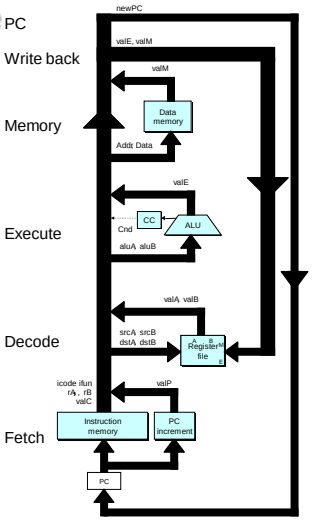
SEQ Hardware Structure

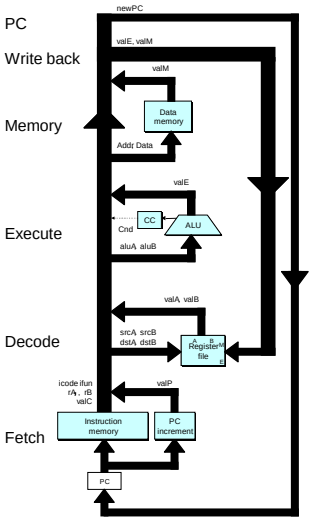
- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter



61

SEQ Stages

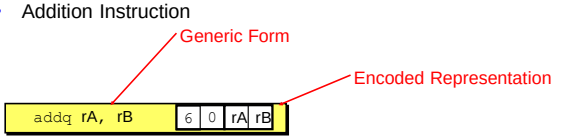
- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



62

Instruction Example

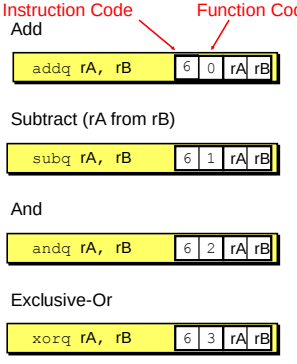
- Addition Instruction



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

63

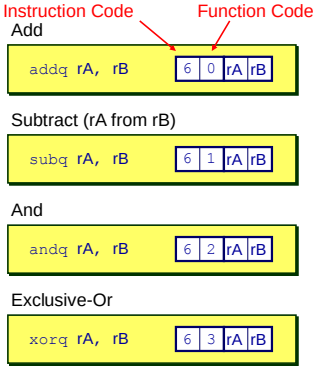
Arithmetic and Logical Operations



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

64

Arithmetic and Logical Operations



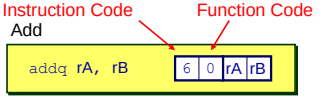
- Refer to generically as "OPq"
- Encodings differ only by "function code"
  - Low-order 4 bits in first instruction word
- Set condition codes as side effect

65

65

Executing an ADD instruction

- How does the processor execute `addq %rax,%rsi`
- The binary encoding is `60 06`

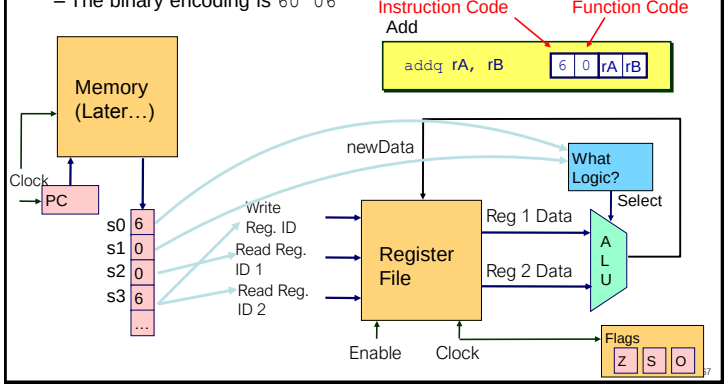


66

66

Executing an ADD instruction

- How does the processor execute `addq %rax,%rsi`
- The binary encoding is `60 06`



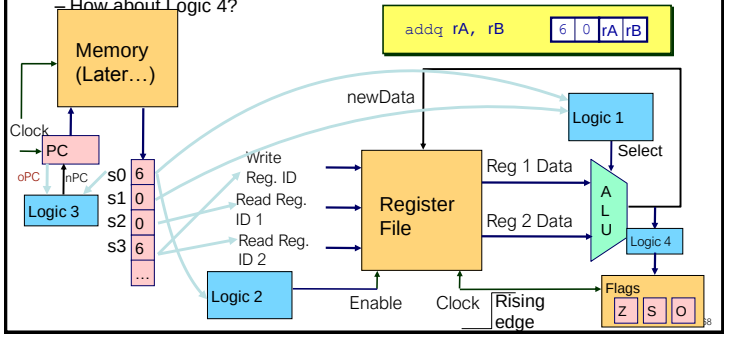
67

67

Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?

