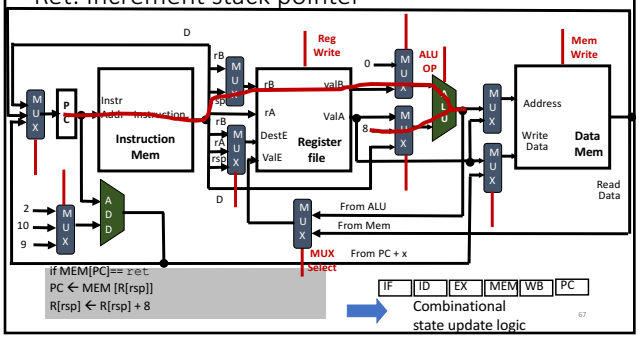

if MEM[PC]== `ret`
PC ← MEM [R[rsp]]
R[rsp] ← R[rsp] + 8

IF | ID | EX | MEM | WB | PC

Combinational state update logic

66

## Ret: Increment stack pointer



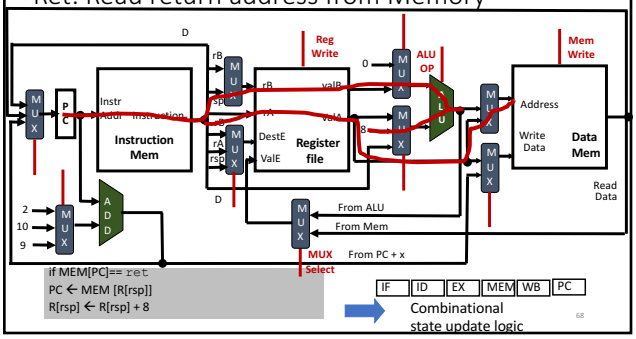

if MEM[PC]== `ret`
PC ← MEM [R[rsp]]
R[rsp] ← R[rsp] + 8

IF | ID | EX | MEM | WB | PC

Combinational state update logic

67

## Ret: Read return address from Memory



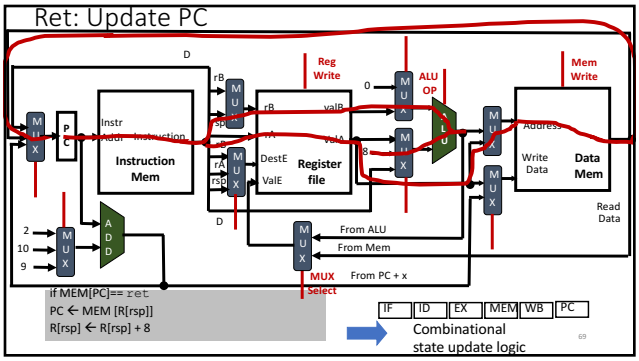

if MEM[PC]== `ret`
PC ← MEM [R[rsp]]
R[rsp] ← R[rsp] + 8

IF | ID | EX | MEM | WB | PC

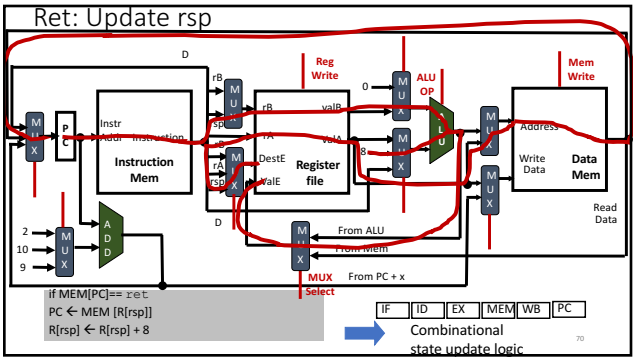
Combinational state update logic

68

## Ret: Update PC




## Ret: Update rsp




# Single-Cycle Datapath for *Stack Operations*

## Executing popq

popq rA

| B | 0 | rA | 8 |
|---|---|---|---|

- 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

