Simple Processor Design

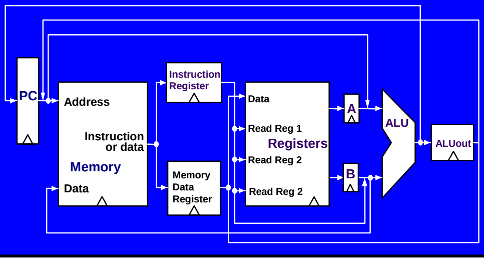
Multiple Cycle Implementation

Chapter 5.5
EEC170 FQ 2005

Courtesy of Prof. Kent Wilken

Multicycle Implementation

- One clock cycle for each step
  - shorter clock cycle
- Single memory unit, single ALU shared across cycles

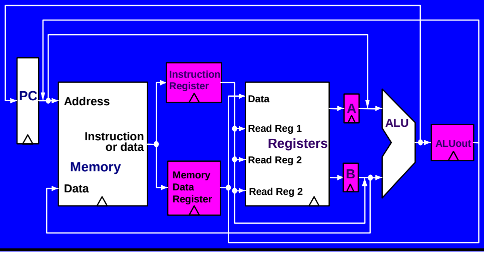


Multicycle Implementation

- Instructions that use more functional units (e.g., Load) take more cycles
- Instructions that use fewer units (e.g., Jump) take fewer cycles
- Maybe CPI x CCT will be lower

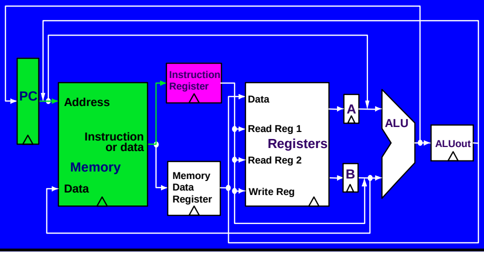
Multicycle Implementation

- New latches hold intermediate results between clock cycles
  - IR and MDR get output of memory
  - A and B get output of Register File
  - ALUout gets output of ALU



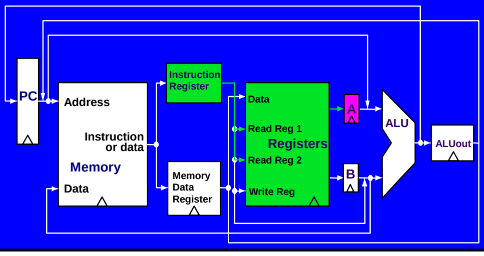
Multicycle Implementation: LW

- Instruction fetch



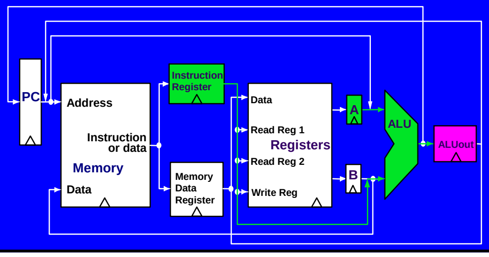
Multicycle Implementation: LW

- Register read



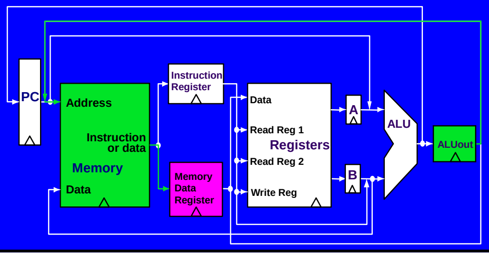
Multicycle Implementation: LW

- Address computation



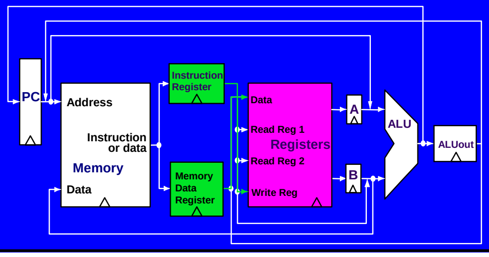
Multicycle Implementation: LW

- Memory read



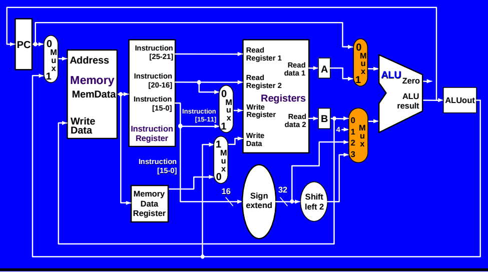
Multicycle Implementation: LW

- Register write



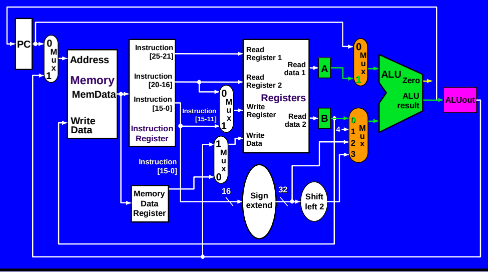
Multi-Use ALU

- More ALU input selection because one ALU shared for:
  - R-type ALU ops, Branch condition
  - Address computation, I-type ALU ops
  - PC+4
  - Branch target