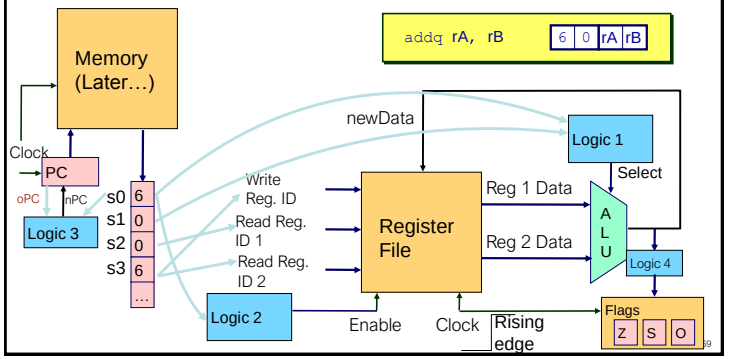


68

68

Executing an ADD instruction

– When the rising edge of the clock arrives, the RF/PC/Flags will be written.
– So the following has to be ready: newData, nPC, which means Logic1, Logic2, Logic3, and Logic4 has to finish.



69

Jump Instructions

Jump (Conditionally)

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

70

Jump Instructions

Jump Unconditionally

Jump When Less or Equal

Jump When Less

Jump When Equal

Jump When Not Equal

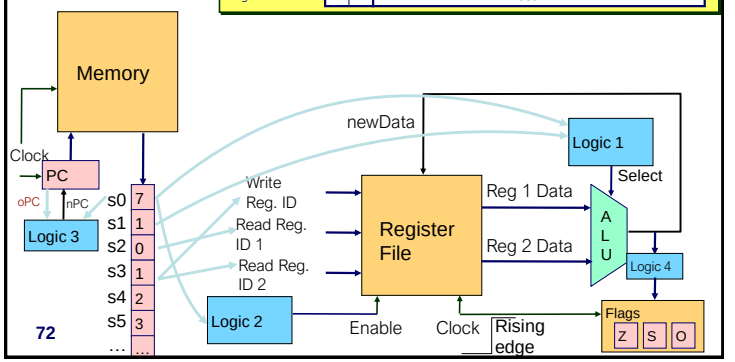
Jump When Greater or Equal

Jump When Greater

71

Executing a JLE instruction

– Let's say the binary encoding for `jle .L0` is `71 0123000000000000`
– What are the logics now?

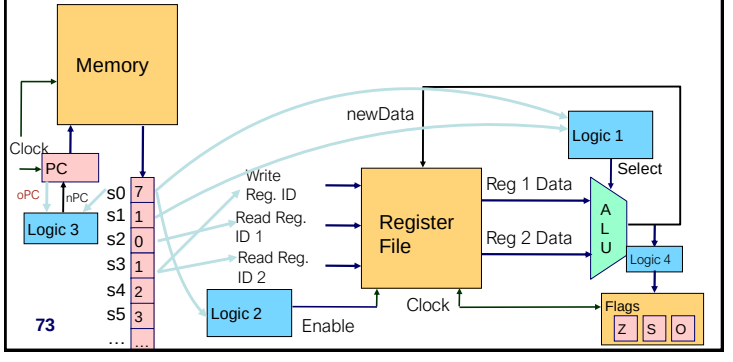


72

72

Executing a JLE instruction

– Logic 1: if (s0 == 6) select = s1;
– Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;

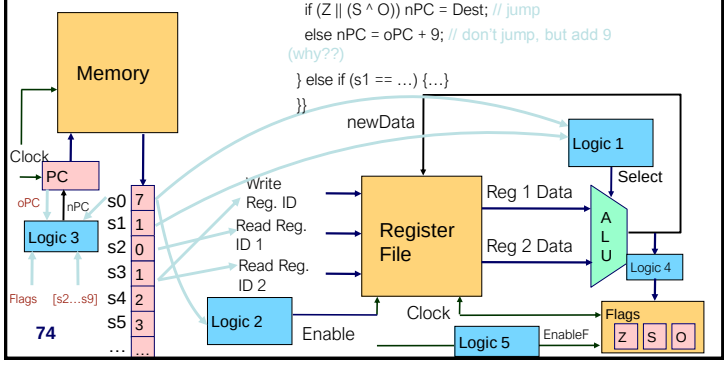


73

Executing a JLE instruction

– Logic 3??

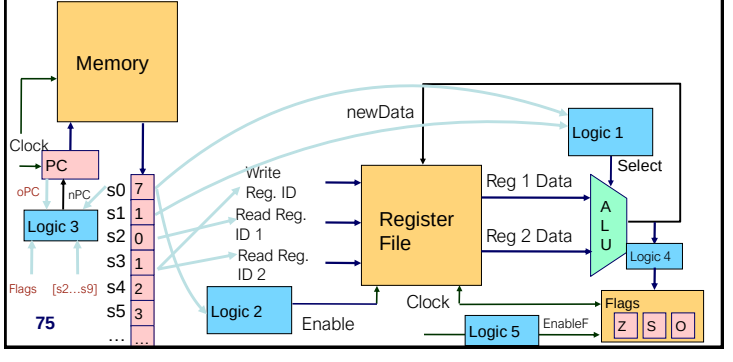
```
if (s0 == 6) nPC = oPC + 2;
else if (s0 == 7) {
  if (s1 == 1) { // jLE
    if (Z || (S ^ O)) nPC = Dest; // jump
    else nPC = oPC + 9; // don't jump, but add 9 (why??)
  } else if (s1 == …) {…}
  }}
```



