Book's Definition of Performance

- For some program running on machine X,

  $$\text{Performance}_X = 1 / \text{Execution time}_X$$

- "X is n times faster than Y"

  $$\text{Performance}_X \;/\; \text{Performance}_Y = n$$

- Problem:
  - machine A runs a program in 20 seconds
  - machine B runs the same program in 25 seconds

Example

- **Our favorite program runs in 10 seconds on computer A, which has a 400 Mhz. clock. We are trying to help a computer designer build a new machine B, that will run this program in 6 seconds. The designer can use new (or perhaps more expensive) technology to substantially increase the clock rate, but has informed us that this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for the same program.   What clock rate should we tell the designer to target?"**

- **Don't Panic, can easily work this out from basic principles**

Now that we understand cycles

- **A given program will require**
  - **some number of instructions (machine instructions)**
  - **some number of cycles**
  - **some number of seconds**
- **We have a vocabulary that relates these quantities:**
  - **cycle time (seconds per cycle)**
  - **clock rate (cycles per second)**
  - **CPI (cycles per instruction)**

    a floating point intensive application might have a higher CPI
  - **MIPS (millions of instructions per second)**

    this would be higher for a program using simple instructions

Performance

- Performance is determined by execution time
- Do any of the other variables equal performance?
  - # of cycles to execute program?
  - # of instructions in program?
  - # of cycles per second?
  - average # of cycles per instruction?
  - average # of instructions per second?

- Common pitfall: thinking one of the variables is indicative of performance when it really isn't.

CPI Example

- **Suppose we have two implementations of the same instruction set architecture (ISA).**

  For some program,

  Machine A has a clock cycle time of 10 ns. and a CPI of 2.0
  Machine B has a clock cycle time of 20 ns. and a CPI of 1.2

  What machine is faster for this program, and by how much?

- ***If two machines have the same ISA which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?***

# of Instructions Example

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles (respectively).

  The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C
  The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.

  Which sequence will be faster? How much?
  What is the CPI for each sequence?

MIPS example

- **Two different compilers are being tested for a 100 MHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require one, two, and three cycles (respectively). Both compilers are used to produce code for a large piece of software.**

  The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

  The second compiler's code uses 10 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

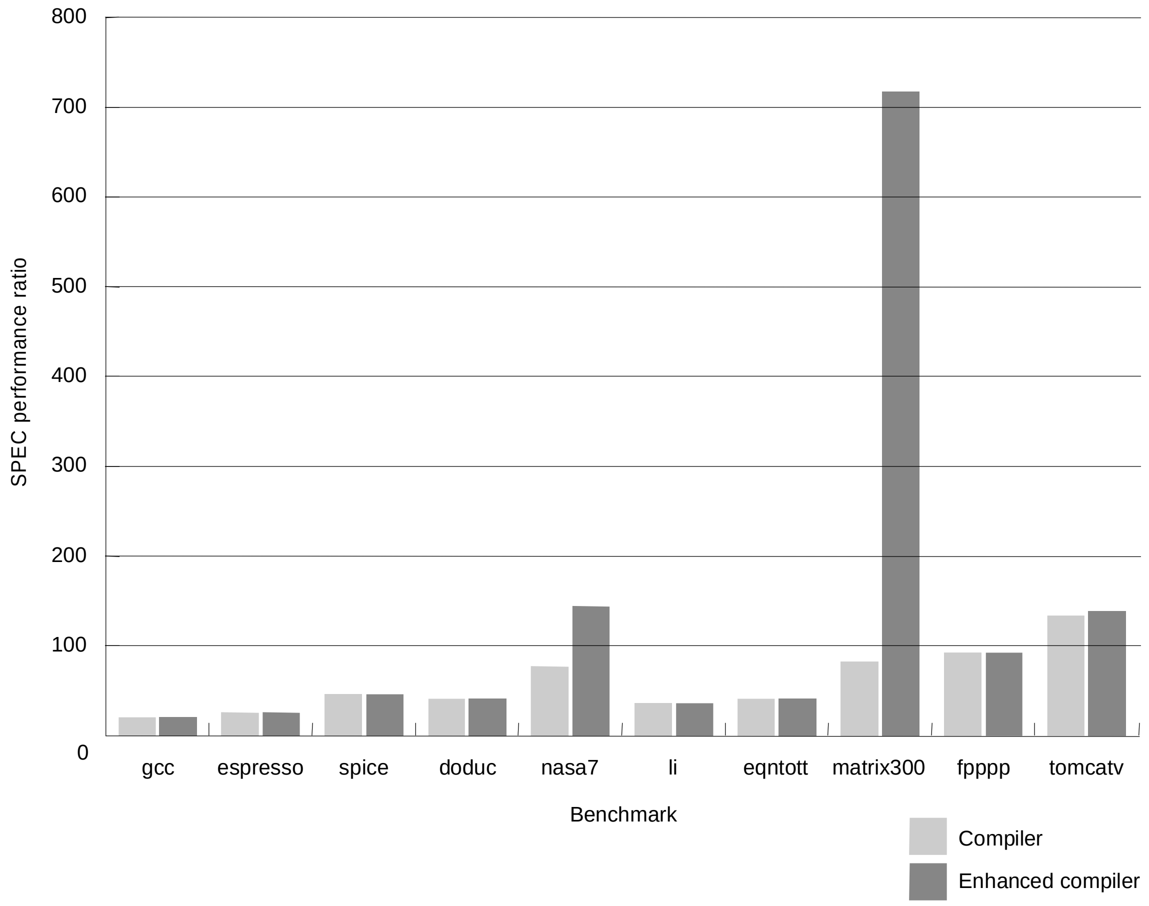
- **Which sequence will be faster according to MIPS?**
- **Which sequence will be faster according to execution time?**

Benchmarks

- Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications
    e.g., compilers/editors, scientific applications, graphics, etc.
- Small benchmarks
  - nice for architects and designers
  - easy to standardize
  - can be abused
- SPEC (System Performance Evaluation Cooperative)
  - companies have agreed on a set of real program and inputs
  - can still be abused (Intel's "other" bug)
  - valuable indicator of performance (and compiler technology)

SPEC '89

- **Compiler "enhancements" and performance**



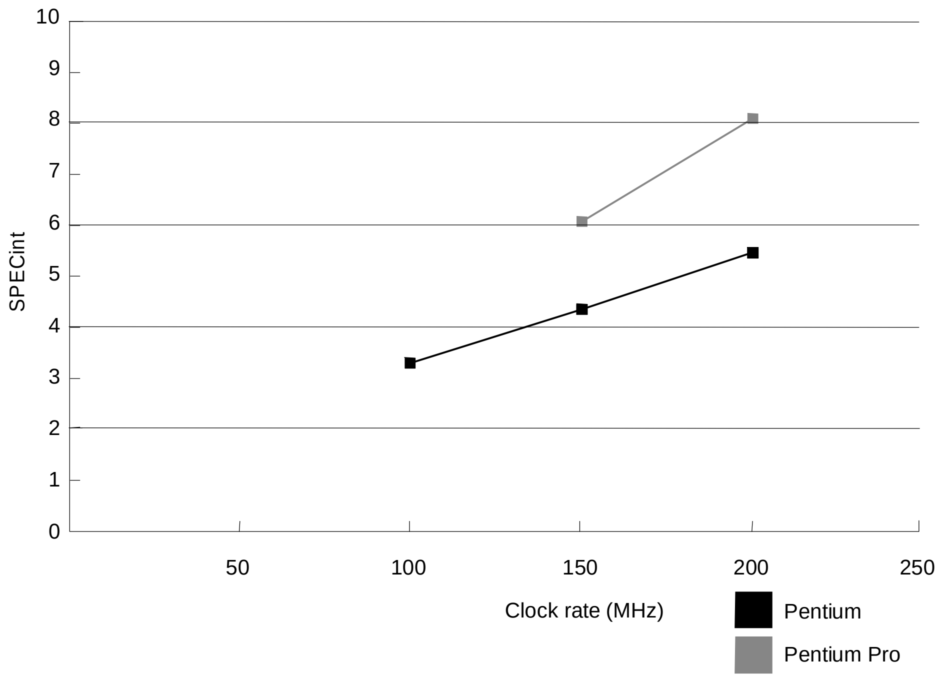
SPEC ‘95

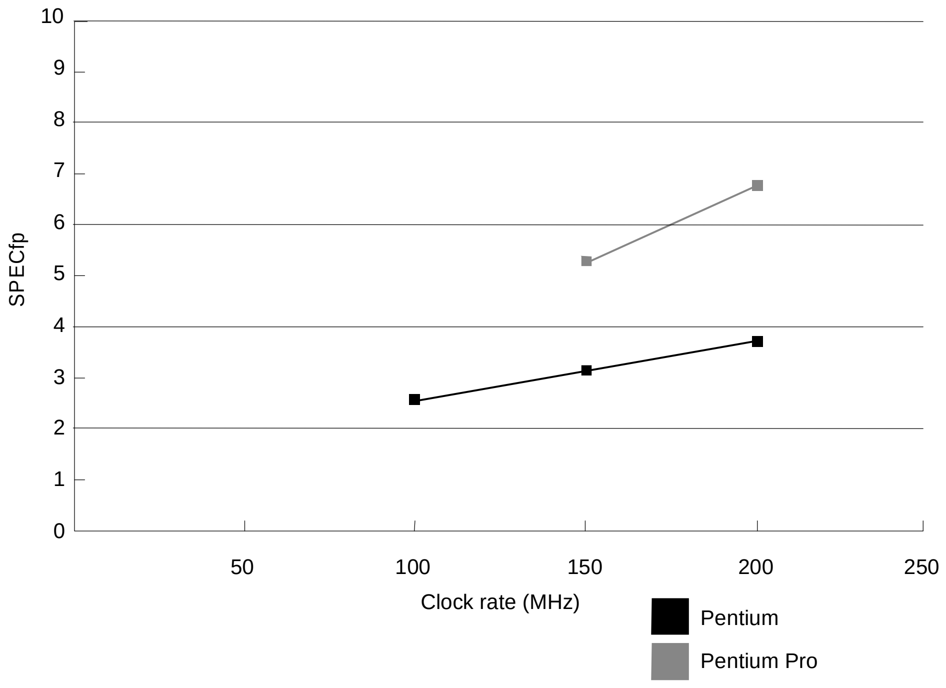
Benchmark	Description
go	Artificial intelligence; plays the game of Go
m88ksim	Motorola 88k chip simulator; runs test program
gcc	The Gnu C compiler generating SPARC code
compress	Compresses and decompresses file in memory
li	Lisp interpreter
ijpeg	Graphic compression and decompression
perl	Manipulates strings and prime numbers in the special-purpose programming language Perl
vortex	A database program
tomcatv	A mesh generation program
swim	Shallow water model with 513 x 513 grid
su2cor	quantum physics; Monte Carlo simulation
hydro2d	Astrophysics; Hydrodynamic Naiver Stokes equations
mgrid	Multigrid solver in 3-D potential field
applu	Parabolic/elliptic partial differential equations
trub3d	Simulates isotropic, homogeneous turbulence in a cube
apsi	Solves problems regarding temperature, wind velocity, and distribution of pollutant
fpppp	Quantum chemistry
wave5	Plasma physics; electromagnetic particle simulation

SPEC '95

Does doubling the clock rate double the performance?

Can a machine with a slower clock rate have better performance?





Amdahl's Law

Execution Time After Improvement =

Execution Time Unaffected +(Execution Time Affected / Amount of Improvement)

- **Example:**

  "Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

  How about making it 5 times faster?

- ***Principle: Make the common case fast***

Example

- **Suppose we enhance a machine making all floating-point instructions run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if half of the 10 seconds is spent executing floating-point instructions?**

- **We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?**

Remember

- Performance is specific to a particular program/s
  - Total execution time is a consistent summary of performance

- For a given architecture performance increases come from:
  - increases in clock rate (without adverse CPI affects)
  - improvements in processor organization that lower CPI
  - compiler enhancements that lower CPI and/or instruction count

- Pitfall: expecting improvement in one aspect of a machine’s

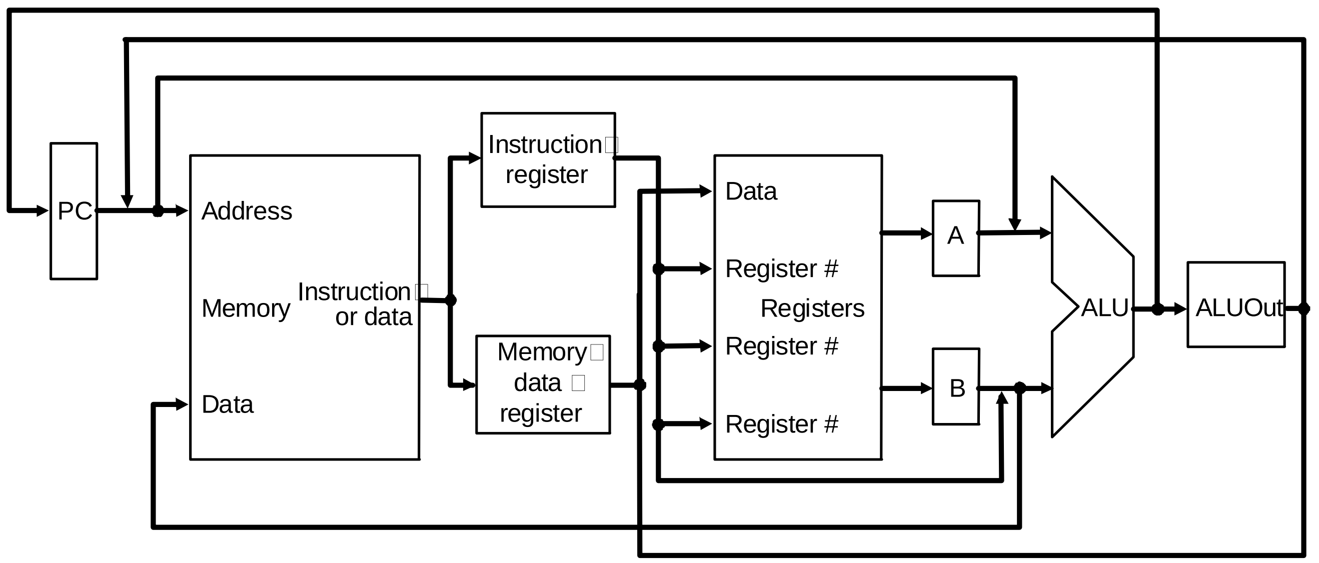
  performance to affect the total performance

- You should not always believe everything you read! Read carefully!