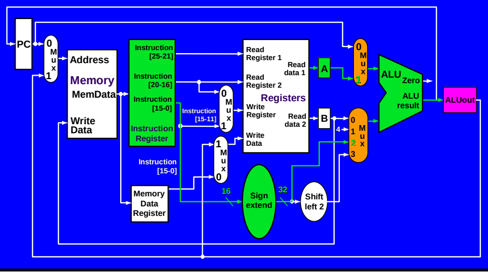
Multi-Use ALU

- R-type ALU ops, Branch condition



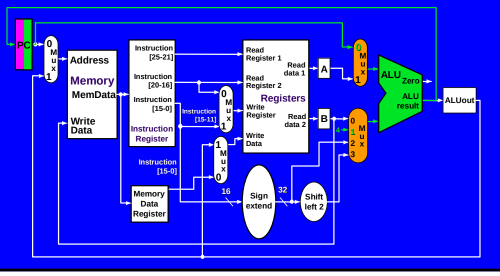
Multi-Use ALU

- Address computation, I-type ALU ops



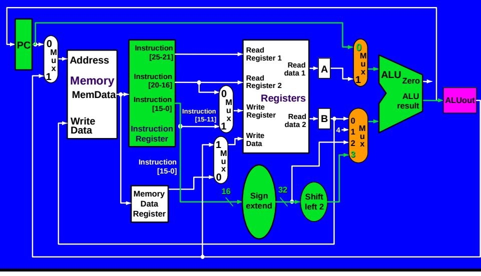
Multi-Use ALU

- PC+4



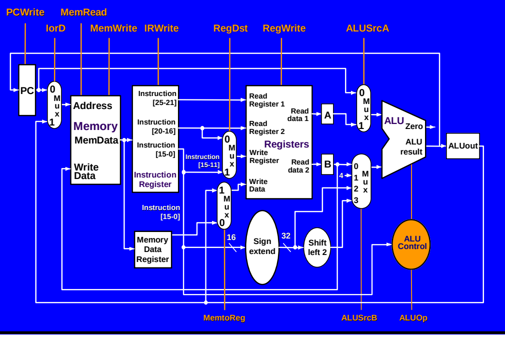
Multi-Use ALU

- Branch target



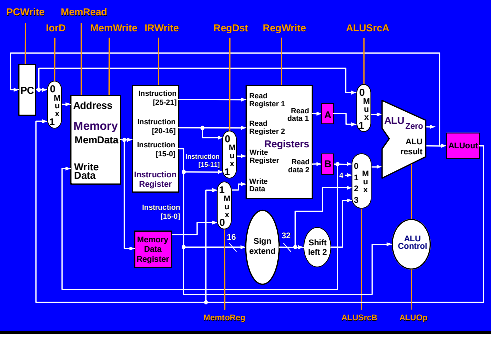
Multicycle Control Lines

- Control more complex than Single Cycle: must activate for given instruction during specific clock cycle

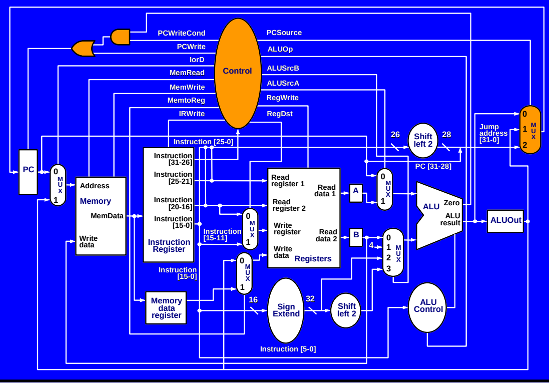


Multicycle Control Lines

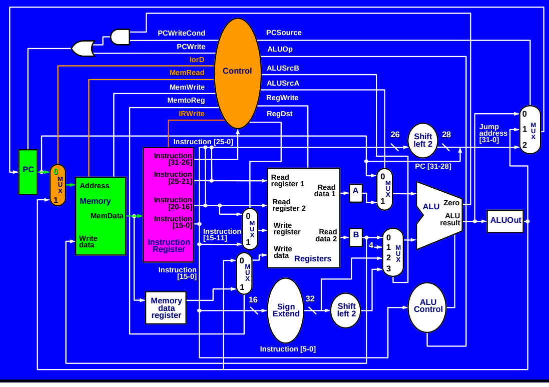
- Latches A, B, ALUout and Memory Data register are latched every cycle, don't need control lines