74

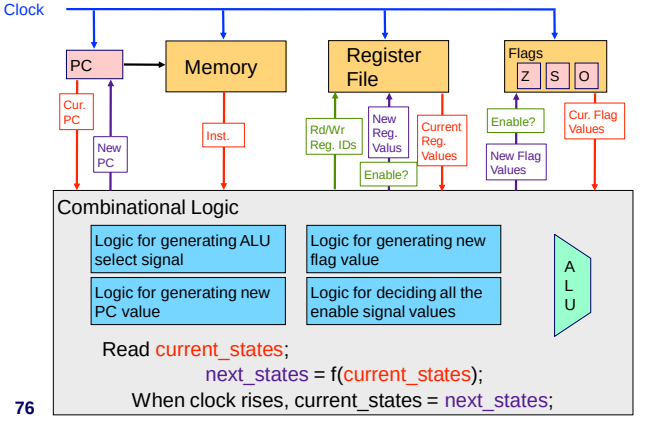
Executing a JLE instruction

– Logic 4? Does JLE write flags?
– Need another piece of logic.
– Logic 5: if (s0 == 7) EnableF = 0; else if (s0 == 6) EnableF = 1;



75

Microarchitecture (So far)



76

Move Operations

Register ➔ Register

Immediate ➔ Register

Register ➔ Memory

Memory ➔ Register

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

77

Move Instruction Examples

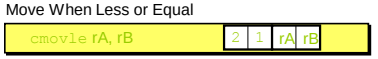
X86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
Encoding: `30 82 cd ab 00 00 00 00 00 00`	
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`
Encoding: `20 43`	
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`
Encoding: `50 15 f4 ff ff ff ff ff ff ff`	
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`
Encoding: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

78

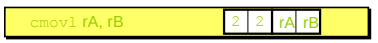
Conditional Move Instructions

Move Unconditionally

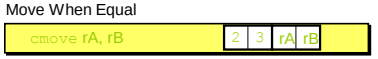
Move When Less or Equal



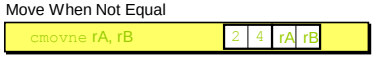
Move When Less



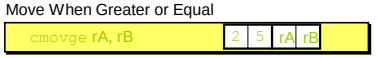
Move When Equal



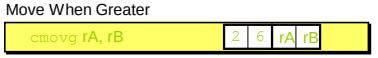
Move When Not Equal



Move When Greater or Equal



Move When Greater

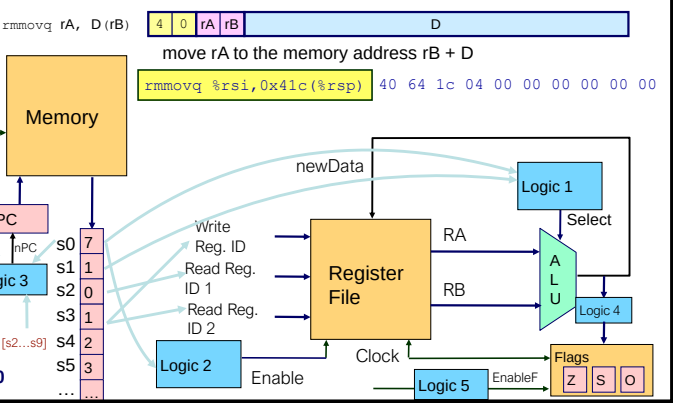


- Refer to generically as "`cmovXX`"
- Encodings differ only by "function code"
- Based on values of condition codes
- Variants of `rrmovq` instruction
  - (Conditionally) copy value from source to destination register