  (see newspaper articles, e.g., Exercise 2.37)

Where we are headed

- **Single Cycle Problems:**
  - **what if we had a more complicated instruction like floating point?**
  - **wasteful of area**
- **One Solution:**
  - **use a “smaller” cycle time and use different numbers of cycles for each instruction using a “multicycle” datapath:**



MIPS Instruction Format Again

31 26	25 21	20 16	15 11	10 6	5 0
LW	REG 1	REG 2	LOAD ADDRESS		OFFSET

31 26	25 21	20 16	15 11	10 6	5 0
SW	REG 1	REG 2	STORE ADDRESS		OFFSET

31 26	25 21	20 16	15 11	10 6	5 0
R-TYPE	REG 1	REG 2	DST	SHIFT AMOUNT	ADD/AND/OR/SLT

31 26	25 21	20 16	15 11	10 6	5 0
BEQ/BNE	REG 1	REG 2	BRANCH ADDRESS		OFFSET

31 26	25 21	20 16	15 11	10 6	5 0
JUMP	JUMP			ADDRESS	

Operation for Each Instruction

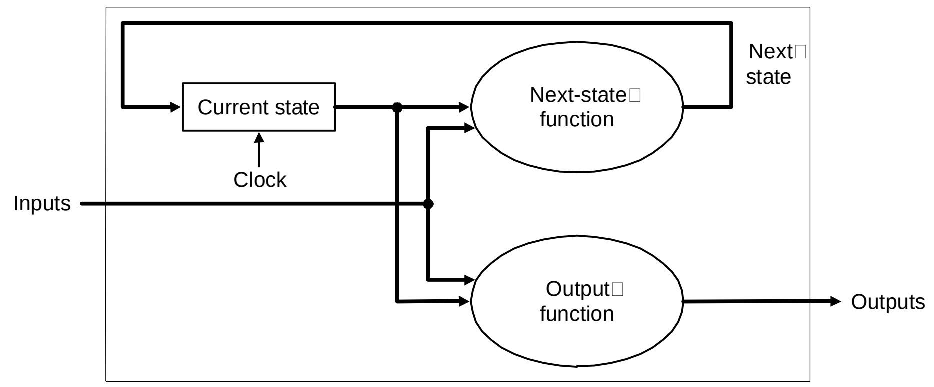
LW:	SW:	R-Type:	BR-Type:	JMP-Type:
1. READ INST	1. READ INST	1. READ INST	1. READ INST	1. READ INST
2. READ REG 1 *READ REG 2*	2. READ REG 1 READ REG 2	2. READ REG 1 READ REG 2	2. READ REG 1 READ REG 2	2.
3. ADD REG 1 + OFFSET	3. ADD REG 1 + OFFSET	3. OPERATE on REG 1 / REG 2	3. SUB REG 2 from REG 1	3.
4. READ MEM	4. WRITE MEM	4.	4.	4.
5. WRITE REG2	5.	5. WRITE DST	5.	5.

Multicycle Approach

- We will be reusing functional units
  - Break up the instruction execution in smaller steps
  - Each functional unit is used for a specific purpose in one cycle
  - Balance the work load
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- At the end of cycle, store results to be used again
  - Need additional registers
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control

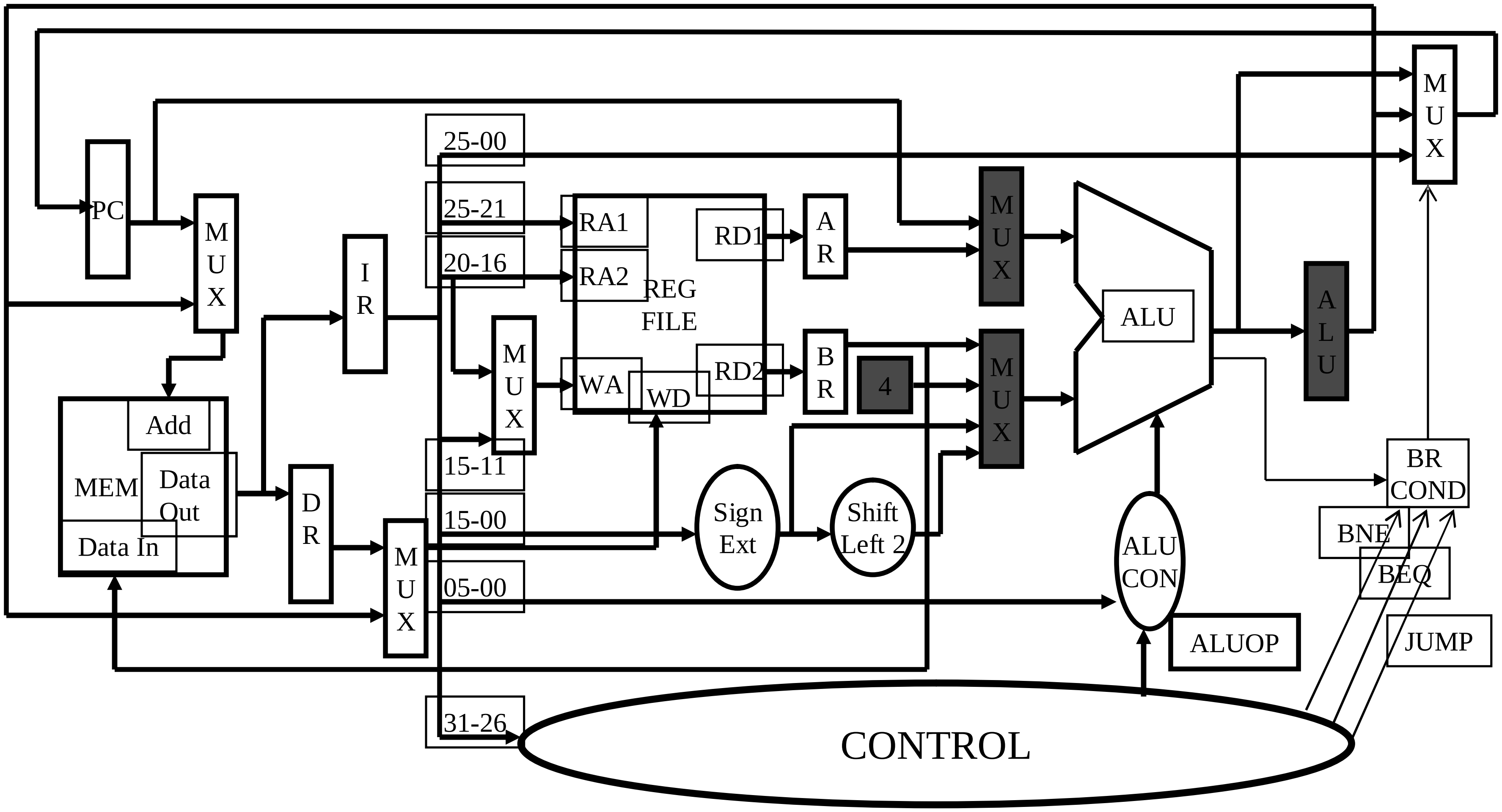
Review: finite state machines

- **Finite state machines:**
  - **a set of states and**
  - **next state function (determined by current state and the input)**
  - **output function (determined by current state and possibly input)**



  - **We'll use a Moore machine (output based only on current state)**

Multi-Cycle DataPath Operation



Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];
PC = PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- **ALU is performing one of three functions, based on instruction type**
- **Memory Reference:**

```
ALUOut = A + sign-extend(IR[15-0]);
```

- **R-type:**

```
ALUOut = A op B;
```

- **Branch:**

```
if (A==B) PC = ALUOut;
```

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR = Memory[ALUOut];
       or
Memory[ALUOut] = B;
```

- R-type instructions finish