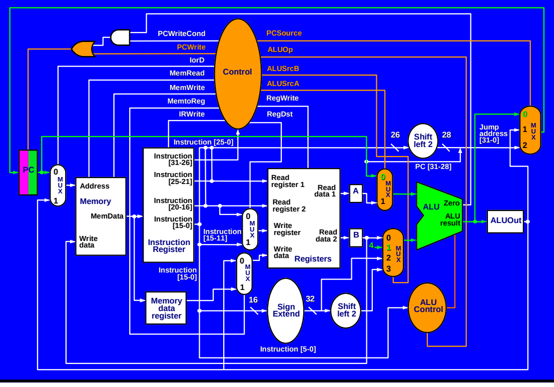
Complete Multi-Cycle Design



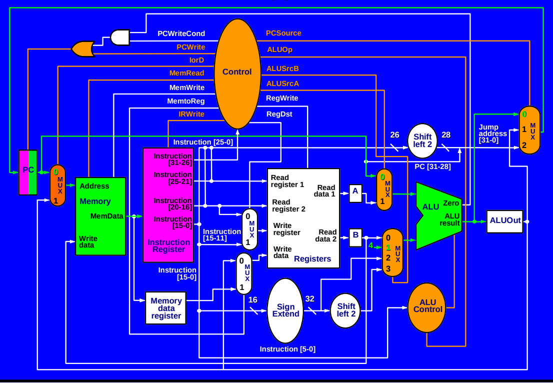
Instruction Fetch



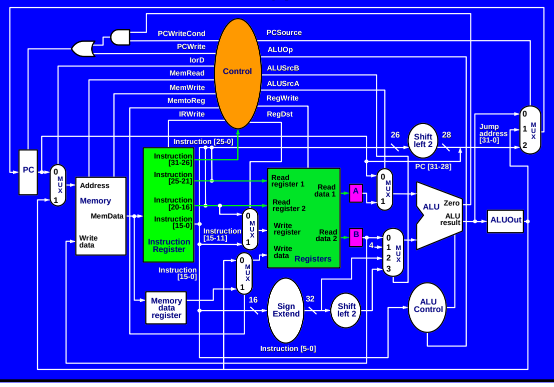
Compute PC + 4



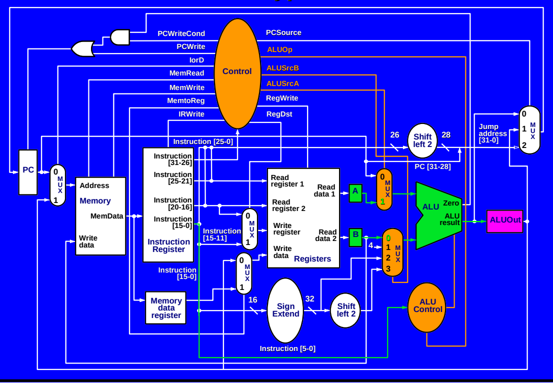
Complete Instruction Fetch Cycle
R-type



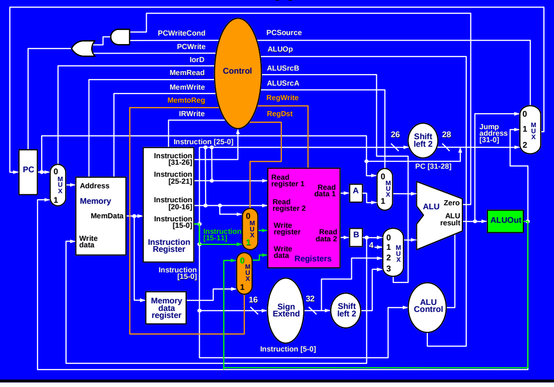
Register Read Cycle
R-type



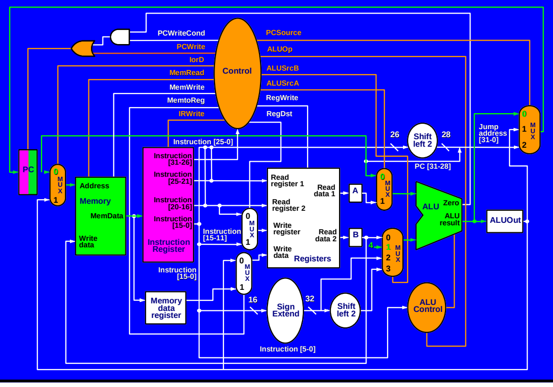
ALU Cycle
R-type



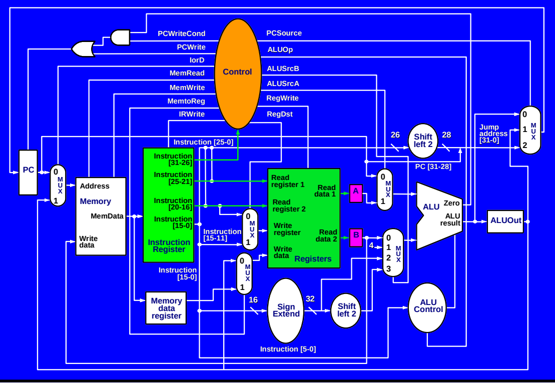
Register Write Cycle
R-type



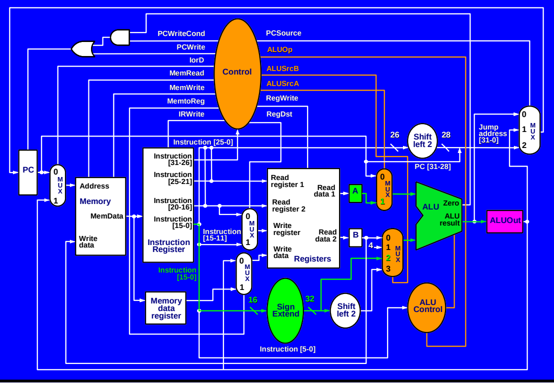
Instruction Fetch Cycle
Load Word



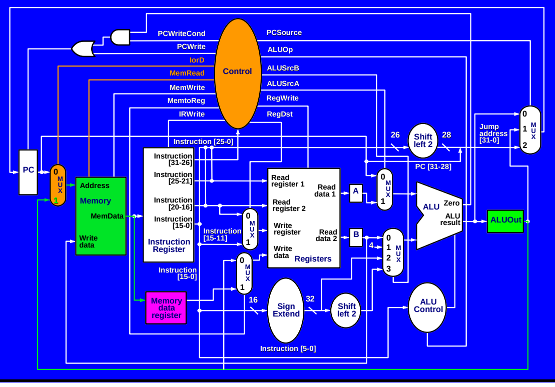
Register Read Cycle
Load Word



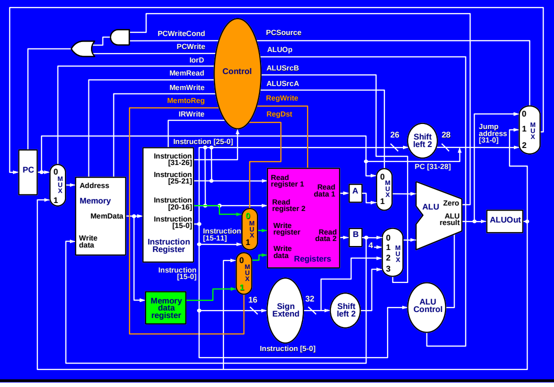
Address Computation Cycle
Load Word



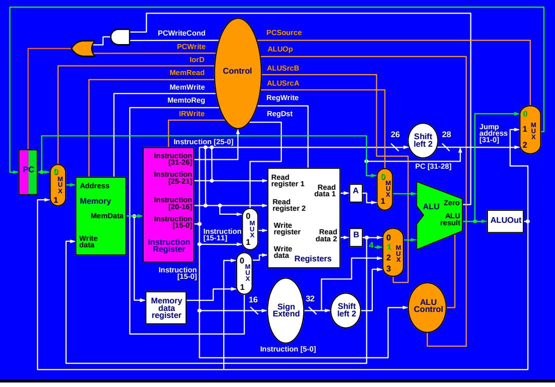
Memory Access Cycle
Load Word



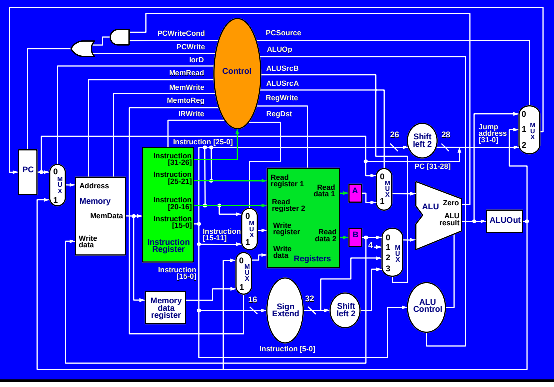