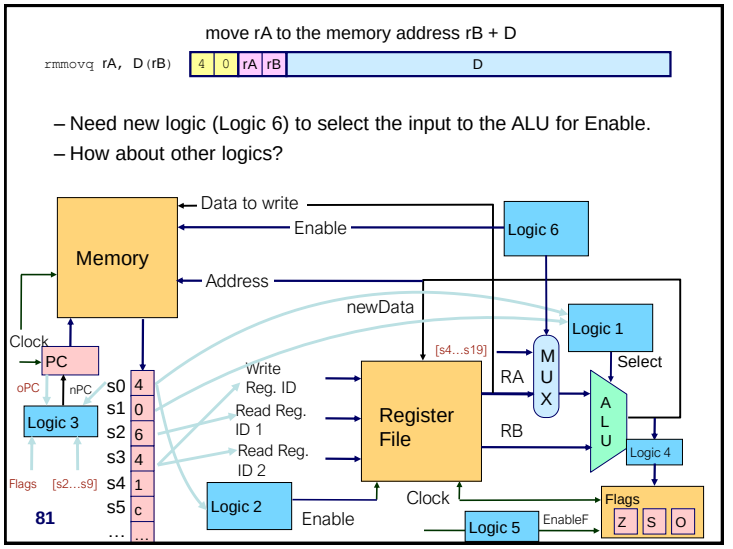
79

Executing a MOV instruction

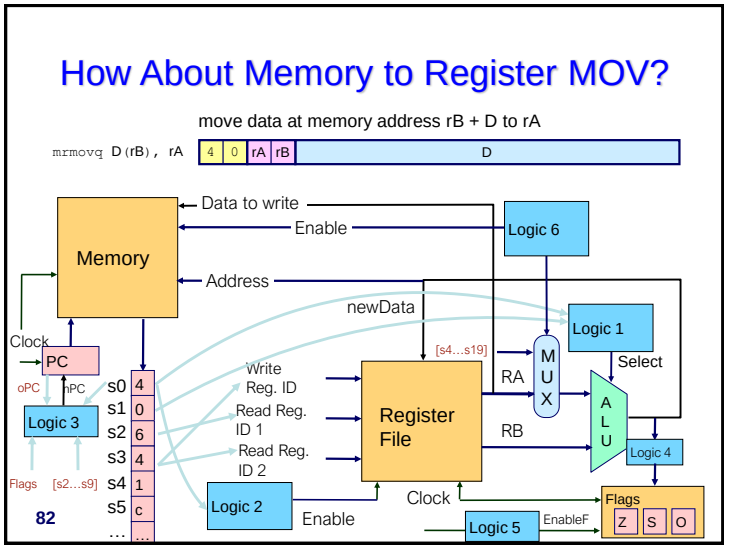
– How do we modify the hardware to execute a move instruction?



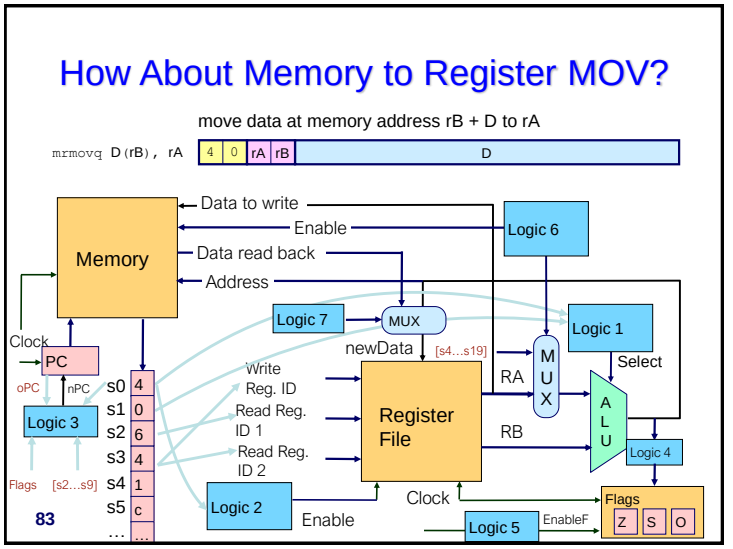
80


move rA to the memory address rB + D
rmmovq rA, D(rB)
4 0 rA rB D
– Need new logic (Logic 6) to select the input to the ALU for Enable.
– How about other logics?
Memory
Data to write
Enable
Logic 6
Address
newData
Logic 1
Select
Clock
PC
oPC
nPC
Logic 3
Flags
[s2…s9]
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
Write Reg. ID
Read Reg. ID 1
Read Reg. ID 2
Register File
[s4…s19]
RA
RB
MUX
ALU
Logic 4
Logic 2
Enable
Clock
Logic 5
EnableF
Flags
Z S O
81

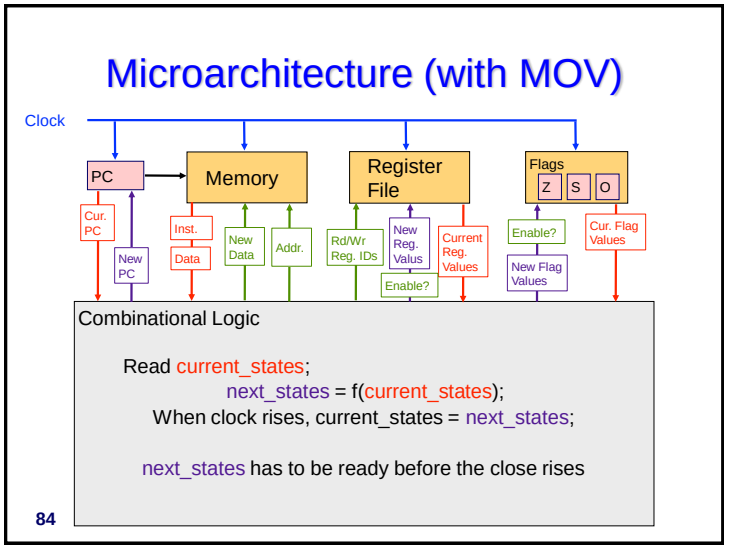
81


How About Memory to Register MOV?
move data at memory address rB + D to rA
mrmovq D(rB), rA
4 0 rA rB D
Memory
Data to write
Enable
Logic 6
Address
newData
Logic 1
Select
Clock
PC
oPC
nPC
Logic 3
Flags
[s2…s9]
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
Write Reg. ID
Read Reg. ID 1
Read Reg. ID 2
Register File
[s4…s19]
RA
RB
MUX
ALU
Logic 4
Logic 2
Enable
Clock
Logic 5
EnableF
Flags
Z S O
82