```
Reg[IR[15-11]] = ALUOut;
```

The write actually takes place at the end of the cycle on the edge

Write-back step

- `Reg[IR[20-16]]= MDR;`

What about all the other instructions?

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

Instruction fetch	IR = Memory[PC]
	PC = PC + 4			

Instruction decode/register fetch	A = Reg [IR[25-21]]
	B = Reg [IR[20-16]]
	ALUOut = PC + (sign-extend (IR[15-0]) << 2)			

Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Instruction Format

31 26	25 21	20 16	15 11	10 6	5 0
LW	REG 1	REG 2	LOAD ADDRESS		OFFSET

31 26	25 21	20 16	15 11	10 6	5 0
SW	REG 1	REG 2	STORE ADDRESS		OFFSET

31 26	25 21	20 16	15 11	10 6	5 0
R-TYPE	REG 1	REG 2	DST	SHIFT AMOUNT	ADD/AND/OR/SLT

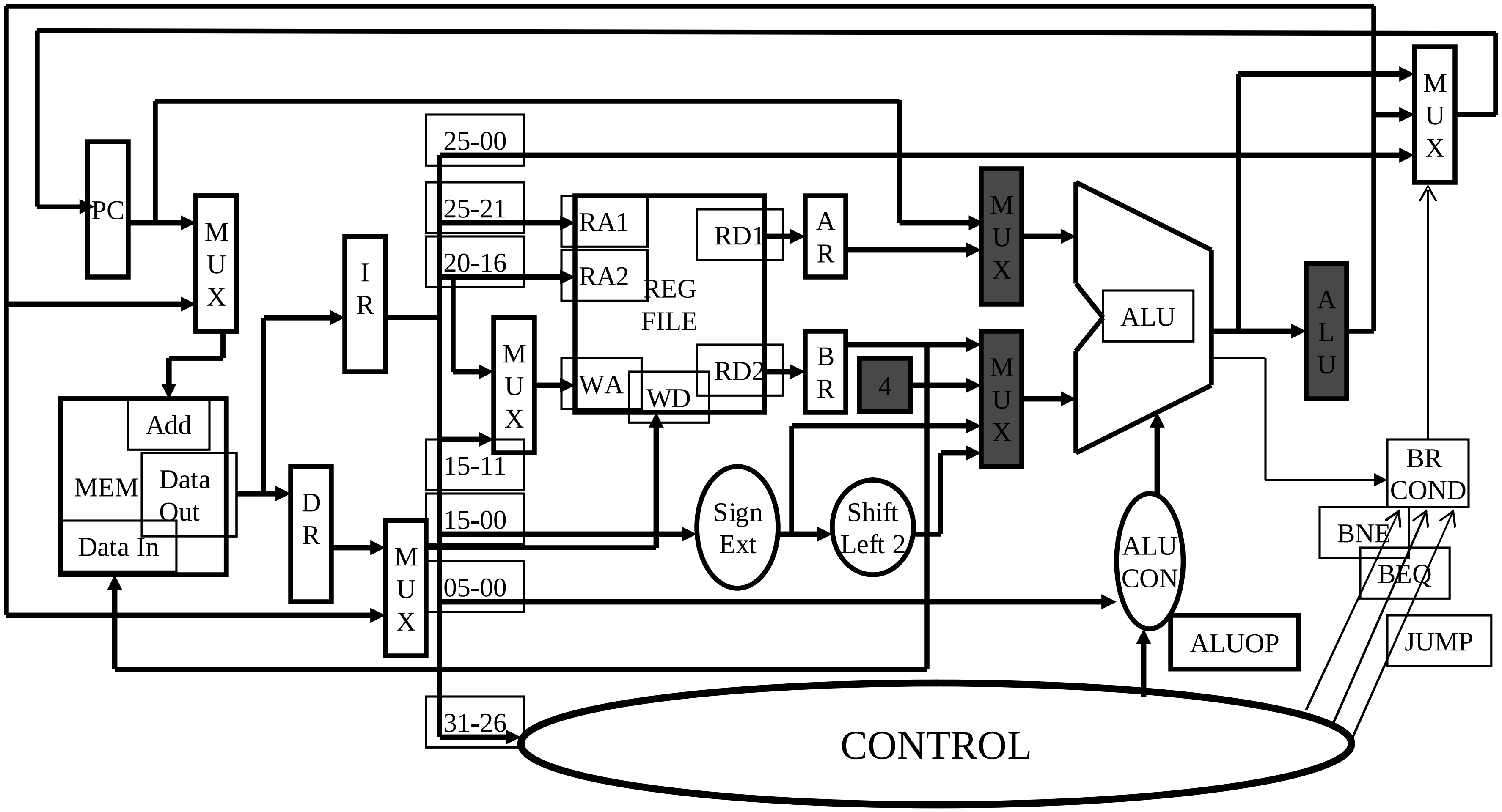
31 26	25 21	20 16	15 11	10 6	5 0
BEQ/BNE	REG 1	REG 2	BRANCH ADDRESS		OFFSET

31 26	25 21	20 16	15 11	10 6	5 0
JUMP	JUMP			ADDRESS	

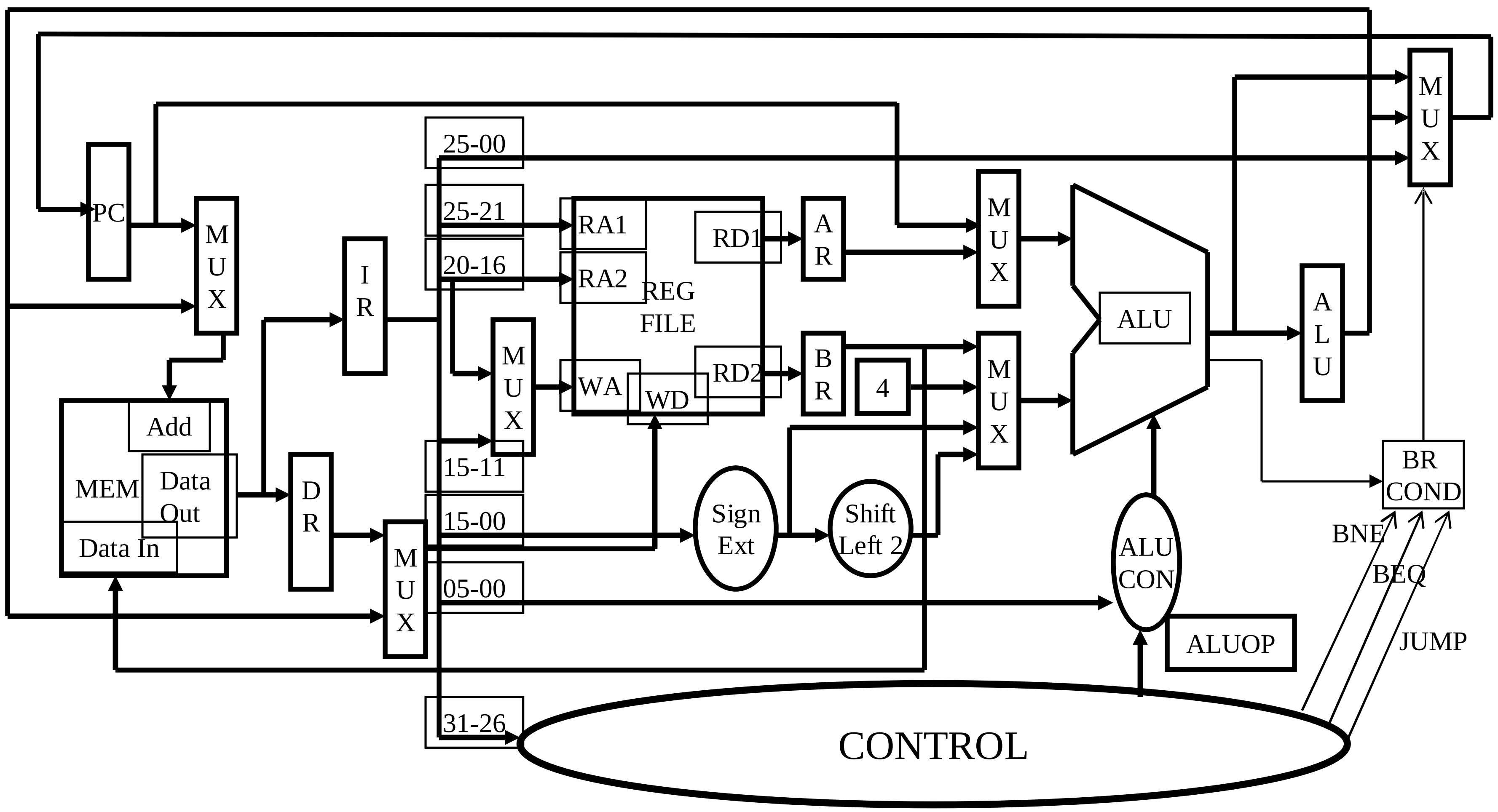
Operation for Each Instruction

LW:	SW:	R-Type:	BR-Type:	JMP-Type:
1. READ INST	1. READ INST	1. READ INST	1. READ INST	1. READ INST
2. READ REG 1 *READ REG 2*	2. READ REG 1 READ REG 2	2. READ REG 1 READ REG 2	2. READ REG 1 READ REG 2	2.