Register Write Cycle
Load Word



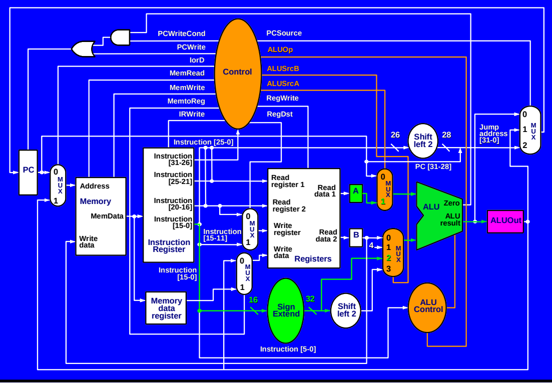
Instruction Fetch Cycle
Store Word



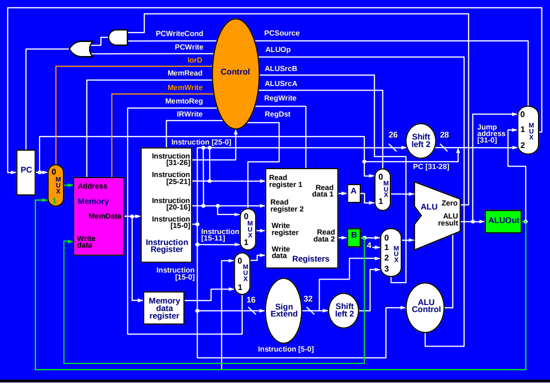
Register Read Cycle
Store Word



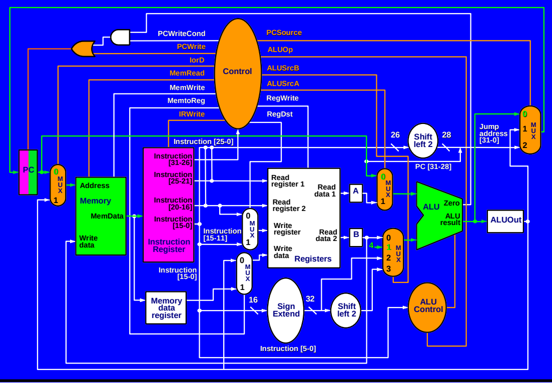
Address Computation Cycle
Store Word



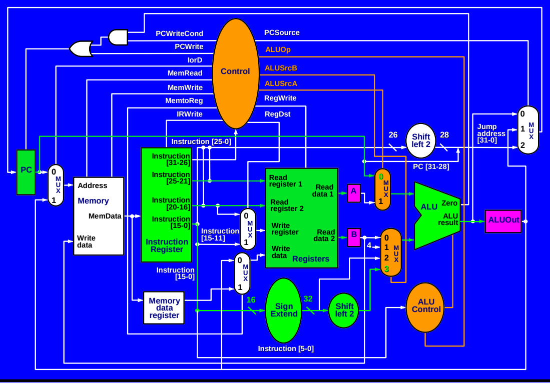
Memory Access Cycle
Store Word



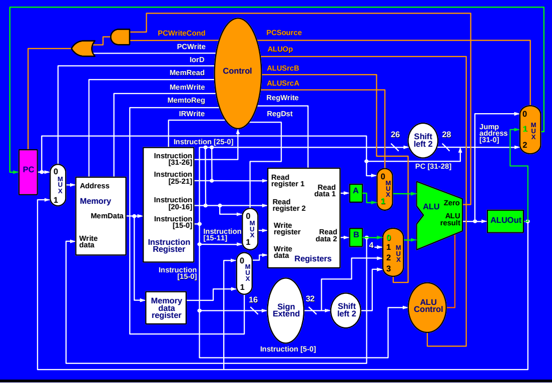
Instruction Fetch Cycle
Branch



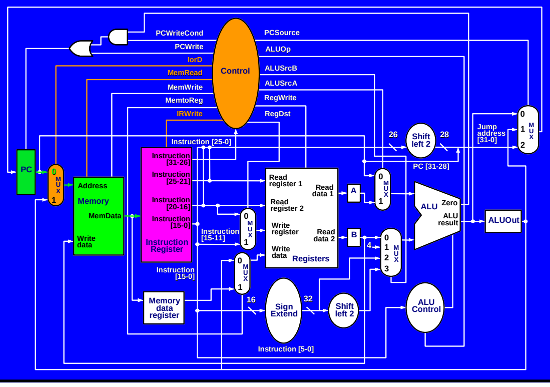
Register Read/Target Address Cycle
Branch



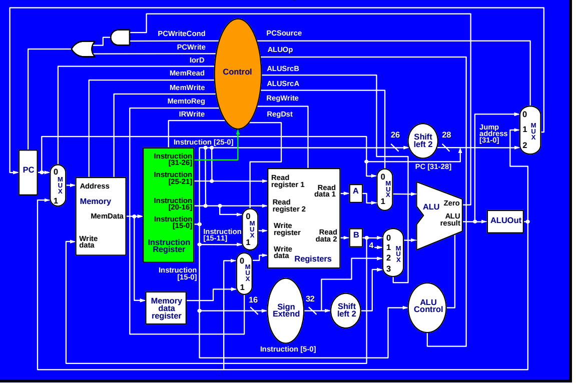
Branch Condition/PC Write Cycle
Branch



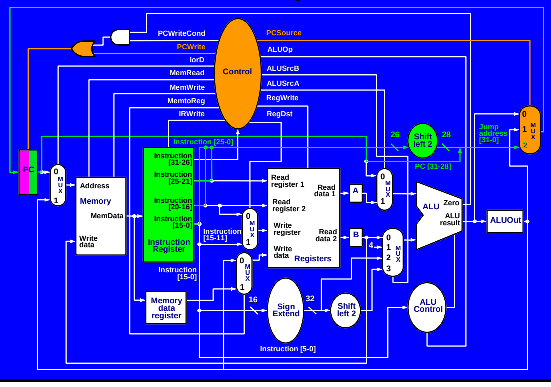
Instruction Fetch Cycle
Jump



Decode Cycle
Jump



PC Write Cycle
Jump



Multicycle Clock Cycle Time

- CCT determined by slowest functional unit:
  - Register file: 50ps
  - ALU and adders: 100ps
  - Memory: 200ps