if MEM[PC]== pop
R[rA] ← MEM [R[rsp]]
R[rsp] ← R[rsp] + 8

72

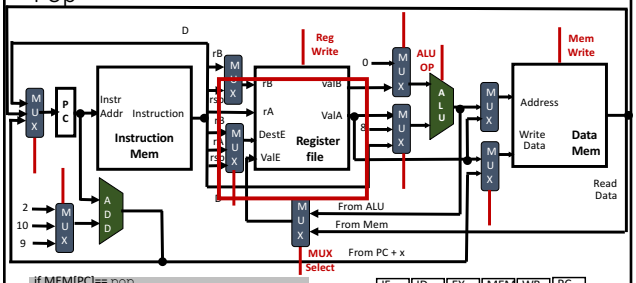
## Stage Computation: popq

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

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

if MEM[PC]== pop
R[rA] ← MEM [R[rsp]]
R[rsp] ← R[rsp] + 8

## Pop



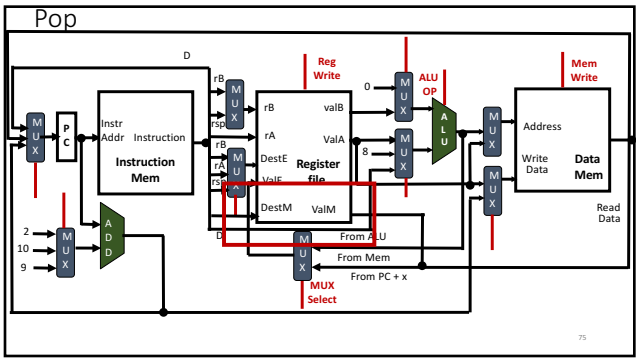

if MEM[PC]== pop
R[rA] ← MEM [R[rsp]]
R[rsp] ← R[rsp] + 8

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

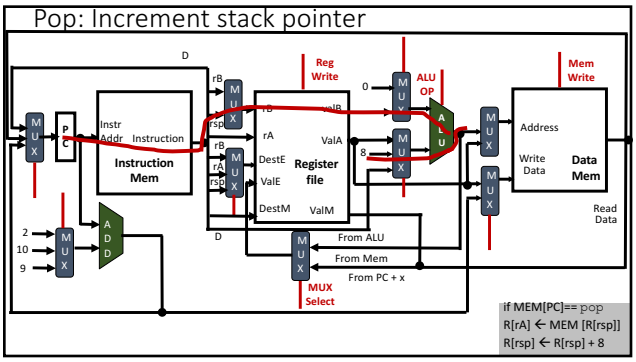
74

## Pop



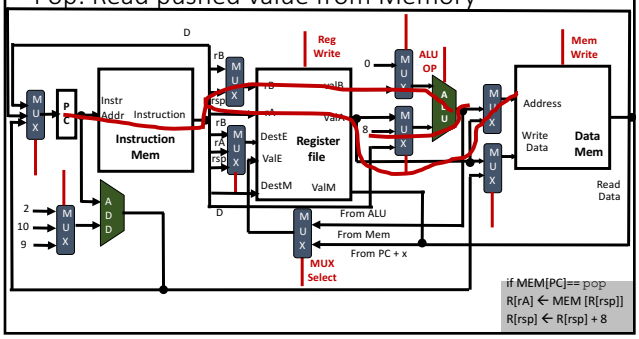

75

## Pop: Increment stack pointer

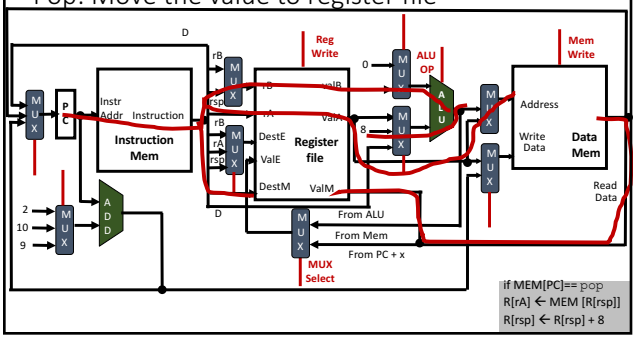



if MEM[PC]== pop
R[rA] ← MEM [R[rsp]]
R[rsp] ← R[rsp] + 8

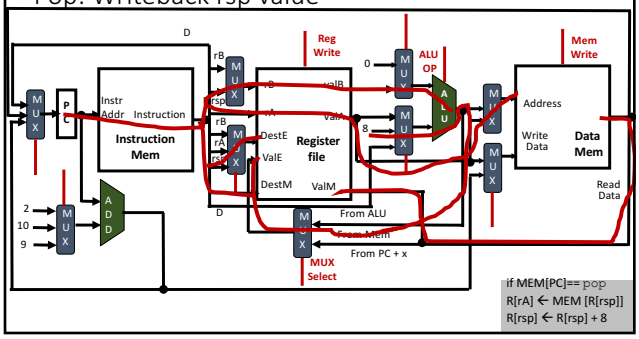
## Pop: Read pushed value from Memory



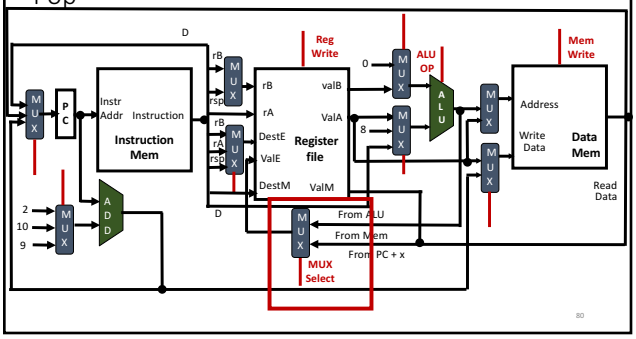

## Pop: Move the value to register file



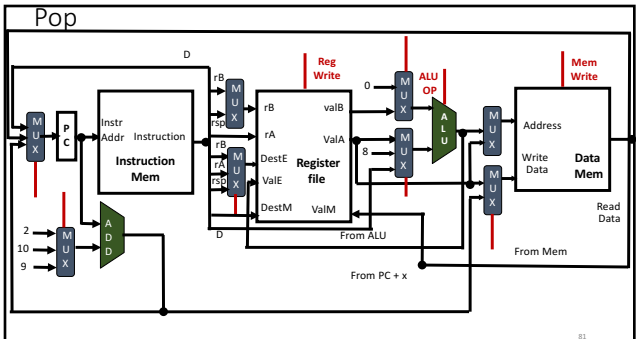

## Pop: Writeback rsp value




## Pop



## Pop




81

# Evaluating the Single-Cycle Microarchitecture

## A Single-Cycle Microarchitecture

- Is *this* a good idea/design?
- When is this a good design?
- When is this a bad design?
- How can we design a better microarchitecture?

83

## A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1
- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute
- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction

84

## What is the Slowest Instruction to Process?

- Let's go back to the basics
- All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

- Do each of the above phases take the same time (latency) for all instructions?