3. ADD REG 1 + OFFSET	3. ADD REG 1 + OFFSET	3. OPERATE on REG 1 / REG 2	3. SUB REG 2 from REG 1	3.
4. READ MEM	4. WRITE MEM	4.	4.	4.
5. WRITE REG2	5.	5. WRITE DST	5.	5.

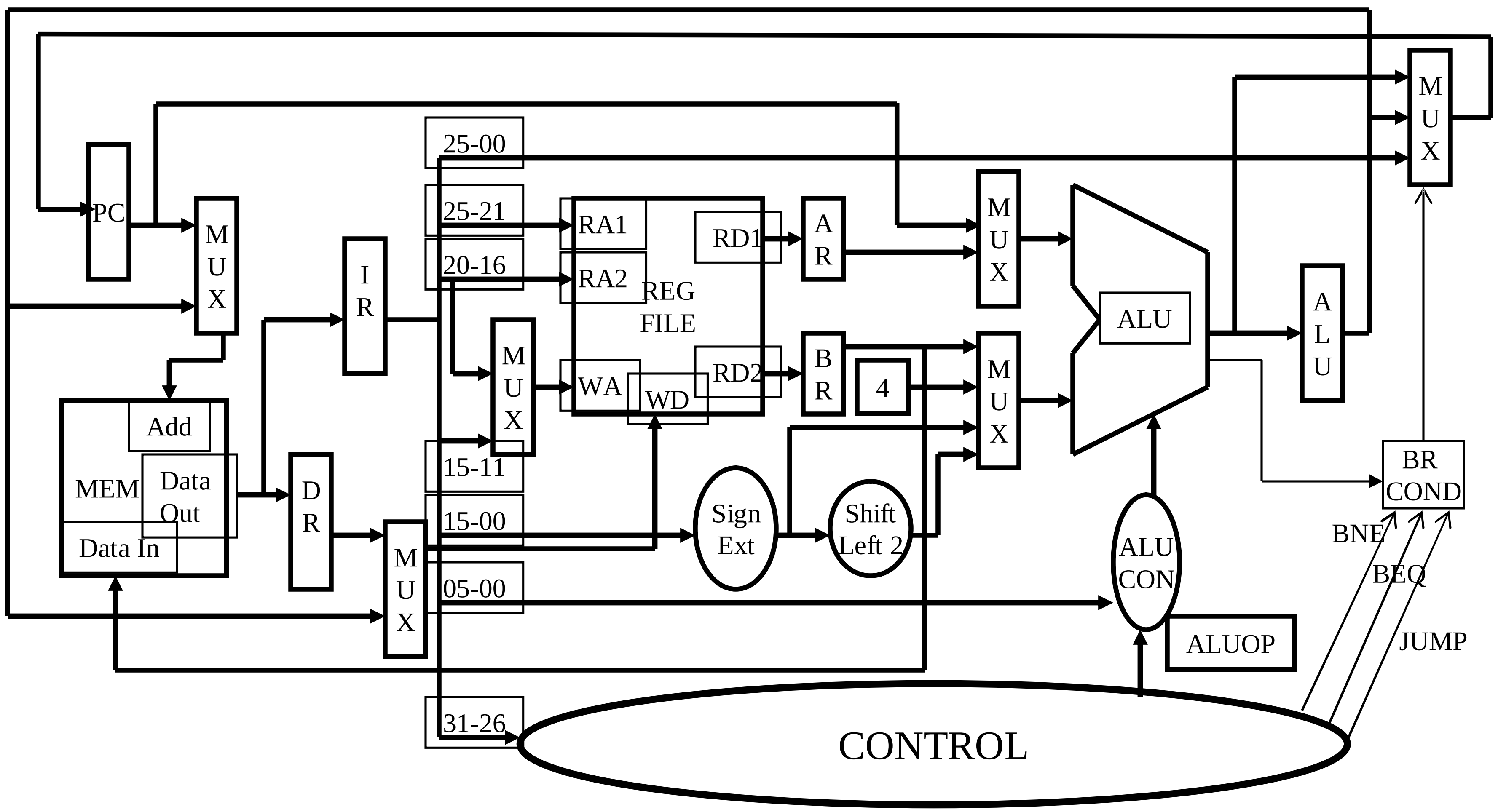
Multi-Cycle DataPath Operation



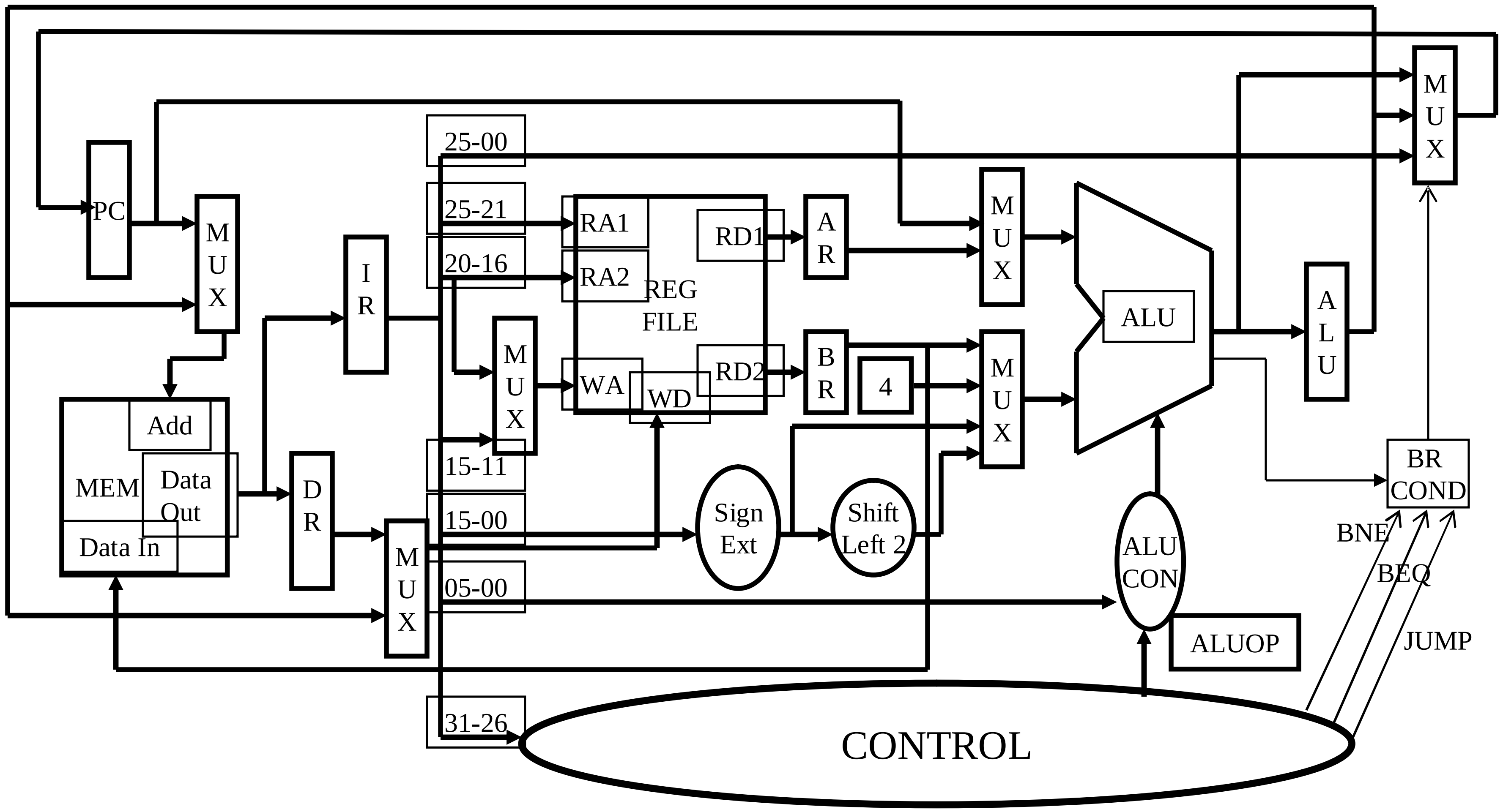
LW Operation on Multi-Cycle Data Path: C1



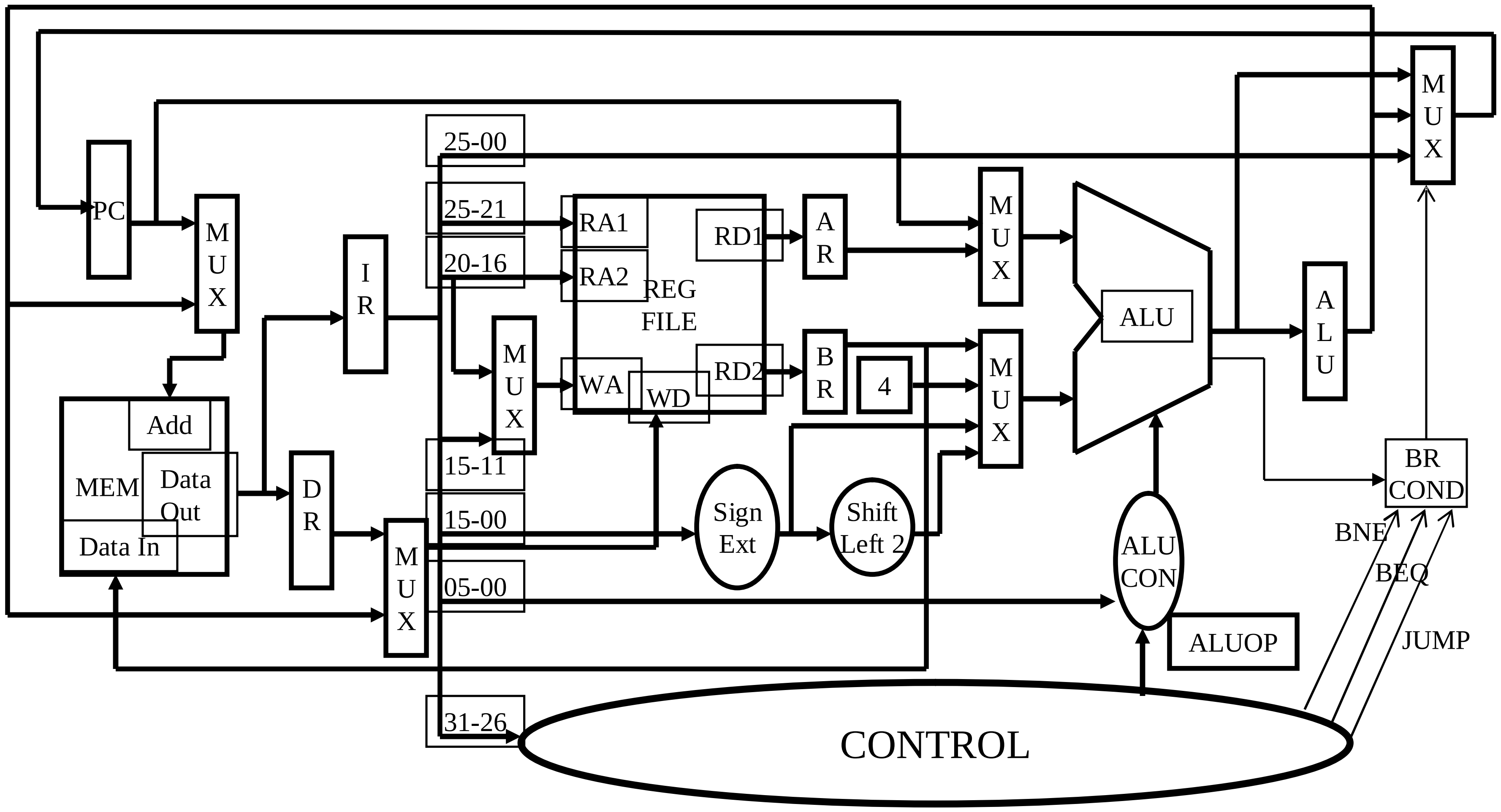
LW Operation on Multi-Cycle Data Path: C2



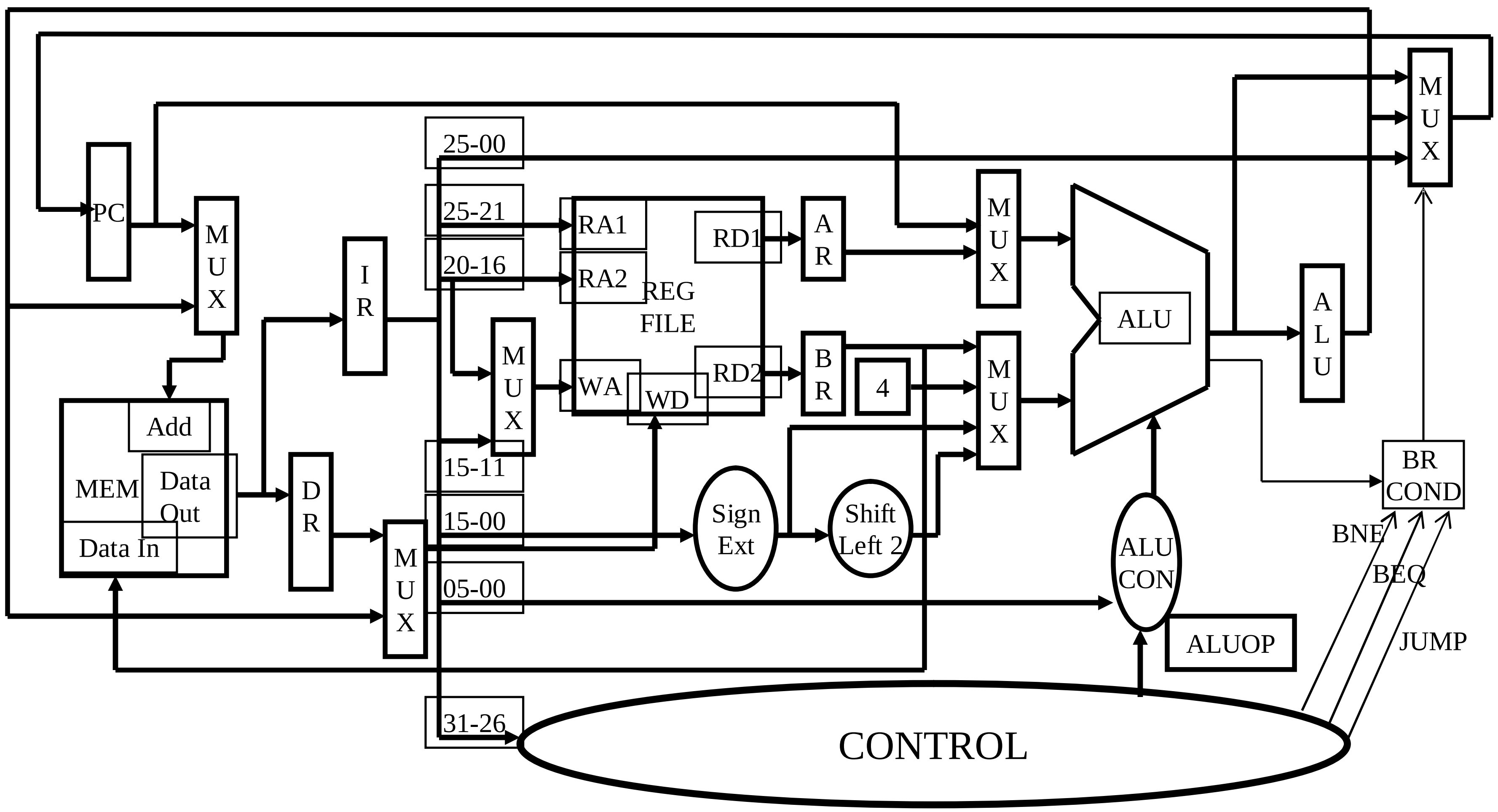
LW Operation on Multi-Cycle Data Path: C3



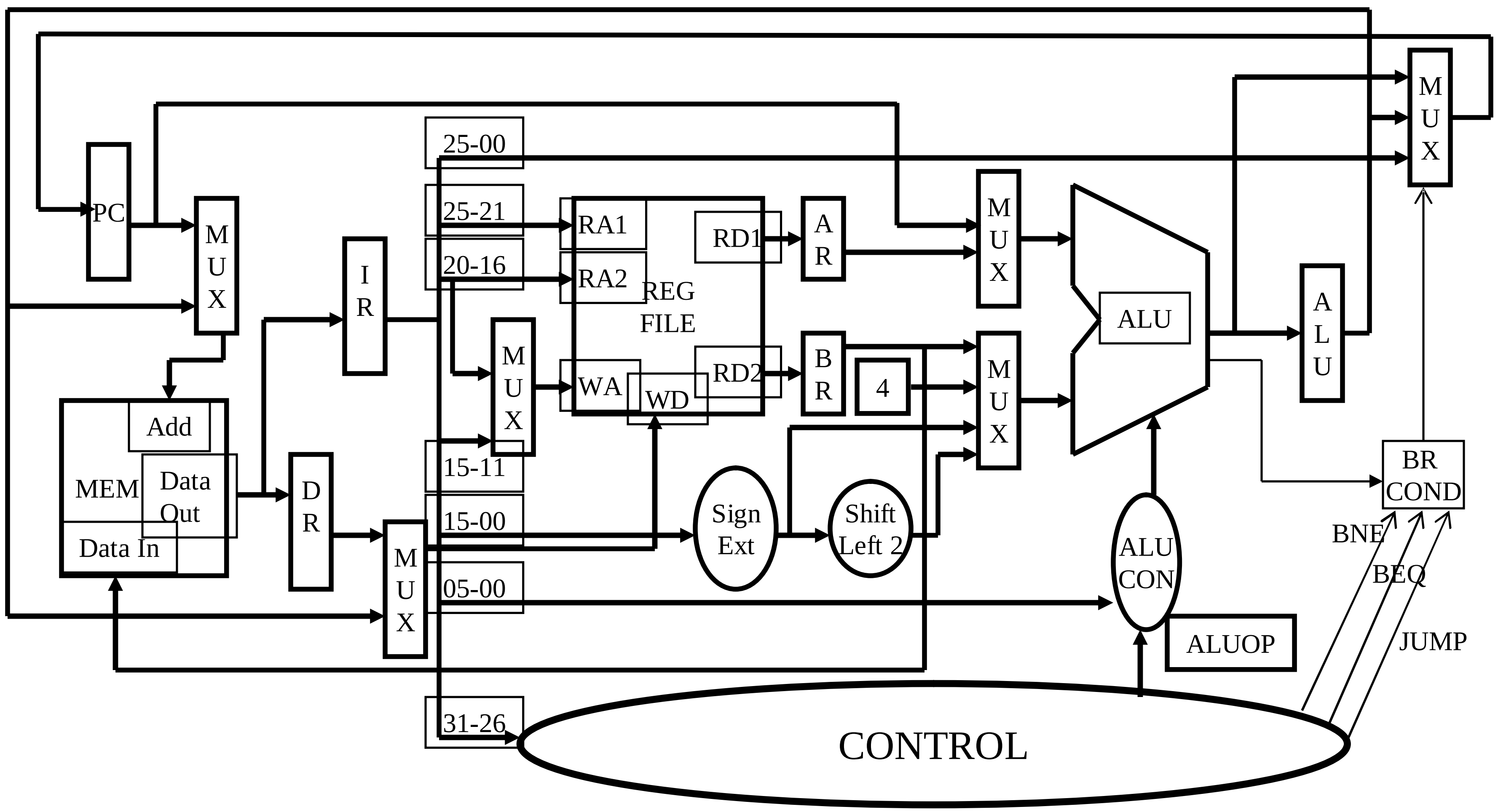
LW Operation on Multi-Cycle Data Path: C4



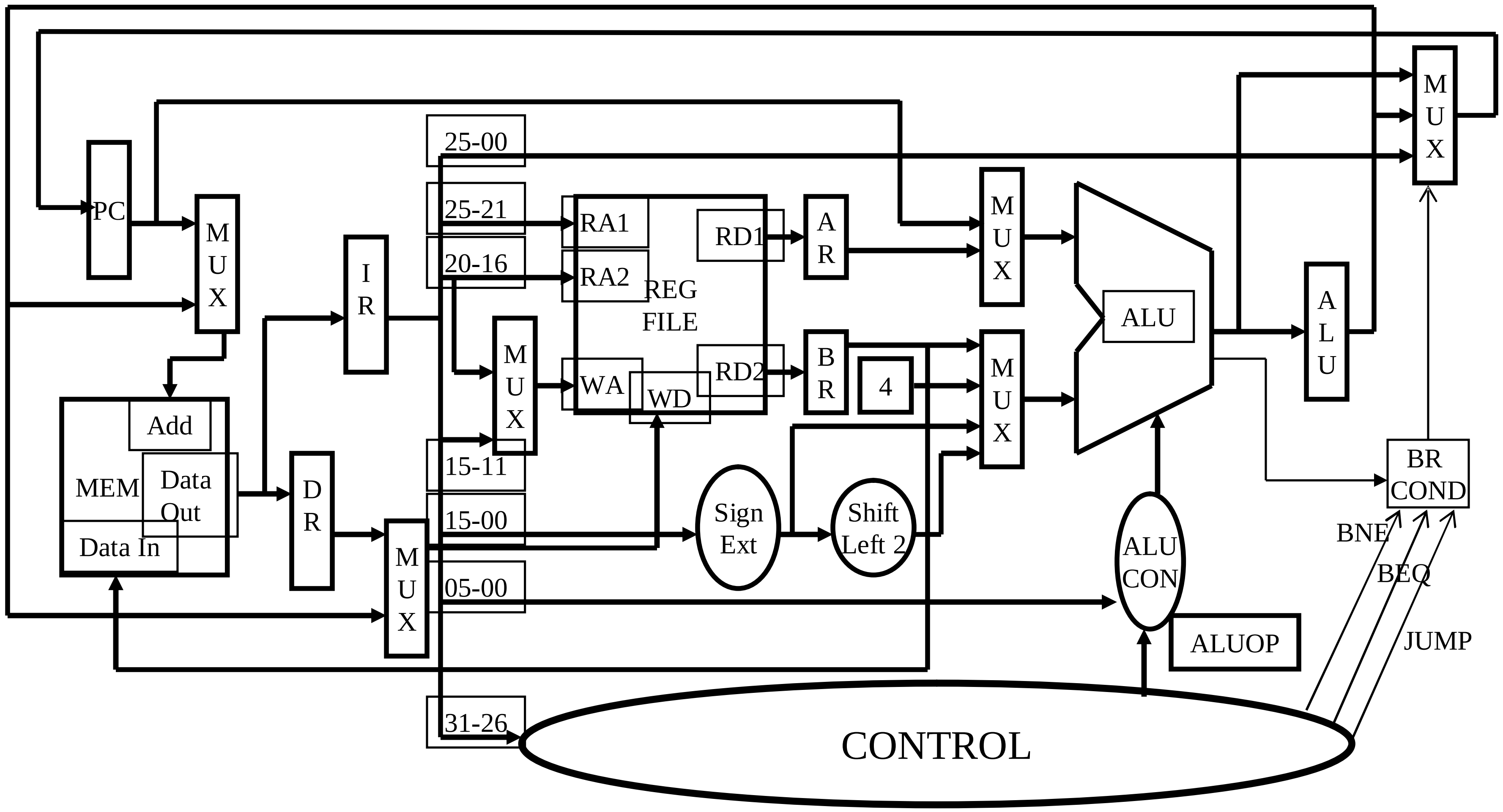