Multicycle CPI

- Cycles for each instruction class is:
  - Load: 5
  - Store: 4
  - ALU Op: 4
  - Branch: 3
  - Jump: 3
- SPECint2000 instruction mix
  - Load: 25%
  - Store: 10%
  - ALU Op: 52%
  - Branch: 11%
  - Jump: 2%

Multicycle CPI Computation

CPI total = Σ CPI$_i$ x f_i =

5x0.25 + 4x0.10 + 4x0.52 + 3x0.11 + 3x0.02 =

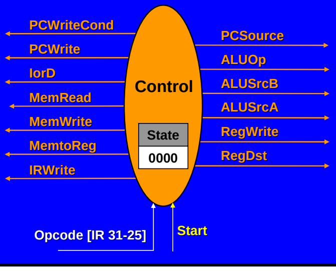
1.25 + 0.4 + 2.08 + 0.33 + 0.06 = 4.12

Multicycle Performance

- Good news:
  - CCT = 200ps, 3x lower than single cycle design
  - Only one adder, one memory unit
- Bad
  - CPI is higher by 4x than single cycle design
  - Control unit is much more complex (see next lecture)
- Average instruction execution time (IET) = CPI x CCT = 4.12 x 200ps = 824ps
  - Worse performance than single cycle at 600ps!

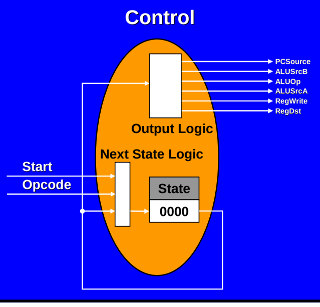
Multicycle Control Unit

- Control line values are based on the instruction opcode and the cycle within that instruction
- Can be implemented as a Moore state machine where outputs are determined by internal state register



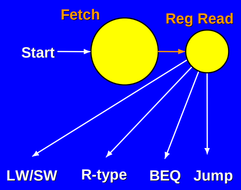
Machine Next State

- Outputs depend on current state
- Next state depends on current state and inputs

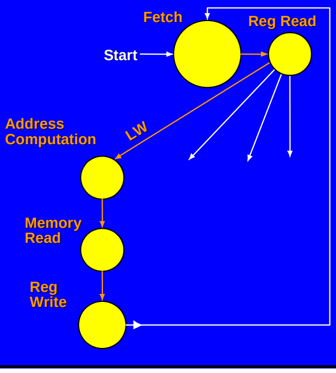


Machine State

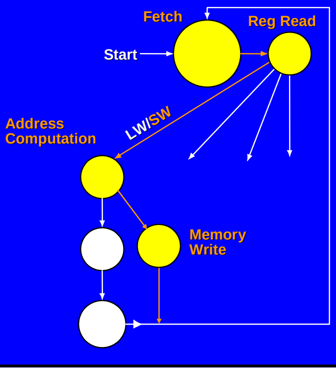
- One state for each instruction cycle
- First two cycles is same for all instructions, states are shared



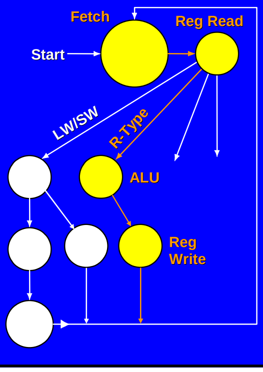
Load States



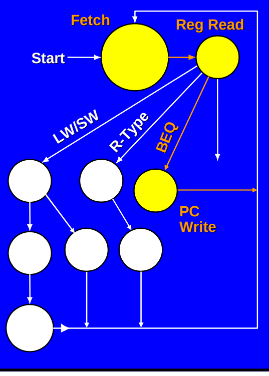
Store States



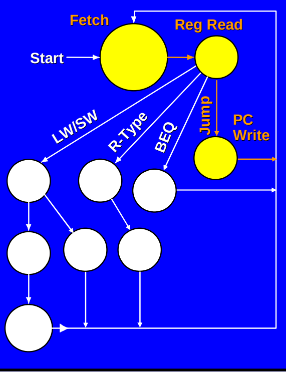
R-Type States



BEQ States

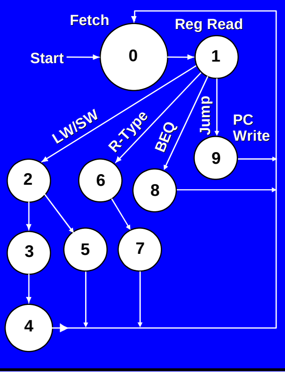


Jump States



State Assignment

- A number is assigned to each state
- Many assignments are possible, 16!/6! for this machine
- Assignment usually made to minimize hardware cost
- Here assignment made to improve clarity



Next State Table

- Next state function of opcode and current state

Current State	Opcode	Next State
0	xxxxxx	1
1	LW	2
1	SW	2
1	R-Type	6
1	BEQ	8
1	Jump	9
2	LW	3
2	SW	5
3	xxxxxx	4
4	xxxxxx	0
5	xxxxxx	0
6	xxxxxx	7
7	xxxxxx	0
8	xxxxxx	0
9	xxxxxx	0