82


How About Memory to Register MOV?
move data at memory address rB + D to rA
mrmovq D(rB), rA
4 0 rA rB D
Memory
Data to write
Enable
Logic 6
Data read back
Address
Logic 7
MUX
newData
Logic 1
Select
Clock
PC
oPC
nPC
Logic 3
Flags
[s2…s9]
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
Write Reg. ID
Read Reg. ID 1
Read Reg. ID 2
Register File
[s4…s19]
RA
RB
MUX
ALU
Logic 4
Logic 2
Enable
Clock
Logic 5
EnableF
Flags
Z S O
83

83


Microarchitecture (with MOV)
Clock
PC
Cur. PC
New PC
Memory
Inst.
Data
New Data
Addr.
Register File
Rd/Wr Reg. IDs
New Reg. Valus
Enable?
Current Reg. Values
Flags
Z S O
Enable?
New Flag Values
Cur. Flag Values
Combinational Logic
Read current_states;
next_states = f(current_states);
When clock rises, current_states = next_states;
next_states has to be ready before the close rises
84

84

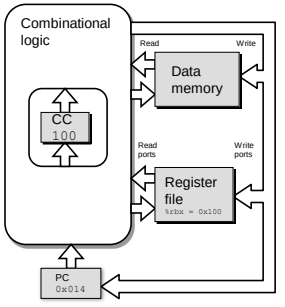
Microarchitecture Overview

Think of it as a state machine

Every cycle, one instruction gets executed. At the end of the cycle, architecture states get modified.

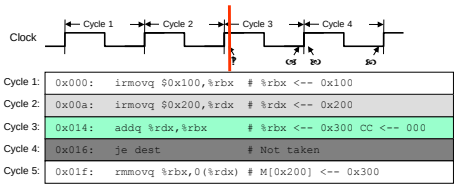
States (All updated as clock rises)

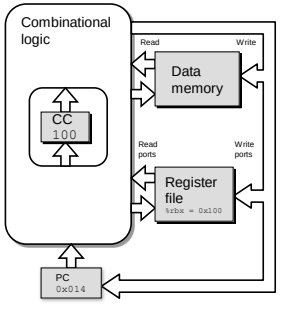
- PC register
- Cond. Code register
- Data memory
- Register file



85

85

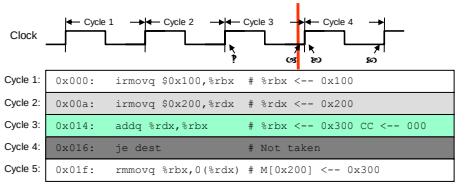


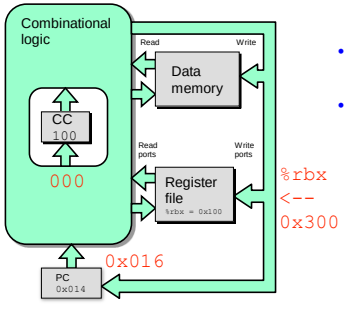


- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

86

86



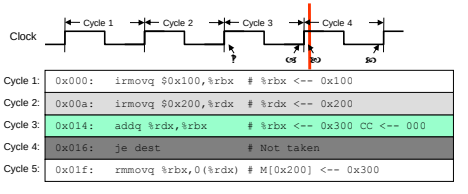


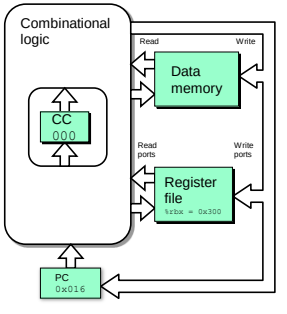
%rbx <-- 0x300

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

87

87

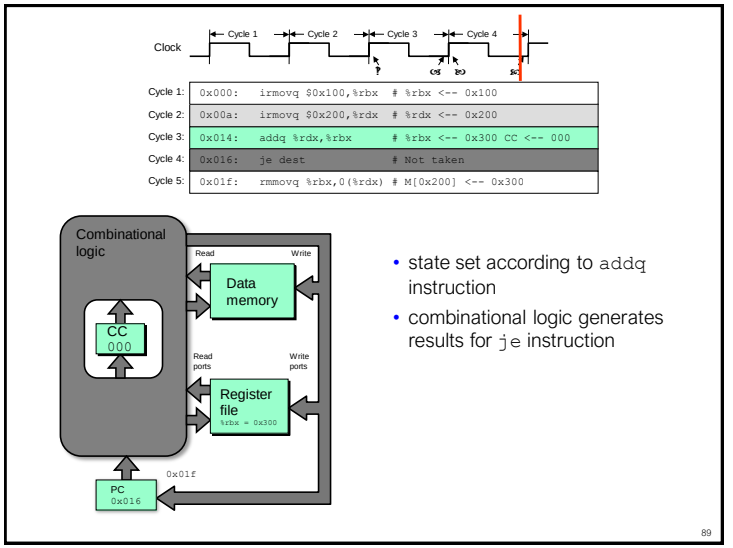




- state set according to `addq` instruction
- combinational logic starting to react to state changes