85

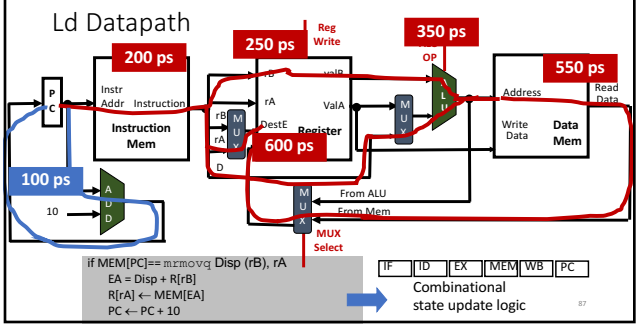
## Single-Cycle Datapath Analysis

- Assume
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

| steps | IF | ID | EX | MEM | WB | Delay |
|---|---|---|---|---|---|---|
| resources | mem | RF | ALU | mem | RF | |
| Reg-Reg Op | 200 | 50 | 100 | | 50 | 400 |
| ir mov | 200 | 50 | 100 | | 50 | 400 |
| LW | 200 | 50 | 100 | 200 | 50 | 600 |
| SW | 200 | 50 | 100 | 200 | | 550 |
| Call | 200 | 50 | 100 | 200 | 50 | 600 |
| Jump | 200 | | | | | 200 |

86

## Ld Datapath




if MEM[PC]== `mrmovq` Disp (rB), rA
EA = Disp + R[rB]
R[rA] ← MEM[EA]
PC ← PC + 10

| IF | ID | EX | MEM | WB | PC |
|---|---|---|---|---|---|

Combinational state update logic

87

## Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction
- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle
- Not necessarily the simplest way to implement an ISA
  - Single-cycle implementation of REP MOVS (x86) or INDEX (VAX)?
- Not easy to optimize/improve performance
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time

88

## (Micro)architecture Design Principles

- Critical path design
  - Find and decrease the maximum combinational logic delay
  - Break a path into multiple cycles if it takes too long
- Bread and butter (common case) design
  - Spend time and resources on where it matters most
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case
- Balanced design
  - Balance instruction/data flow through hardware components
  - Design to eliminate bottlenecks: balance the hardware for the work

89

## Single-Cycle Design vs. Design Principles

- Critical path design
- Bread and butter (common case) design
- Balanced design

*How does a single-cycle microarchitecture fare in light of these principles?*

90

# Intro to Microarchitecture: Single-Cycle

**CS 3330**

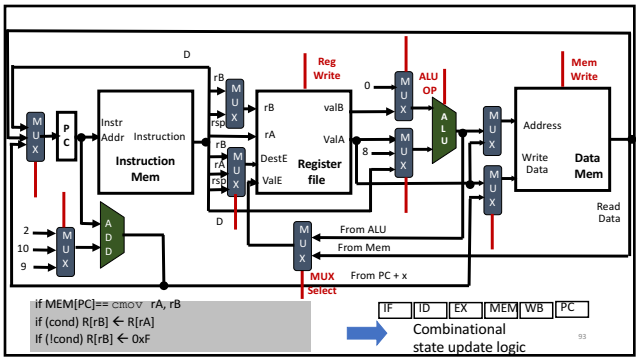
**Samira Khan**
University of Virginia
Feb 14, 2017

## Stage Computation: Cond. Move

| | cmovXX rA, rB | |
|---|---|---|
| Fetch | icode:ifun ← $M_1$[PC] | Read instruction byte |
| | rA:rB ← $M_1$[PC+1] | Read register byte |
| | valP ← PC+2 | Compute next PC |
| Decode | valA ← R[rA] | Read operand A |
| | valB ← 0 | |
| Execute | valE ← valB + valA | Pass valA through ALU |
| | If ! Cond(CC,ifun) rB ← 0xF | (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

if MEM[PC]== `cmov` rA, rB
if (cond) R[rB] ← R[rA]
If (!cond) R[rB] ← 0xF


D
Reg Write
ALU OP
Mem Write
MUX
PC
Instr Addr
Instruction
Instruction Mem
rB
rsp
rA
DestE
ValE
valB
ValA
Register file
0
8
ALU
Address
Write Data
Data Mem
Read Data
2
10
9
ADD
From ALU
From Mem
From PC + x
MUX Select
if MEM[PC]== cmov rA, rB
if (cond) R[rB] ← R[rA]
If (!cond) R[rB] ← 0xF
IF
ID
EX
MEM
WB
PC
Combinational state update logic
93