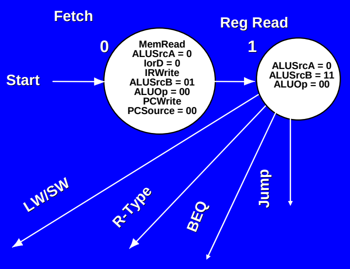
Next State Table in Binary

- Really four tables for NS_0, NS_1, NS_2 and NS_3

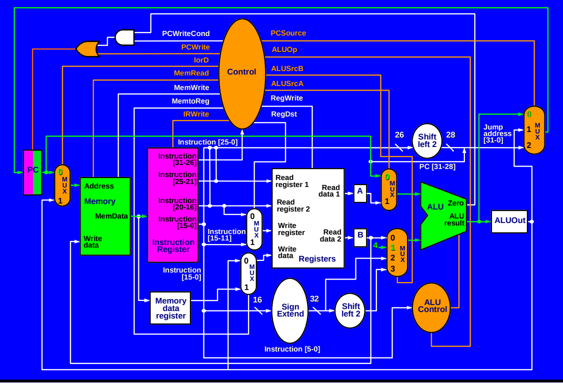
Current State	Opcode	Next State
0000	xxxxxx	0001
0001	100011	0010
0001	101011	0010
0001	000000	0110
0001	000100	1000
0001	000010	1001
0010	100011	0011
0010	101011	0101
0011	xxxxxx	0100
0100	xxxxxx	0000
0101	xxxxxx	0000
0110	xxxxxx	0111
0111	xxxxxx	0000
1000	xxxxxx	0000
1001	xxxxxx	0000

Control Lines

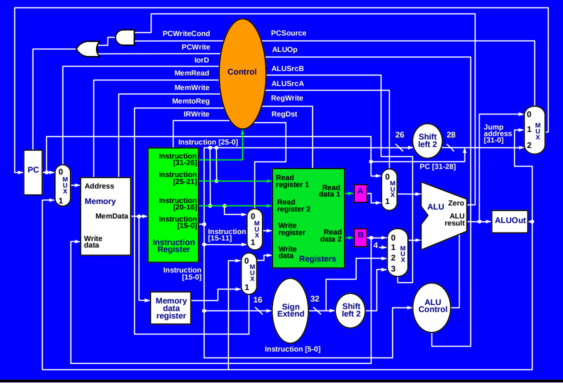
- Control lines are dependent only on the state