88

88



89

Another Way to Look At the Microarchitecture

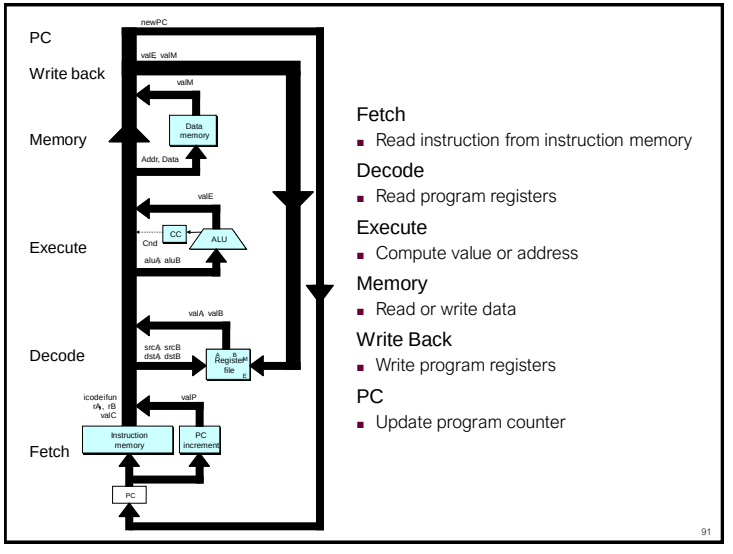
Principles:

- Execute each instruction one at a time, one after another
- Express every instruction as series of simple steps
- Dedicated hardware structure for completing each step
- Follow same general flow for each instruction type

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

90

90



91

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

92

92

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

93

93

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 based on setting of condition codes and branch condition

94

94

Processor Microarchitecture

- Sequential, single-cycle microarchitecture implementation
  - Basic idea
  - Hardware implementation
- Pipelined microarchitecture implementation
  - Basic Principles
  - Difficulties: Control Dependency
  - Difficulties: Data Dependency

95

95

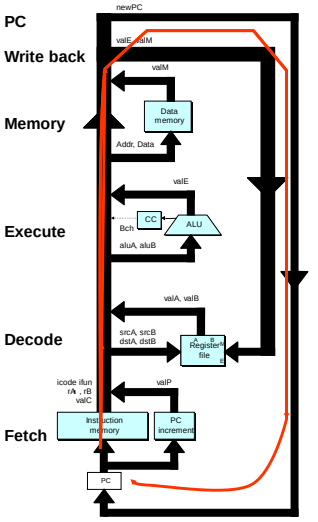
Possible Implementation Strategies

- Single-cycle control
- Multi-cycle control
- Pipelined (in-order execution)
- Superscalar and out-of-order execution

96

SEQ Hardware Structure

- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memory
    - 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



Single-Cycle Control

- Cannot use any hardware unit twice in one cycle
  - Multiple datapath elements (e.g., ALU and PC incrementer) if needed for multiple purposes
  - Multiple reads or writes to memory or register file require multiporting
- Clock cycle must accommodate instruction with the longest latency

Multi-cycle Control

- Split single instruction into multiple pieces
- Execute individual pieces using hardwired finite state machine (FSM) or microprogramming
- Can finish simple instructions in fewer cycles
- Can run clock faster
- Still execute a single instruction at a time

Pipelined Datapath

- Multiple instructions overlapped in execution
- Improve throughput, not individual instruction execution time
- Exploit parallelism among instructions in a sequential stream
- Balance length of each stage. Ideally -
  - Time between instrs $_{pipelined}$ = Time between instrs $_{non\text{-}pipelined}$ /Number of pipe stages

Real-World Pipelines: Car Washes

Sequential

Pipelined

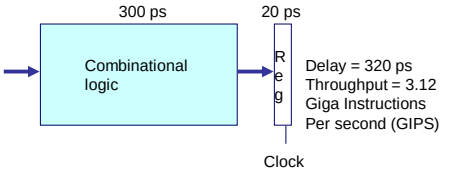
Idea

- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed

101

101

Computational Example



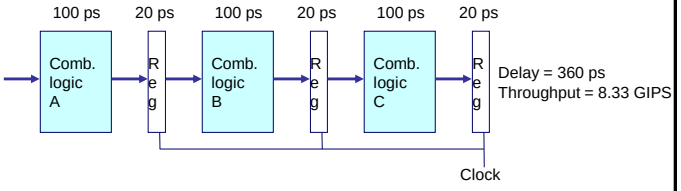
System

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle time of at least 320 ps

102

102

3-Stage Pipelined Version



System

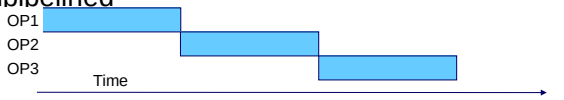
- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
  - Begin new operation every 120 ps
- Overall latency increases
  - 360 ps from start to finish