LW Operation on Multi-Cycle Data Path: C5



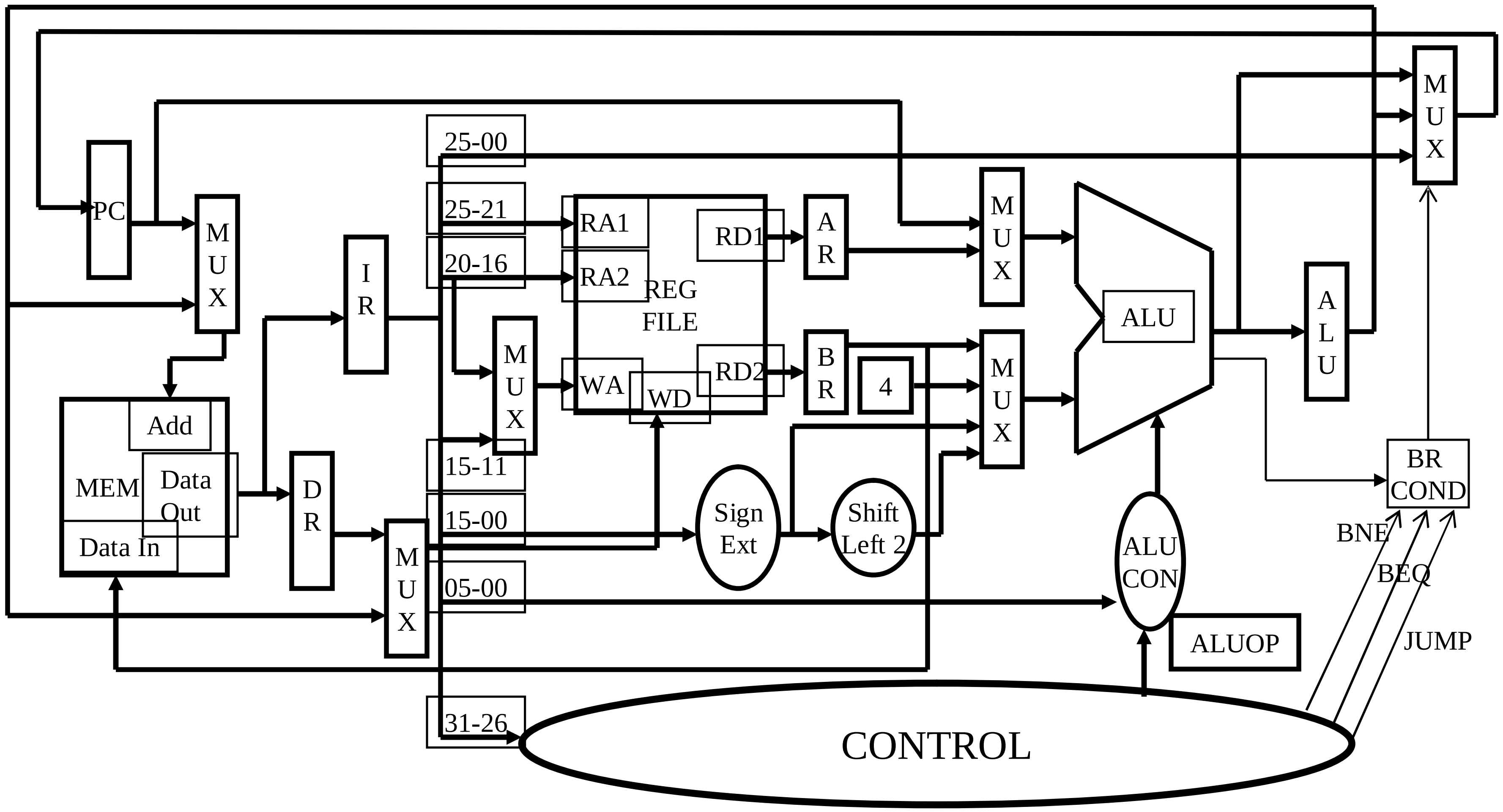
SW Operation on Multi-Cycle Data Path: C1



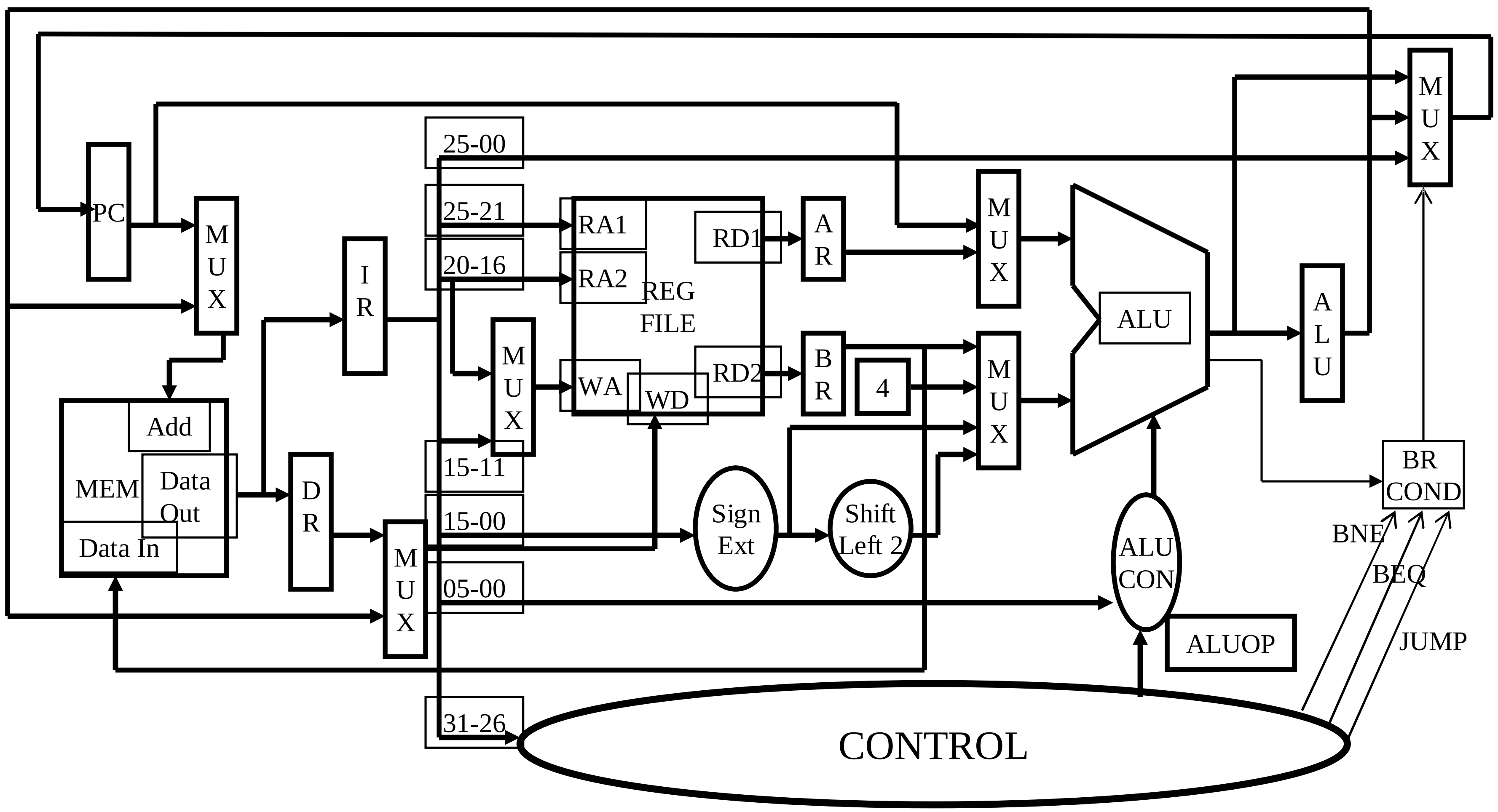
SW Operation on Multi-Cycle Data Path: C2



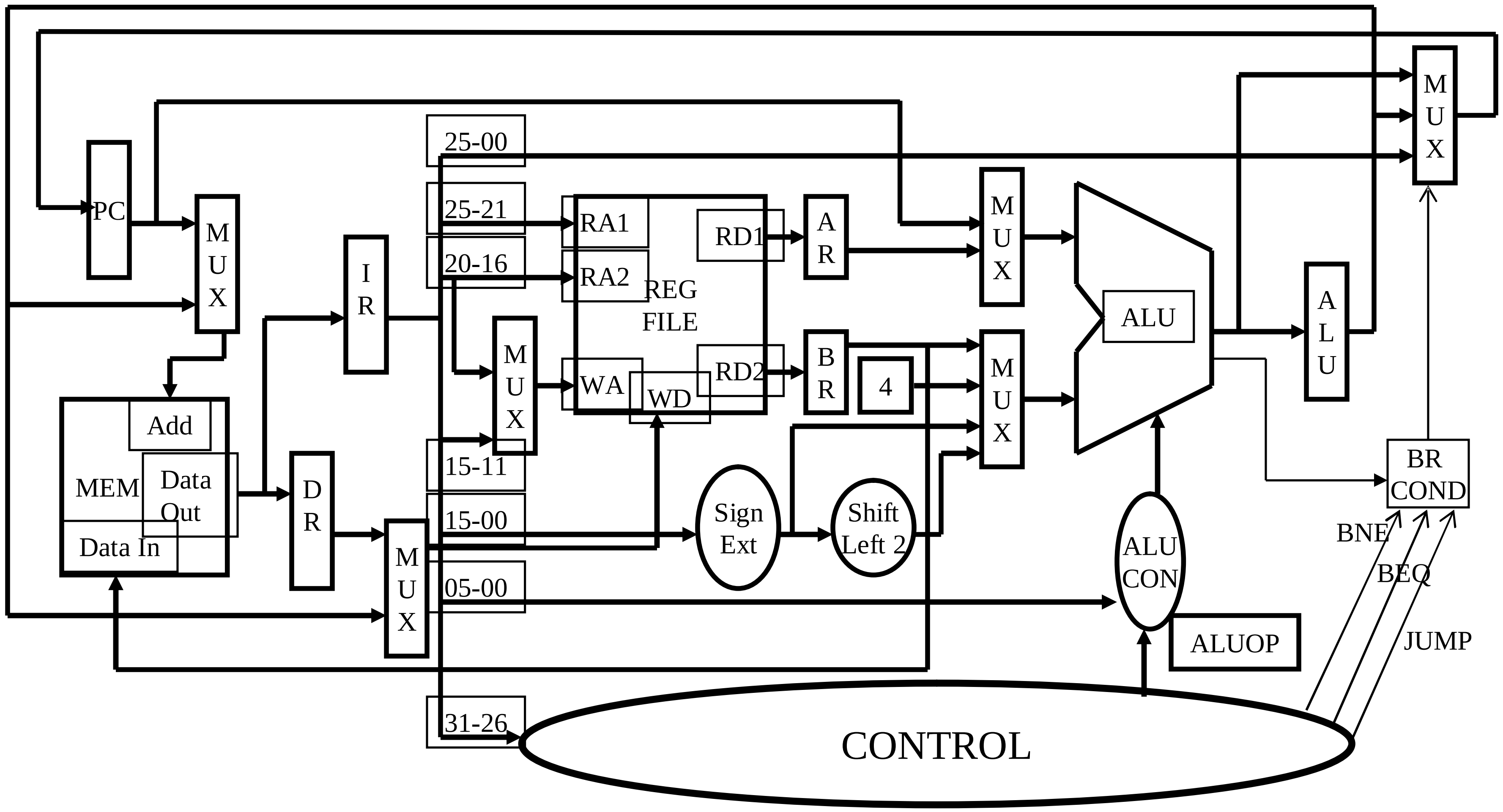
SW Operation on Multi-Cycle Data Path: C3



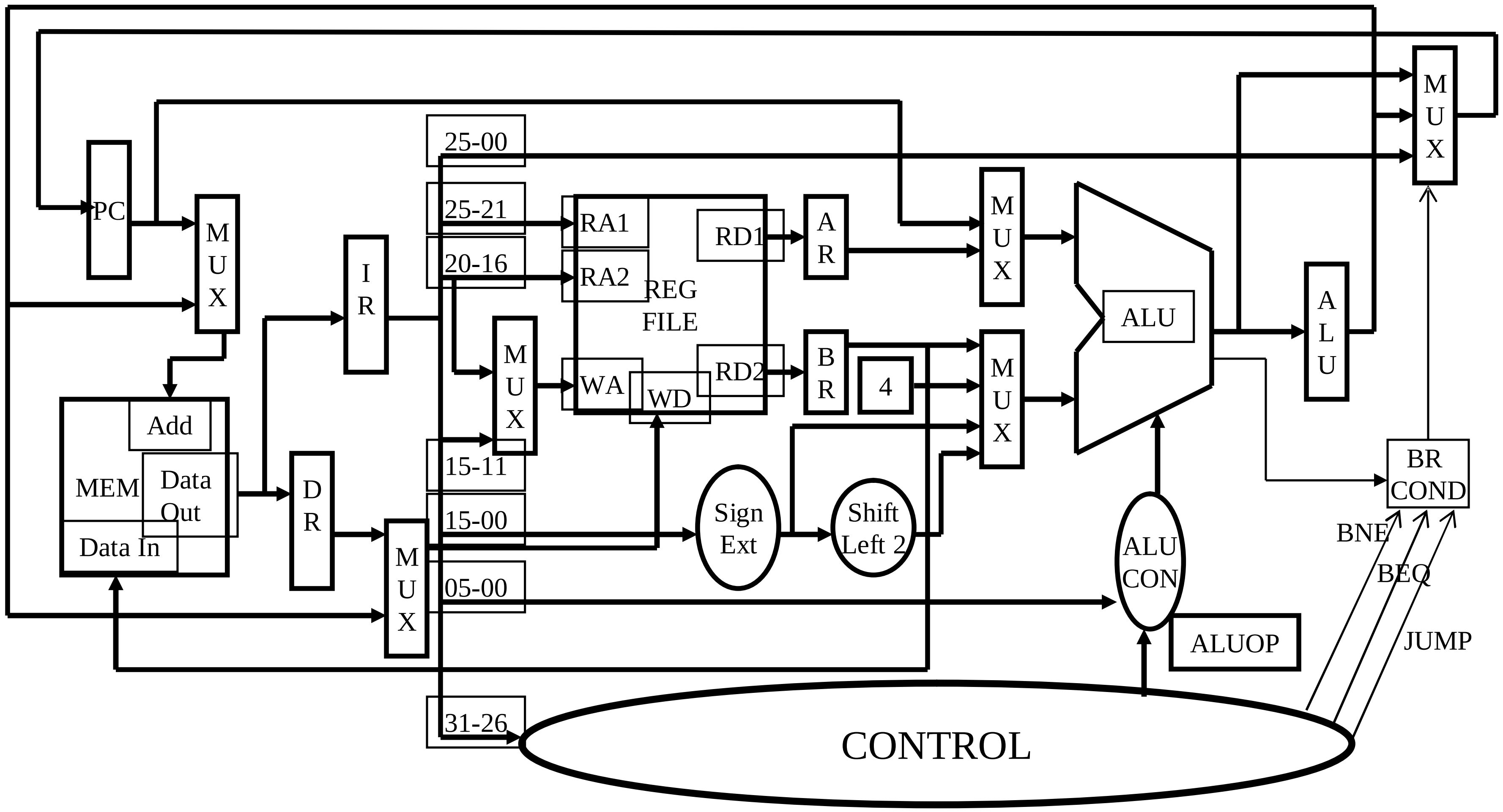
SW Operation on Multi-Cycle Data Path: C4



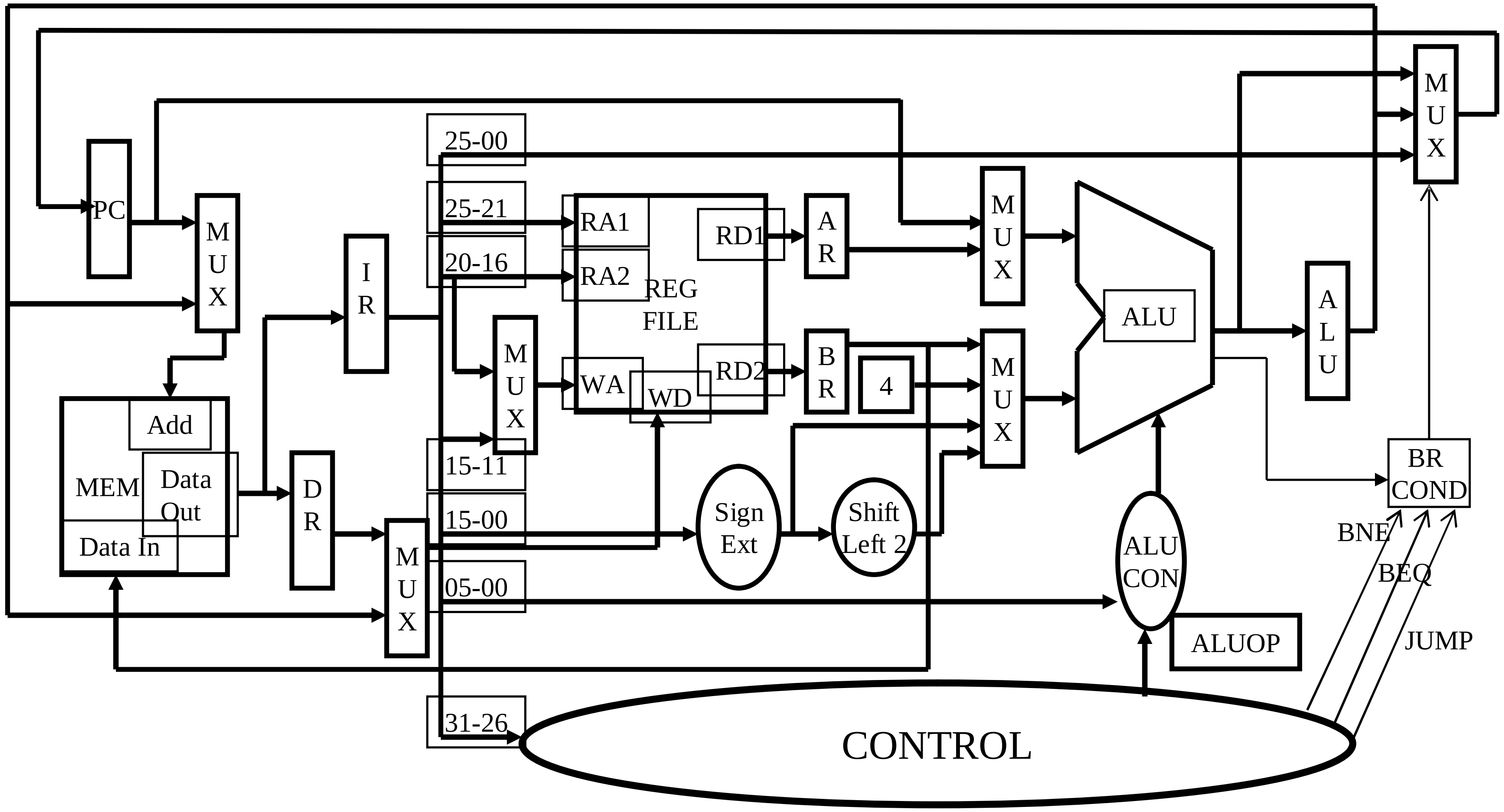
R-TYPE Operation on Multi-Cycle Data Path: C1



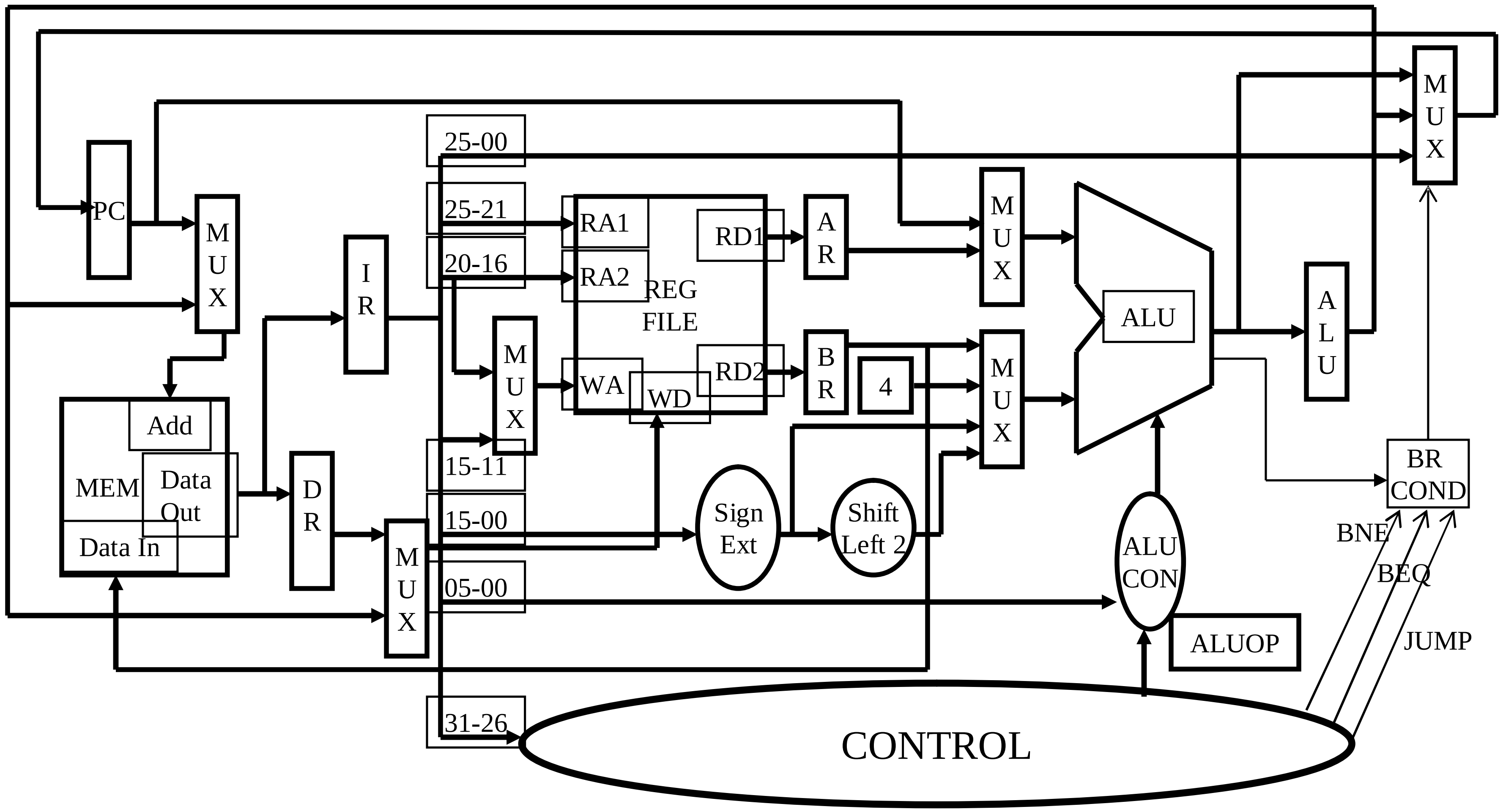
R-TYPE Operation on Multi-Cycle Data Path: C2