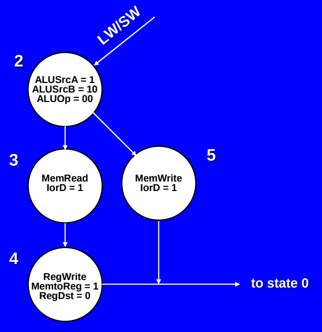
Instruction Fetch Cycle
Store Word



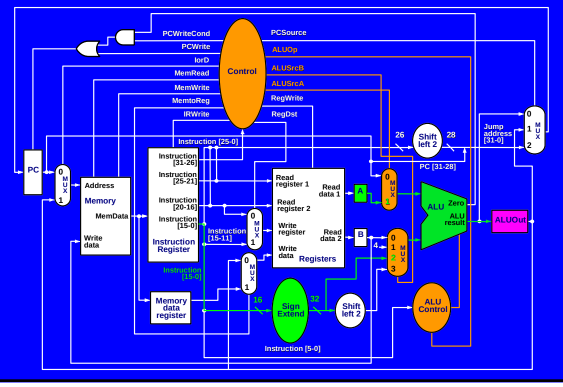
Register Read Cycle
Store Word



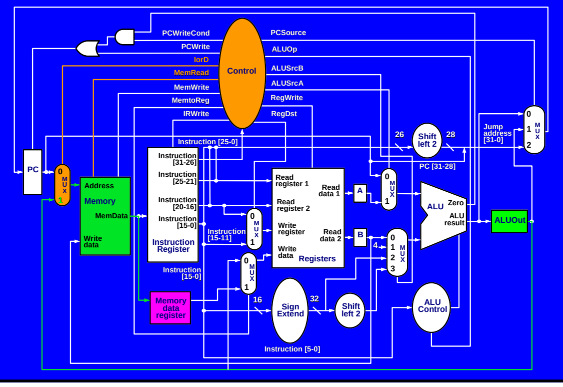
Control Lines for LW & SW



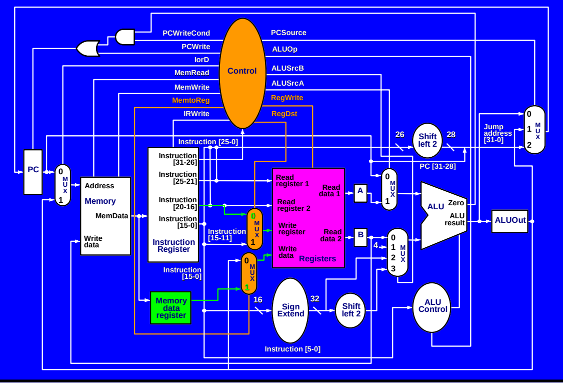
Address Computation Cycle
Load Word



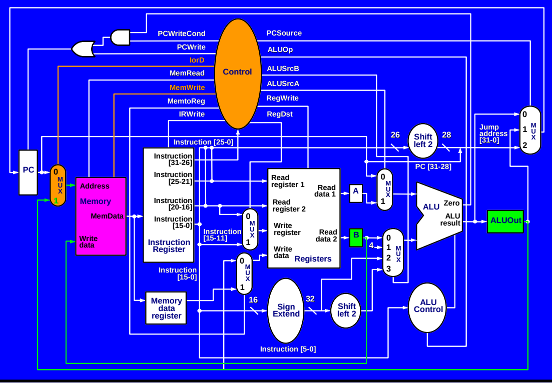
Memory Access Cycle
Load Word



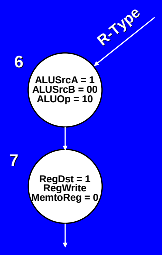
Register Write Cycle
Load Word



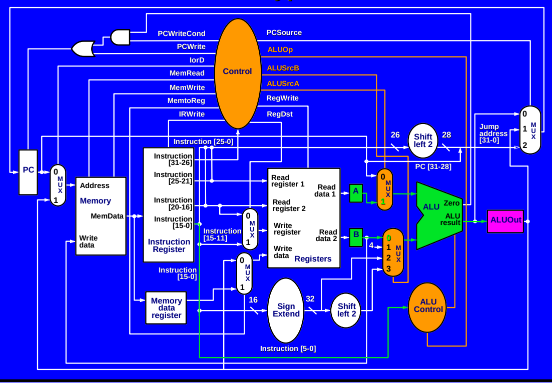
Memory Access Cycle
Store Word



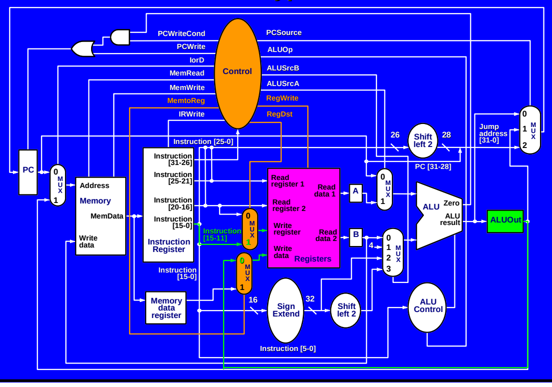
Control Lines for R-Type



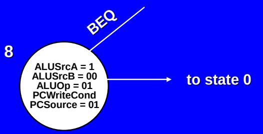
ALU Cycle
R-type



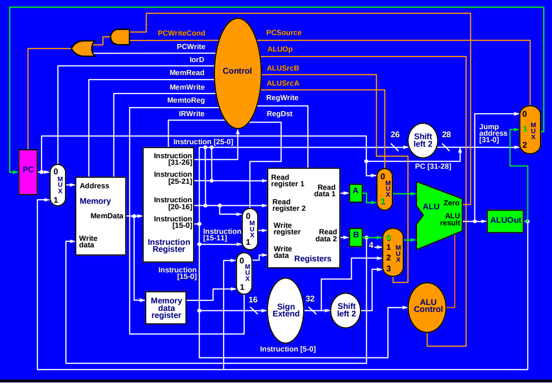
Register Write Cycle
R-type



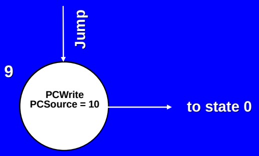
Control Lines for Branch



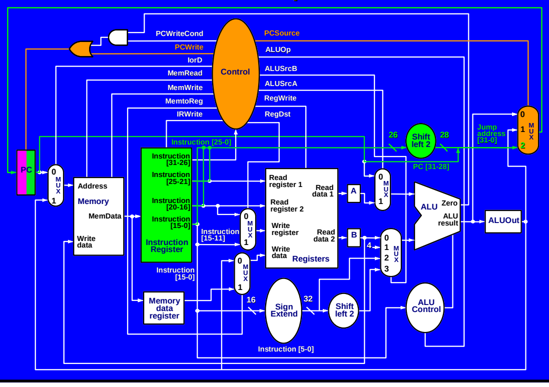
Branch Condition/PC Write Cycle
Branch



Control Lines for Jump



PC Write Cycle
Jump



Truth Table for Control Lines

Outputs	Input Values S[3-0]									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

Exceptions

- An exception is an event that occurs within the processor which requires intervention from the OS
- An exception causes execution to change from current program to OS exception handler
- Example of exceptions include:
  - Address bounds violation
  - Arithmetic Overflow
  - Divide by Zero
  - Illegal opcode
  - Call to OS from user program

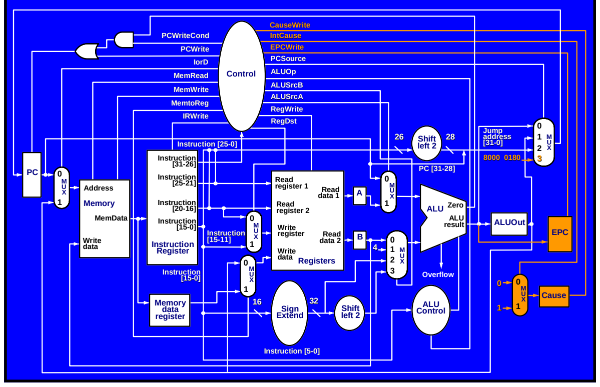
Exception Handling

- An exception is much like a procedure call
  - Requires address of where exception occurred
    - So OS can return to program after exception handling, if appropriate
    - So exception handler can identify instruction causing exception for proper exception processing
  - Requires a parameter indicating what caused the exception
  - Address and parameter cannot be written to normal registers, otherwise current program state would be destroyed
    - Special exception registers are required

Datapath Support for Exceptions

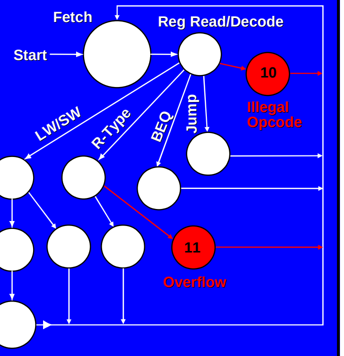
- All exceptions jump to a fixed, hard-wired address within OS: exception handler entry point
- Address of where exception occurred is stored in exception program counter (EPC)
  - Read by exception handler using special instruction
- Exception parameter is stored in cause register

Exception Support in Multicycle Design

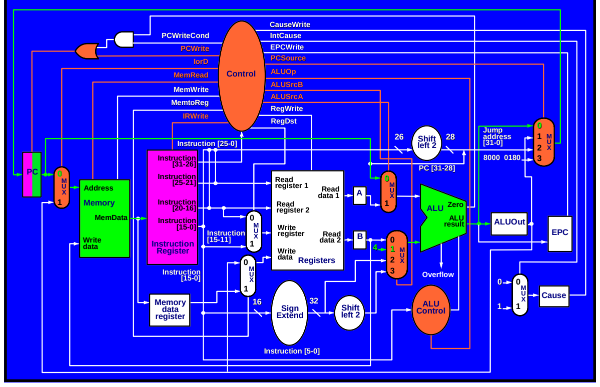


Control Unit Support for Exceptions

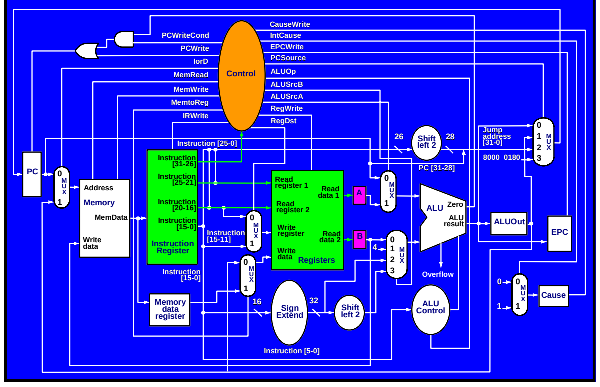
- Exception control cycle occurs just after cycle where exception happens