103

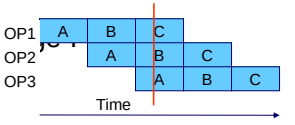
103

Pipeline Diagrams

Unpipelined



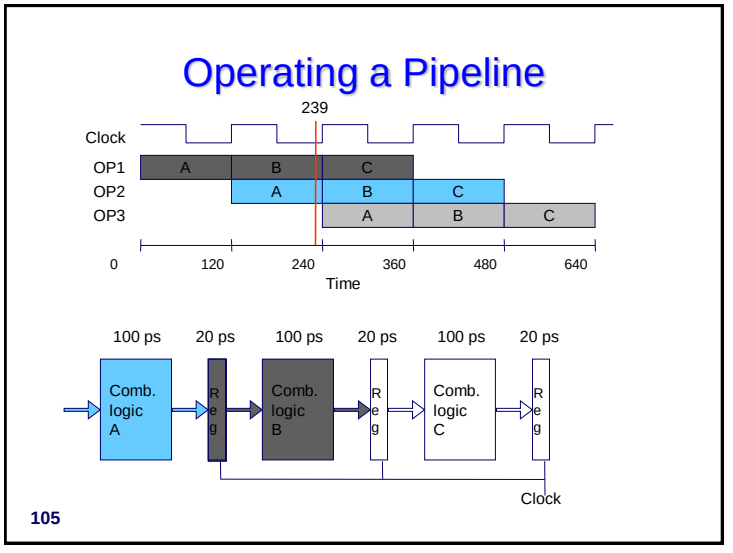
- Cannot start new operation until previous one completes



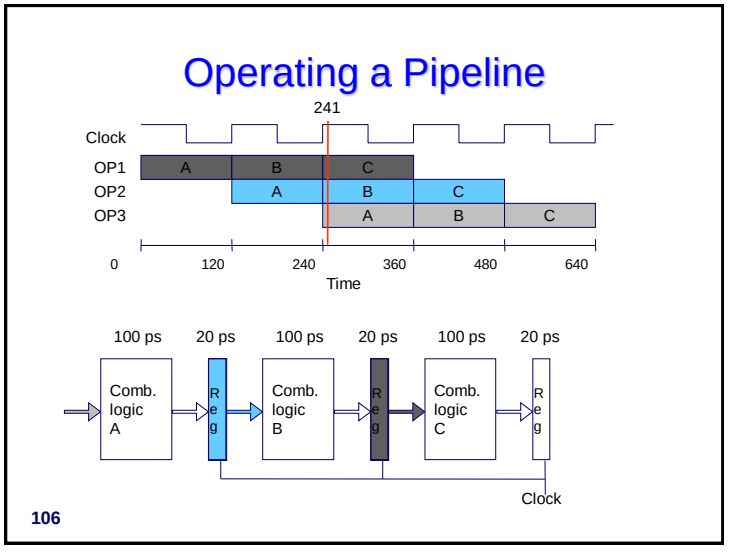
- Up to 3 operations in process simultaneously

104

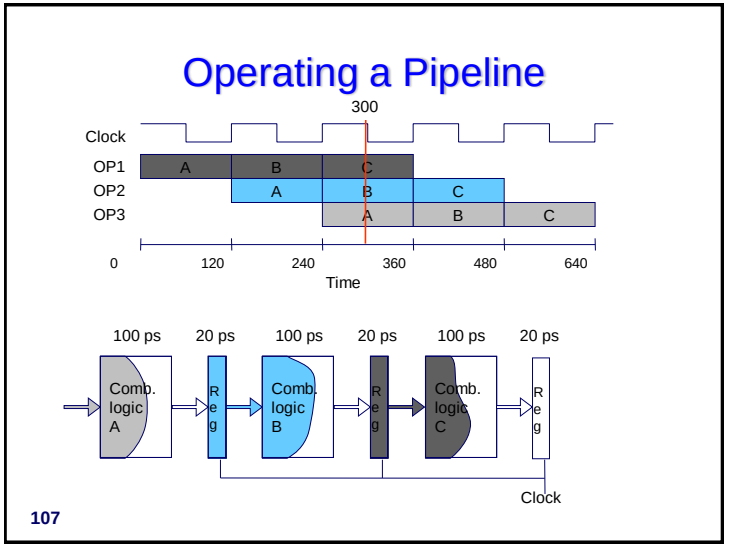
104



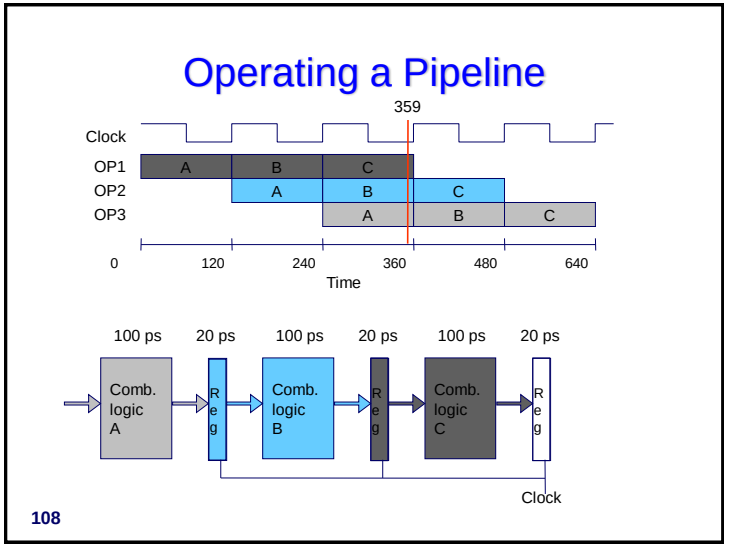
105



106



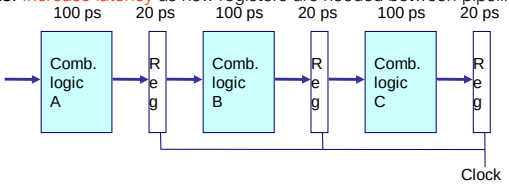
107

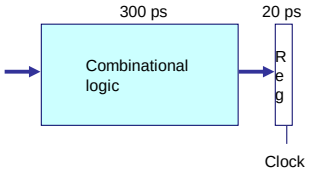


108

Pipeline Trade-offs

- Pros:Increase throughput. Can process more instructions in a given time span.
- Cons: Increase latency as new registers are needed between pipeline stages.