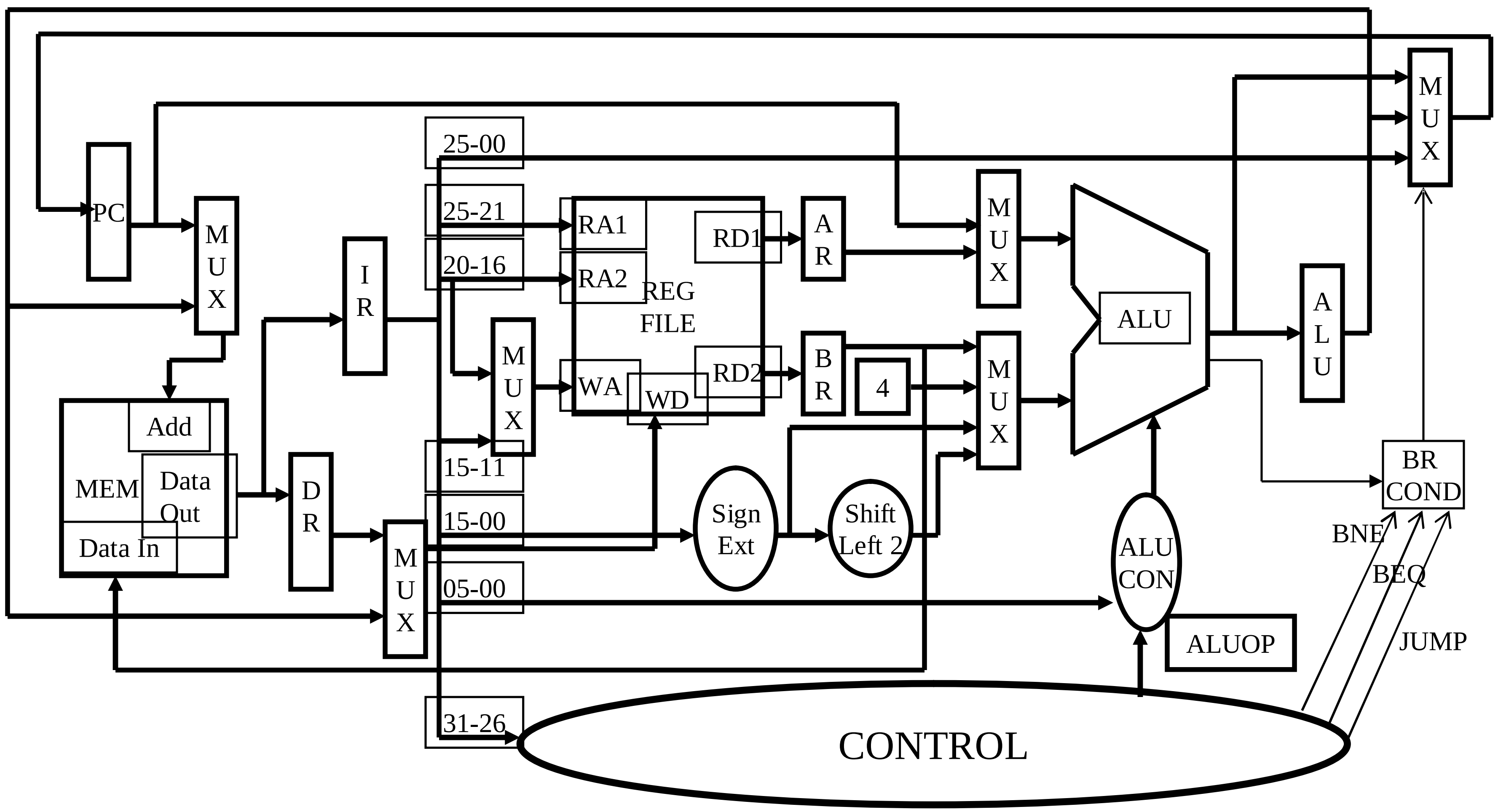
R-TYPE Operation on Multi-Cycle Data Path: C3



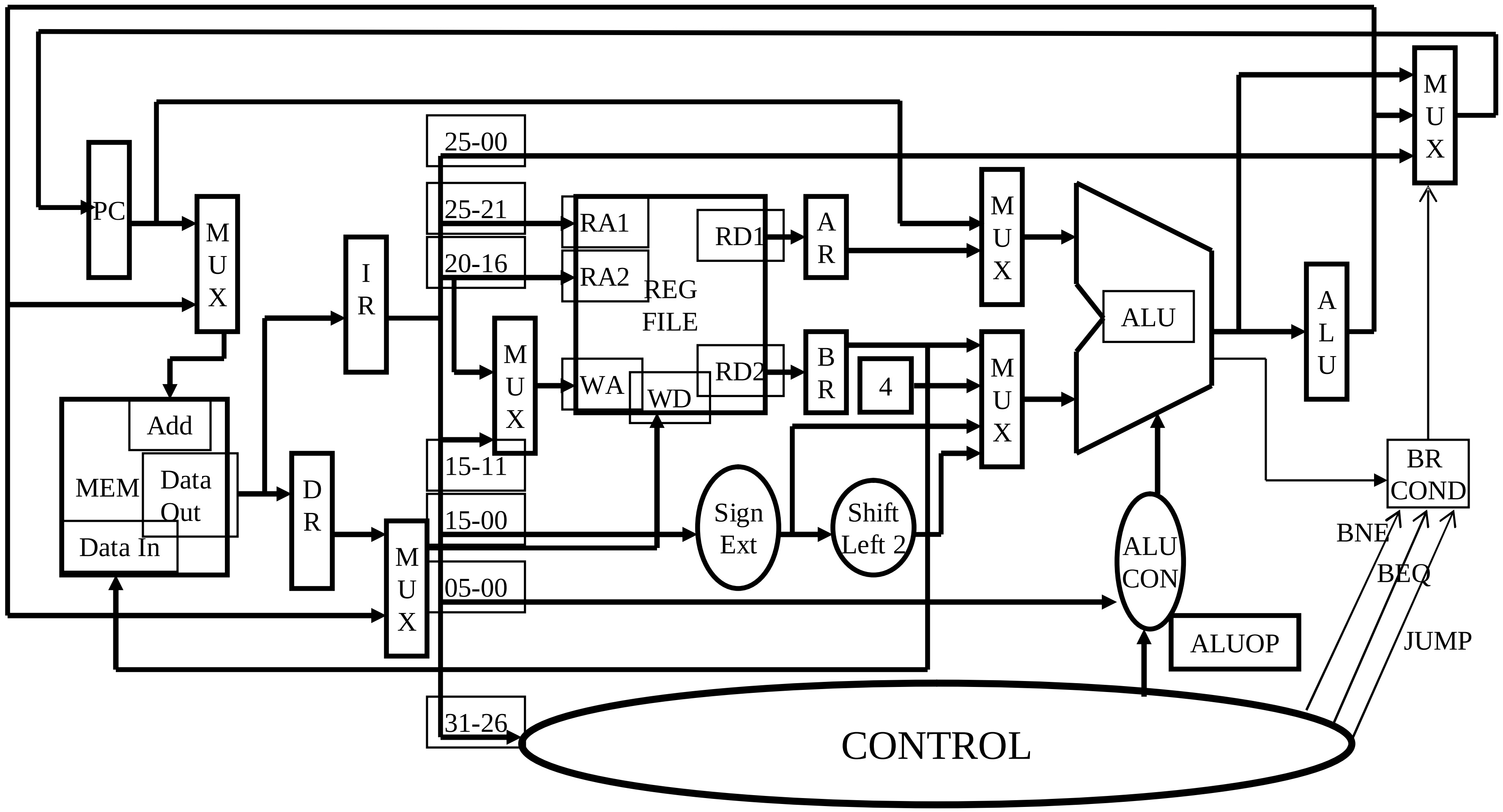
R-TYPE Operation on Multi-Cycle Data Path: C4



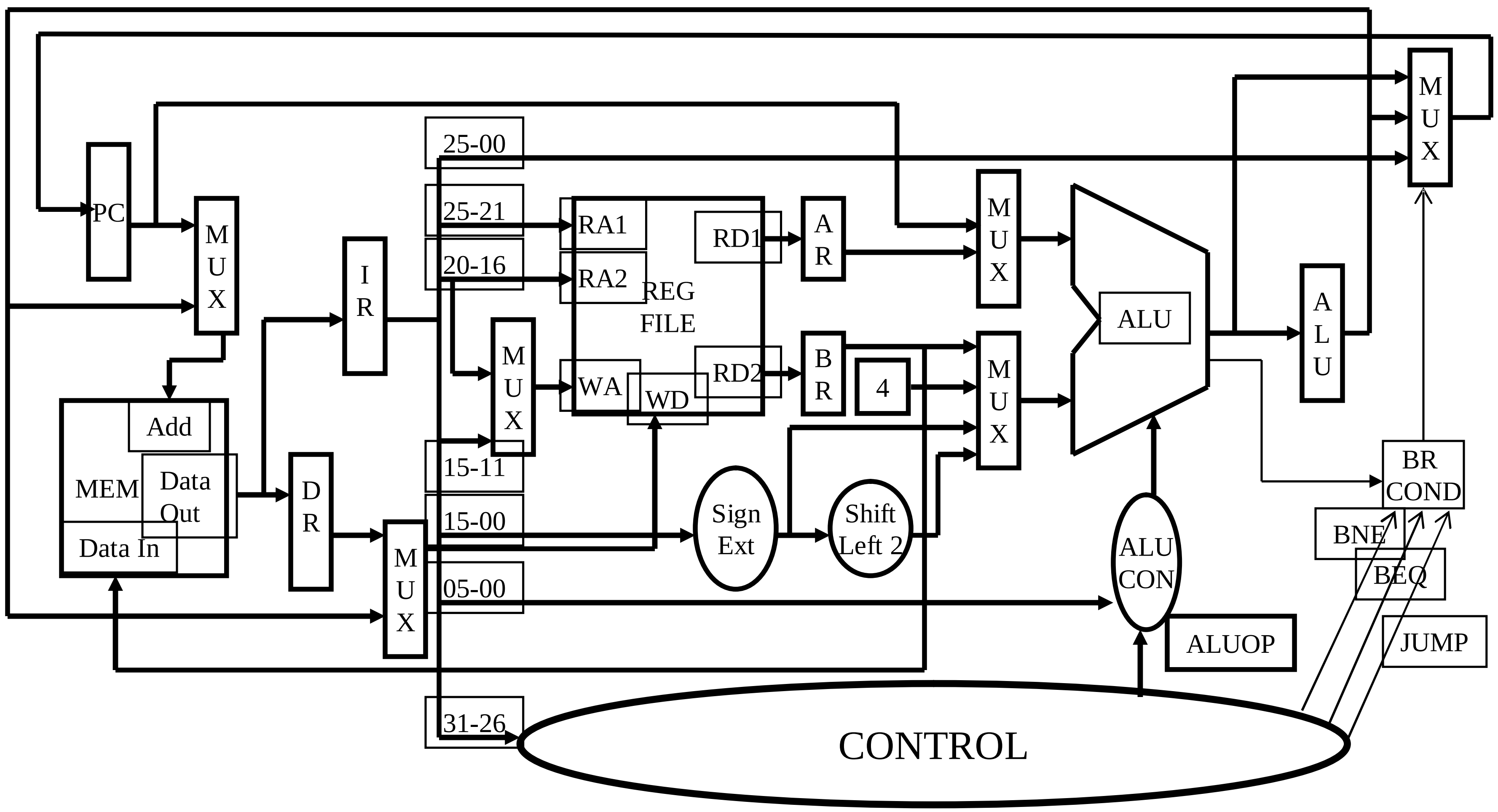
BR Operation on Multi-Cycle Data Path: C1



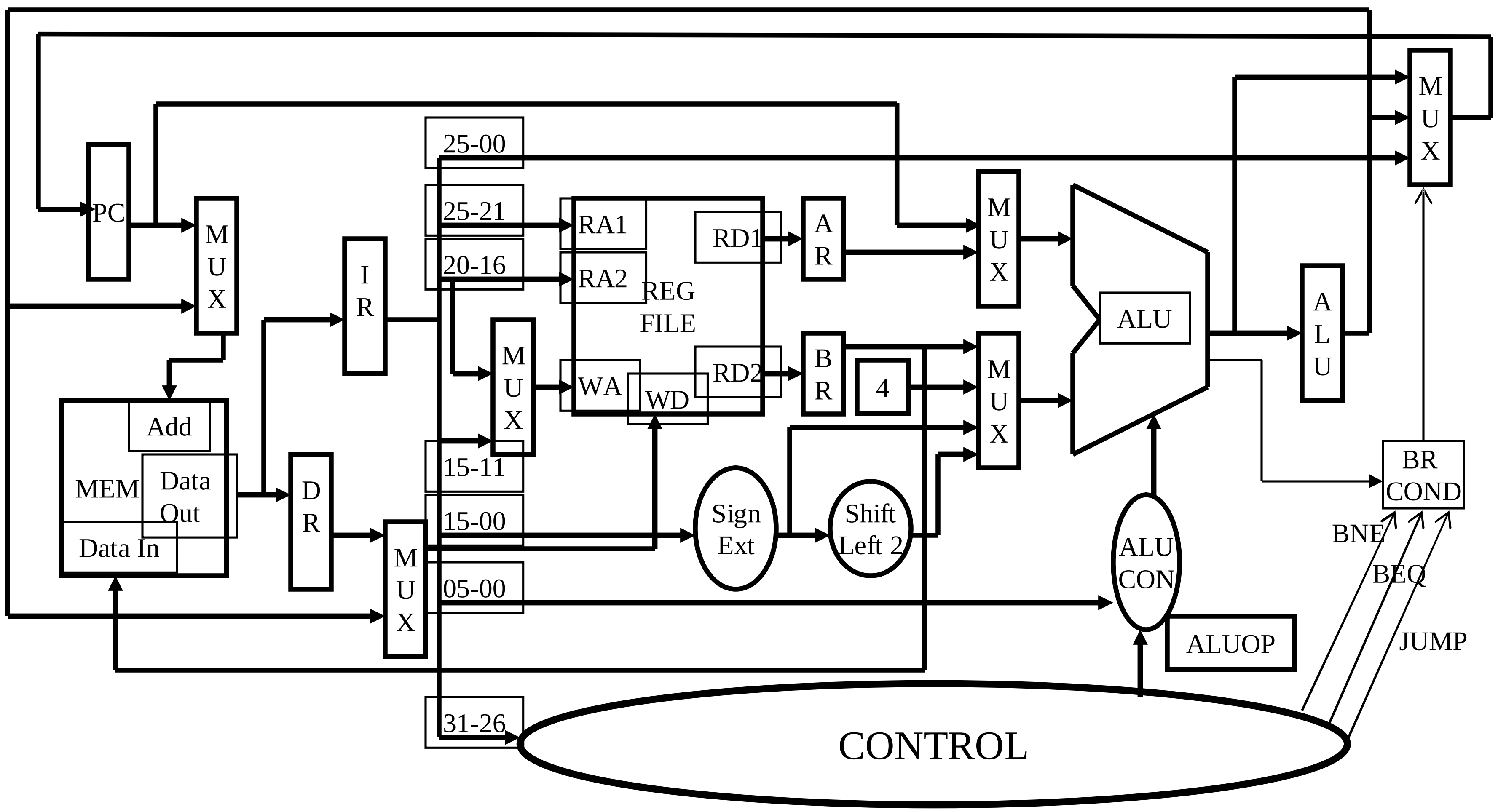
BR Operation on Multi-Cycle Data Path: C2



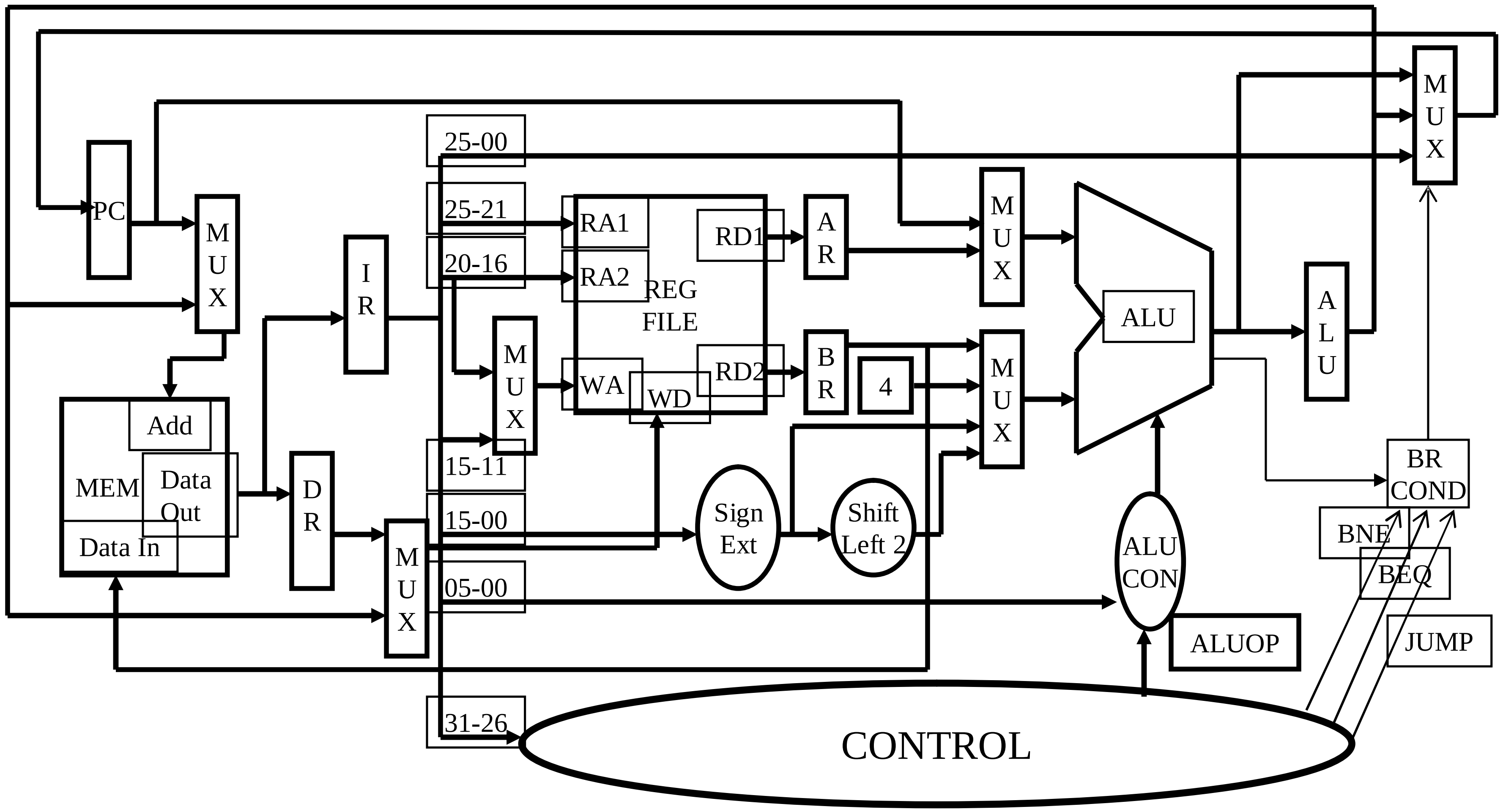
BR Operation on Multi-Cycle Data Path: C3



JUMP Operation on Multi-Cycle Data Path: C1



JUMP Operation on Multi-Cycle Data Path: C2



Simple Questions

- **How many cycles will it take to execute this code?**