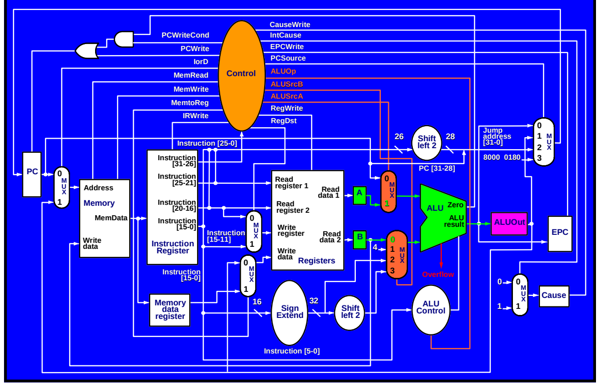
Instruction Fetch Cycle
R-type



Register Read Cycle
R-type

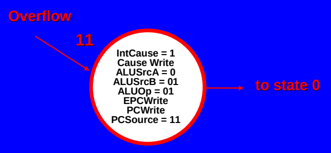


ALU Cycle
R-type Overflow

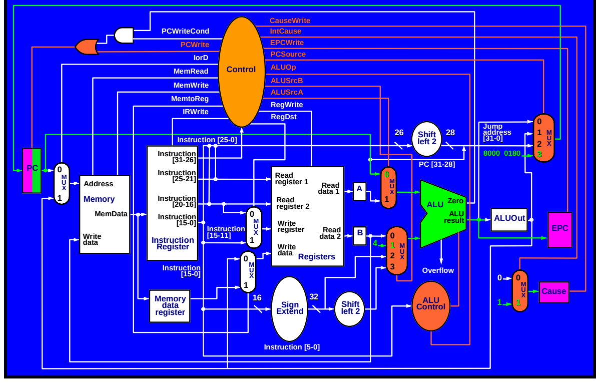


Control Lines for R-Type Overflow

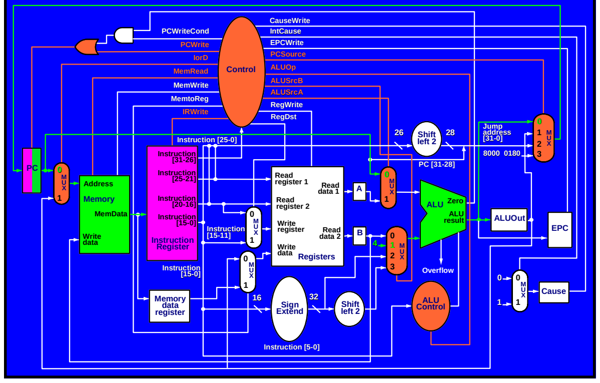
- Set Cause = 1
- Use ALU to compute (PC+4) – 4 = PC
- Set EPC = PC



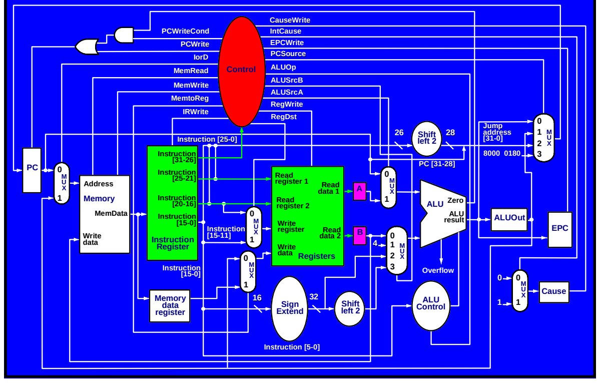
Exception Cycle
R-type Overflow



Instruction Fetch Cycle
Illegal Opcode

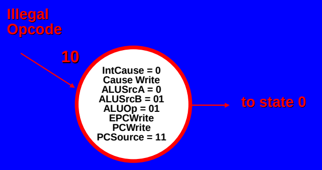


Register Read/Decode Cycle
Illegal Opcode

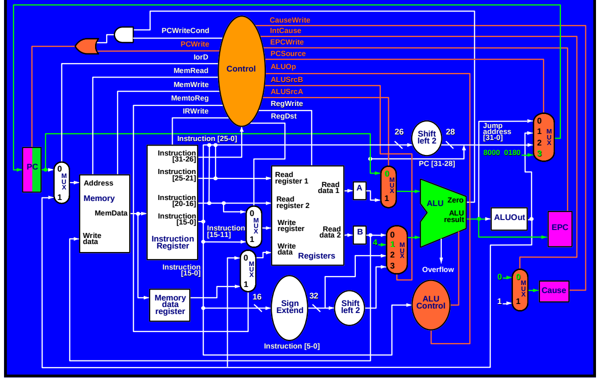


Control Lines for Illegal Opcode

- Set Cause = 0
- Use ALU to compute (PC+4) – 4 = PC
- Set EPC = PC



Exception Cycle
Illegal Opcode


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Control
CauseWrite
IntCause
EPCWrite
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
Address
Memory
MemData
Write data
Instruction [31-26]
Instruction [25-21]
Instruction [20-16]
Instruction [15-0]
Instruction Register
Instruction [15-0]
Memory data register
Instruction [25-0]
Instruction [15-11]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
A
B
4
Sign Extend
16
32
Shift left 2
26
28
PC [31-28]
Jump address [31-0]
8000 0180
ALU
Zero
ALU result
ALUOut
Overflow
ALU Control
Instruction [5-0]
EPC
Cause
0
1
2
3
MUX