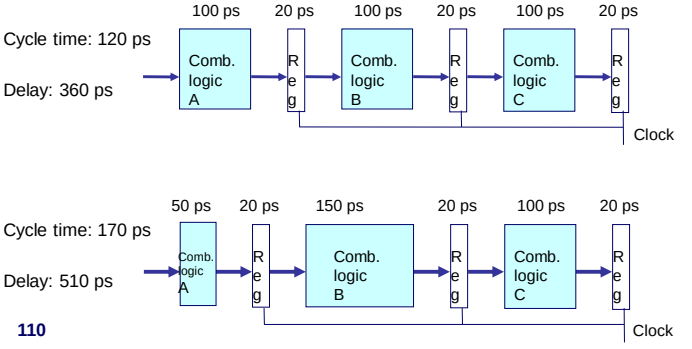


109

109

Unbalanced Pipeline

- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

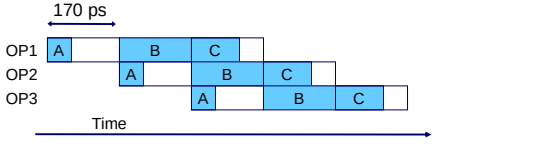


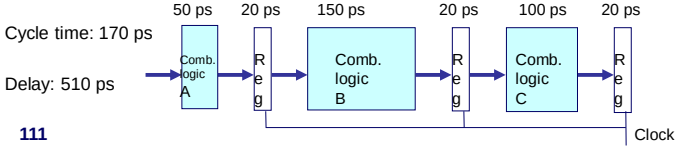
110

110

Unbalanced Pipeline

- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput



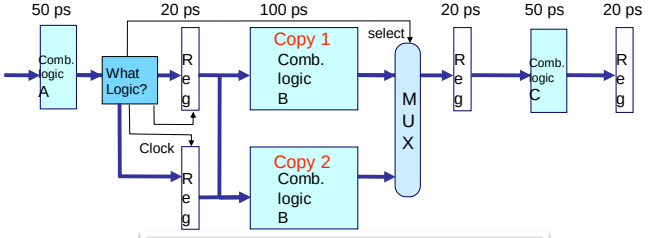


111

111

Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can't further pipeline a stage?
- Solution 2: Use multiple copies of the slow component



What logic do you need there?

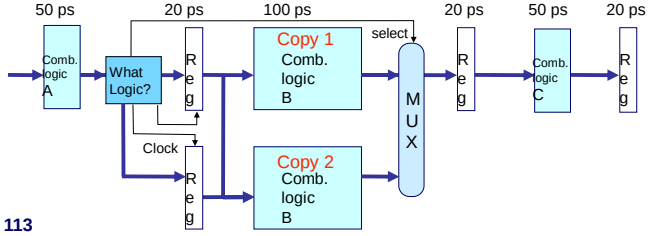
Hint: it needs to control the clock signals of the two registers and the select signal of the MUX.

112

112

Mitigating Unbalanced Pipeline

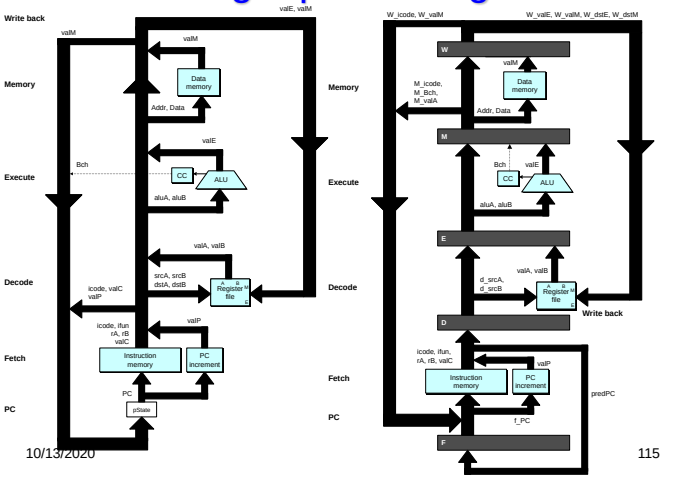
- Data sent to copy 1 in odd cycles and to copy 2 in even cycles.
- This is called 2-way interleaving. Effectively the same as pipelining Comb. logic B into two sub-stages.
- The cycle time is reduced to 70 ps (as opposed to 120 ps) at the cost of extra hardware.



113

113

Adding Pipeline Registers



10/13/2020

115

115