```
          lw $t2, 0($t3)
          lw $t3, 4($t3)
          beq $t2, $t3, Label     #assume not
          add $t5, $t2, $t3
          sw $t5, 8($t3)
Label:    ...
```

- **What is going on during the 8th cycle of execution?**
- **In what cycle does the actual addition of `$t2` and `$t3` takes place?**

Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

- Use the information we’ve accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use micro-programming

- Implementation can be derived from specification

Deciding the Control

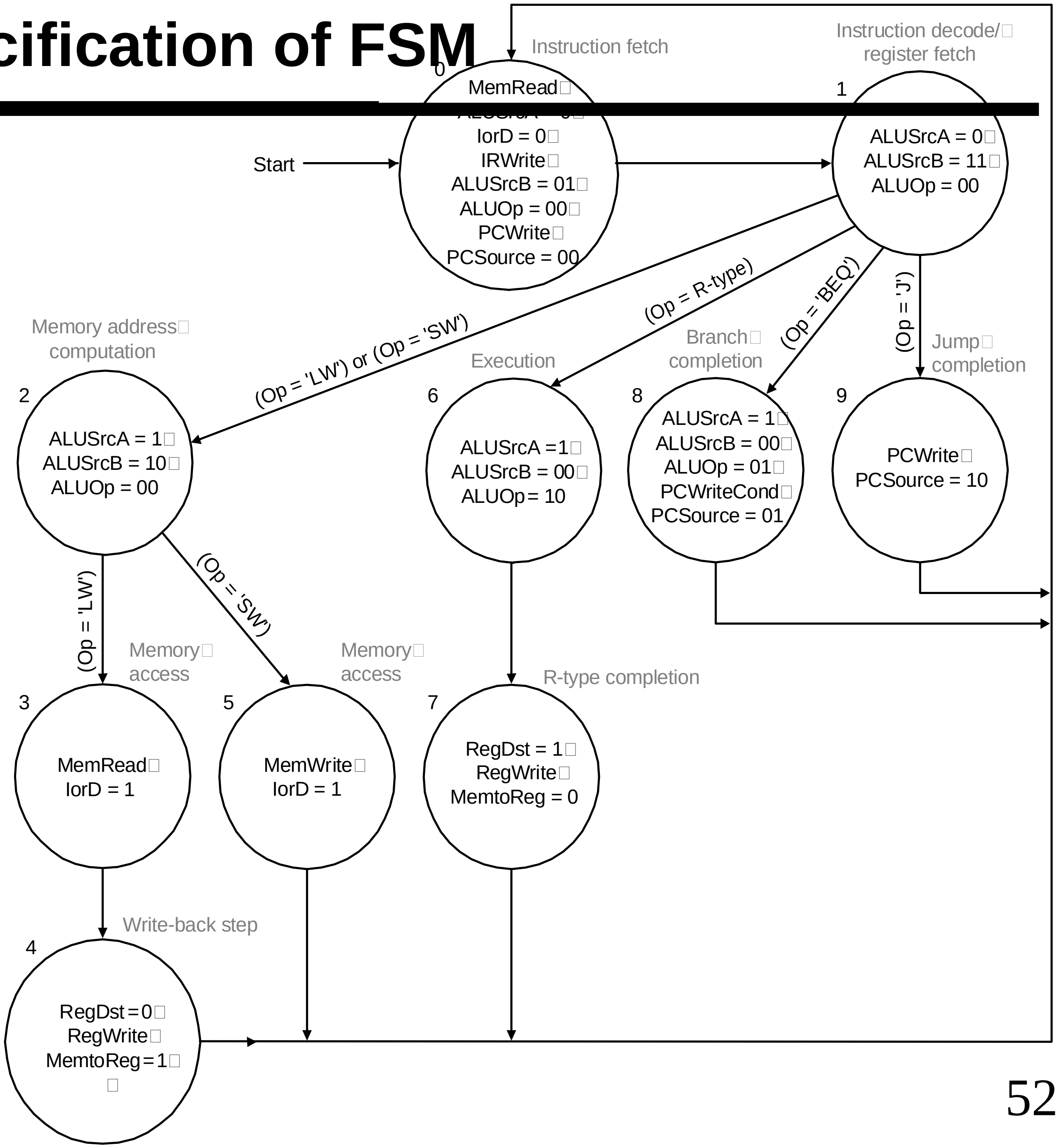
- In each clock cycle, decide all the action that needs to be taken
- The control signal can be 0 and 1 or x (don’t care)
- Make a signal an x if you can to reduce control
- An action that may destroy any useful value be not allowed
- Control Signal required
  - ALU: SRC1 (1 bit), SRC2(2 bits),
  - operation (Add, Sub, or from FC)
  - Memory: address (I or D), read, write, data in IR or MDR
  - Register File: address rt/rd, data (MDR/ALUOUT), read, write
  - PC: PCwrite, PCwrite-conditional, Data (PC+4, branch, jump)
- Control signal can be implied (register file read are values in A and B registers (actually A and B need not be registers at all)
- Explicit control vs indirect control (derived based on input like instruction being executed, or function code field) bits

Graphical Specification of FSM

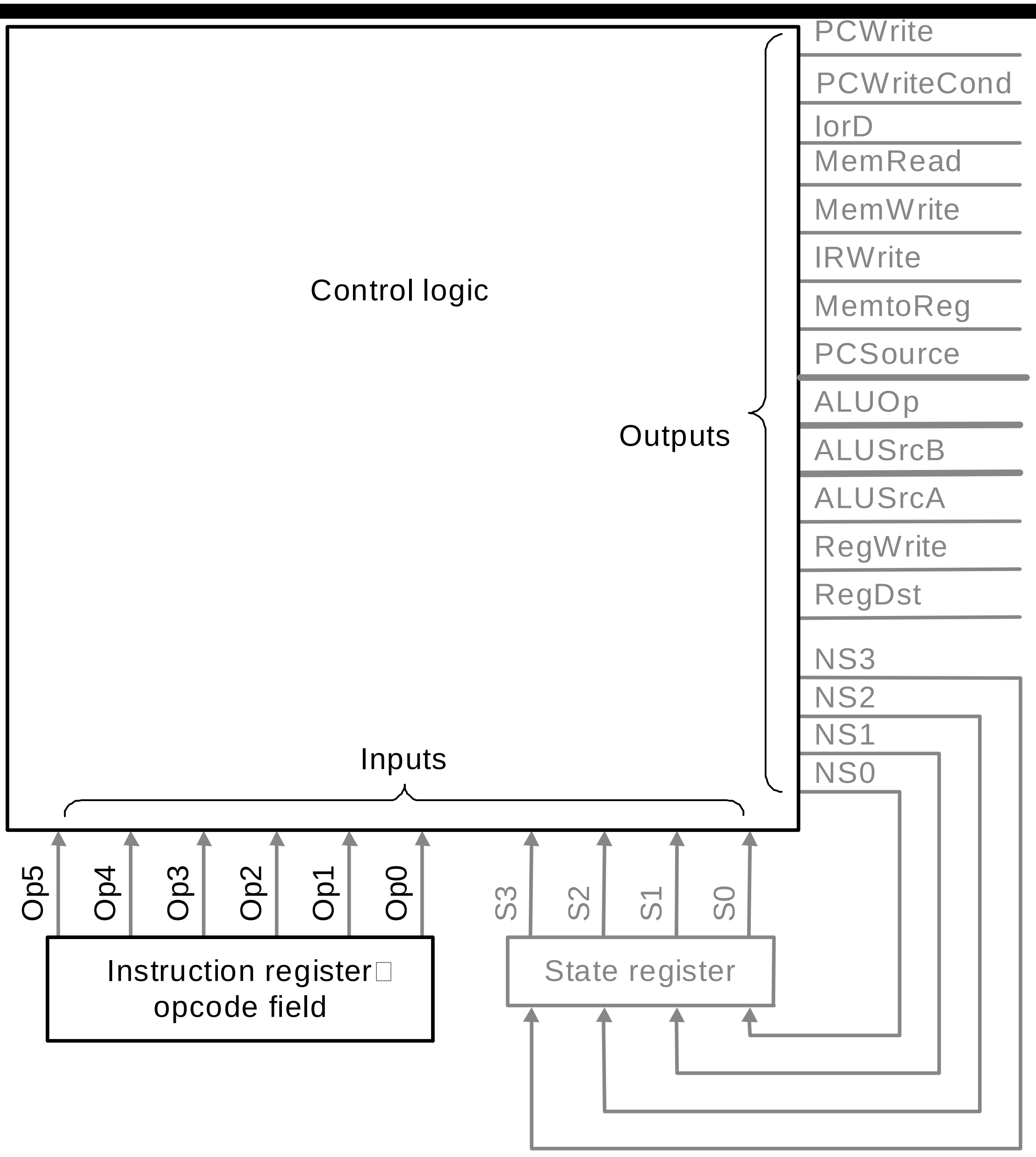
- How many state bits will we need?

- 4 bits.

- Why?

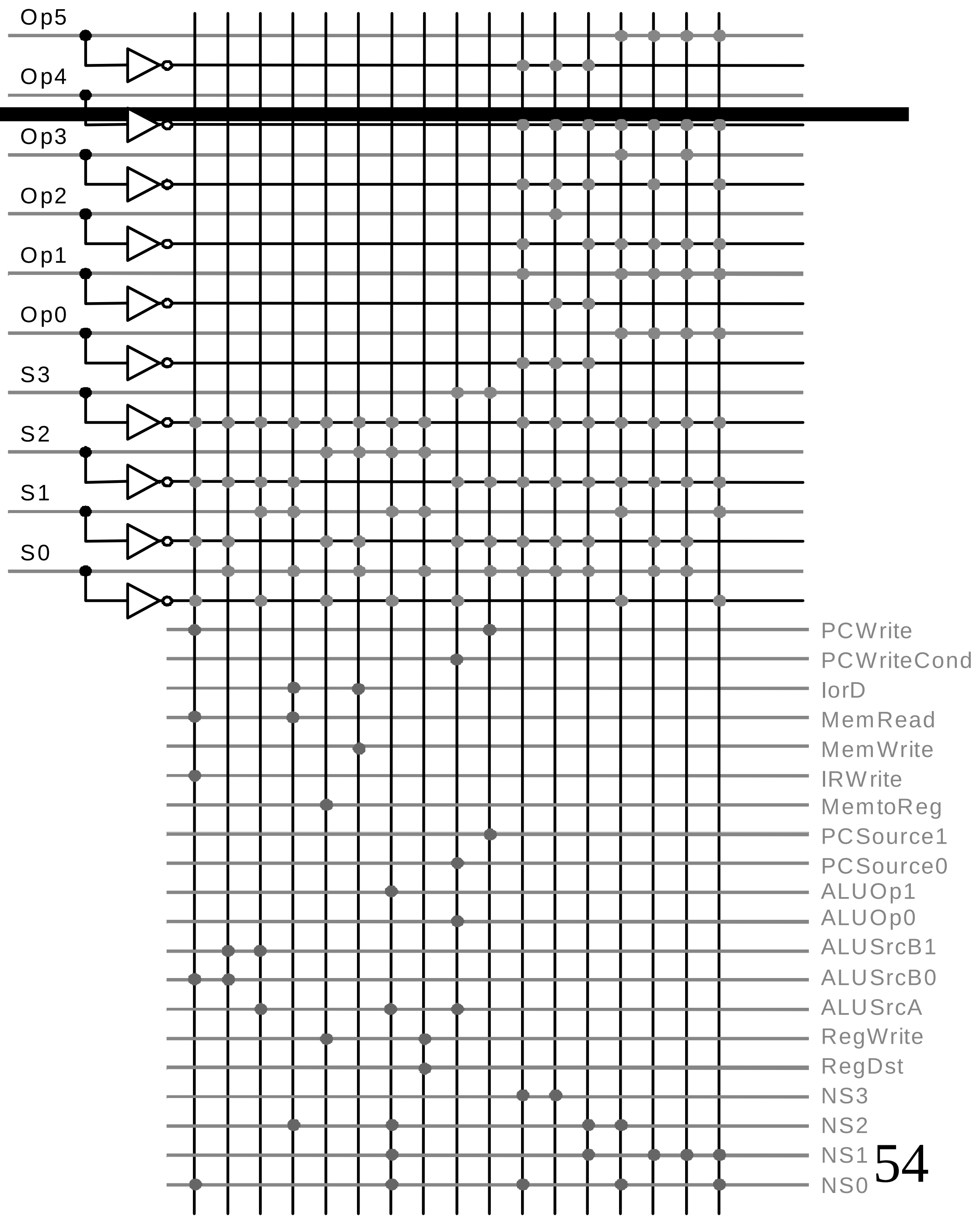


Finite State Machine: Control Implementation



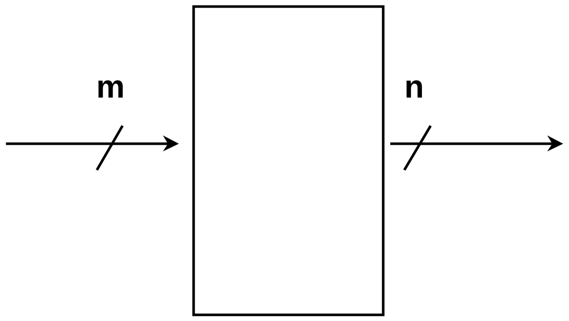
PLA Implementation

- **If I picked a horizontal or vertical line could you explain it?**



ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address 2^m entries in the ROM.
  - our outputs are the bits of data that the address points to.



0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

m is the "height", and n is the "width"

ROM Implementation

- **How many inputs are there?**
  6 bits for opcode, 4 bits for state = 10-bit
  (i.e., 2^{10} = 1024 different addresses)
- **How many outputs are there?**
  16 datapath-control outputs, 4 state bits = 20 bits
- **ROM is 2^{10} x 20 = 20K bits (an unusual size)**

- **Rather wasteful, since for lots of the entries, the outputs are the same**
  — i.e., opcode is often ignored

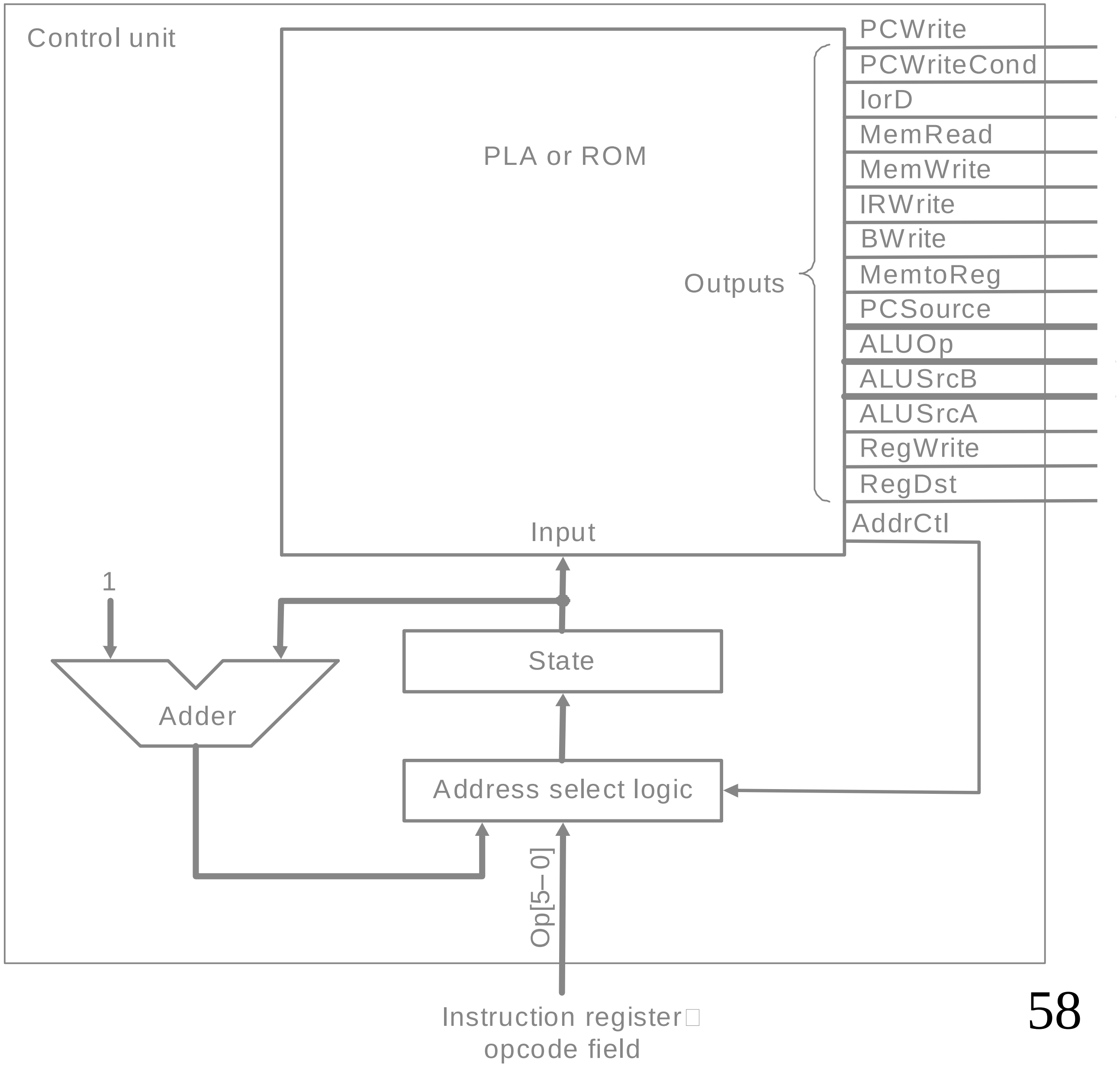
ROM vs PLA

- Break up the table into two parts
  - — 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM
  - — 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM
  - — Total: 4.3K bits of ROM
- PLA is much smaller
  - — can share product terms
  - — only need entries that produce an active output
  - — can take into account don't cares
- Size is (#inputs ´ #product-terms) + (#outputs ´ #product-terms)

  For this example = (10x17)+(20x17) = 460 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

Another Implementation Style

- **Complex instruction: the "next state" is often current state + 1**



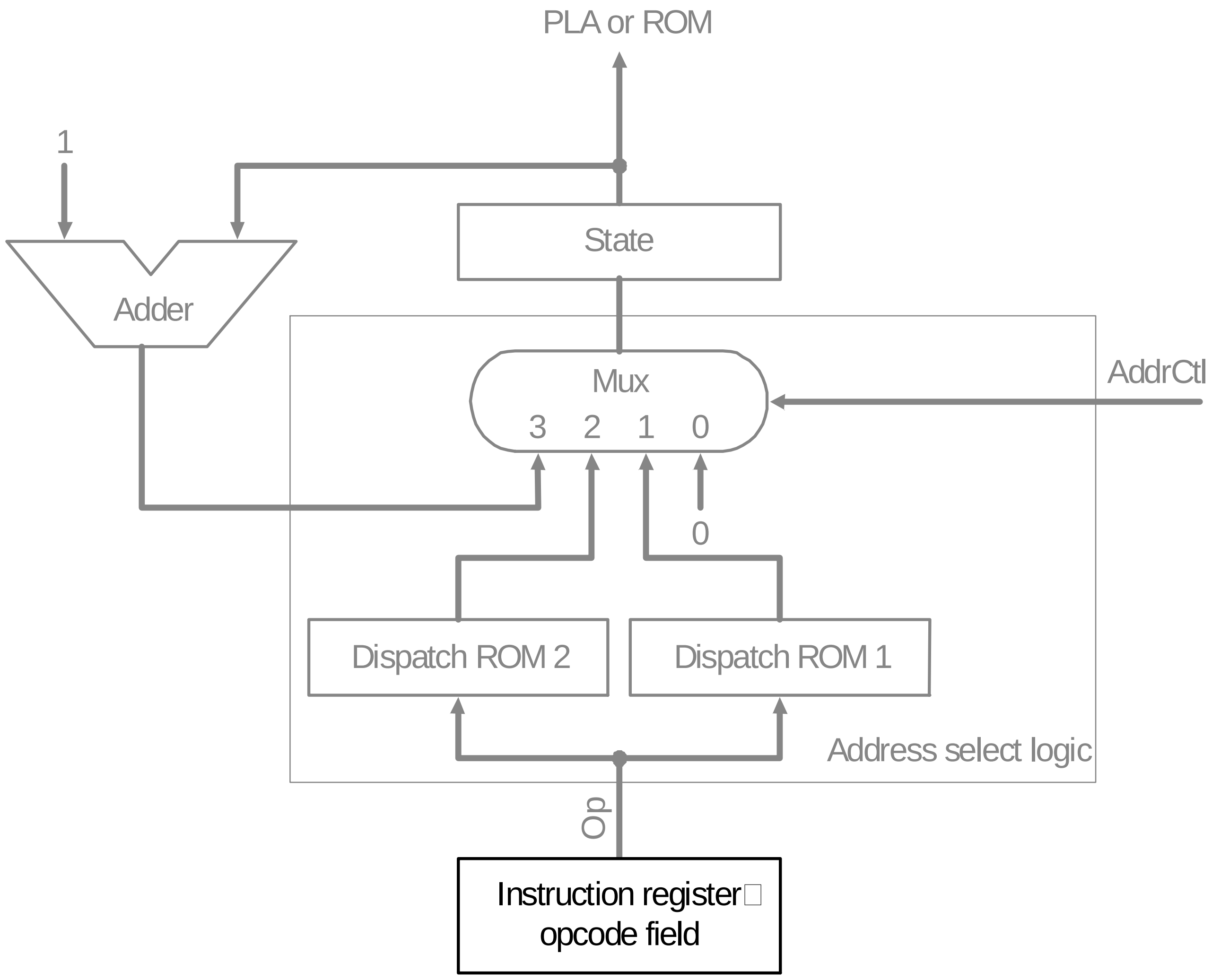
Details-1

Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	`jmp`	1001
000100	`beq`	1000
100011	`lw`	0010
101011	`sw`	0010

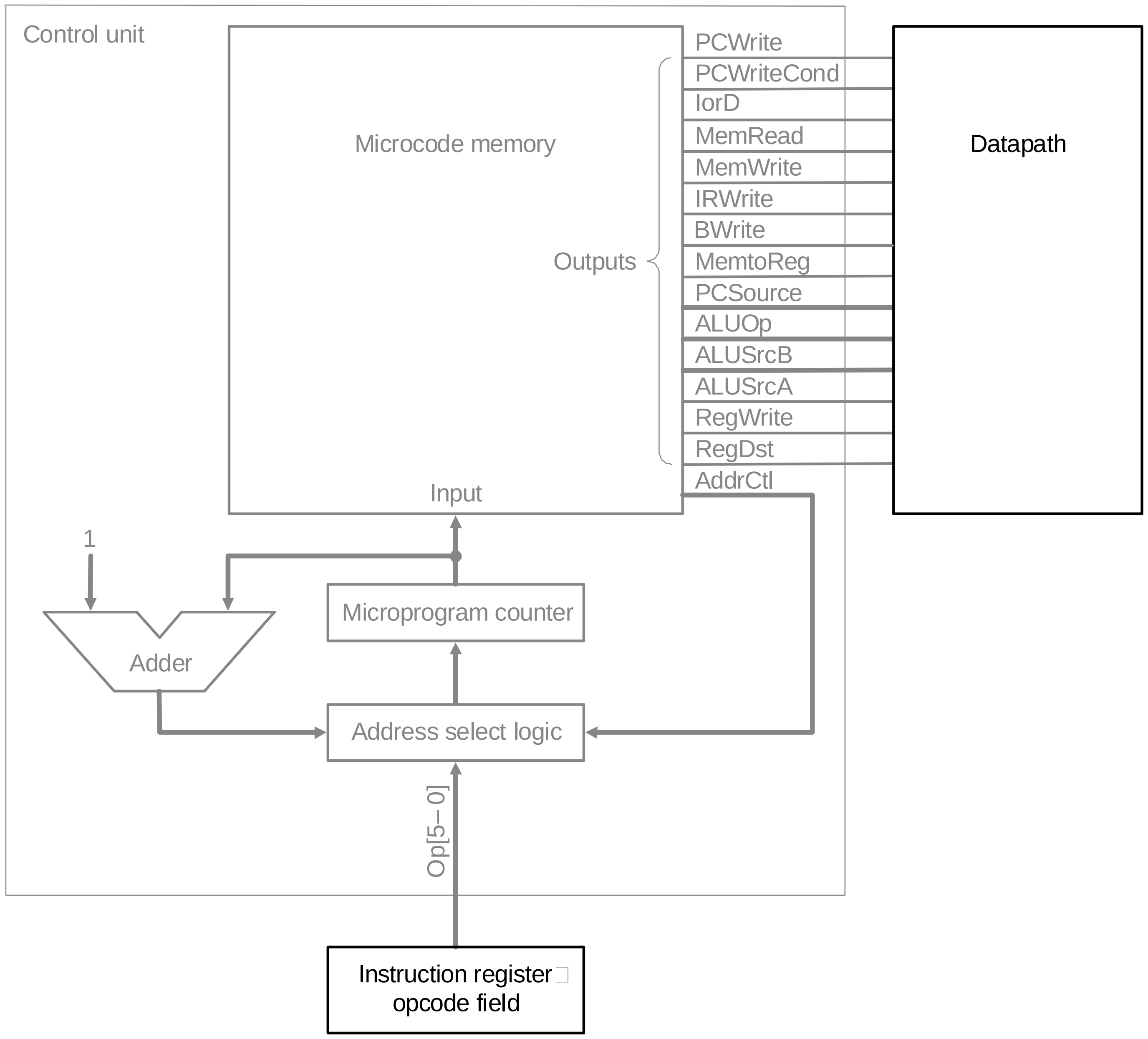
Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	`lw`	0011
101011	`sw`	0101

State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

Details-2


PLA or ROM
1
State
Adder
Mux
3 2 1 0
AddrCtl
0
Dispatch ROM 2
Dispatch ROM 1
Address select logic
Op
Instruction register
opcode field

Microprogramming: What is a “microinstruction”



Microprogramming

- **A specification methodology**
  - **appropriate if hundreds of opcodes, modes, cycles, etc.**
  - **signals specified symbolically using microinstructions**

Label	**ALU control**	**SRC1**	**SRC2**	**Register control**	**Memory**	**PCWrite control**	**Sequencing**
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

- ***Will two implementations of the same architecture have the same microcode?***
- ***What would a micro-assembler do?***

Microinstruction format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

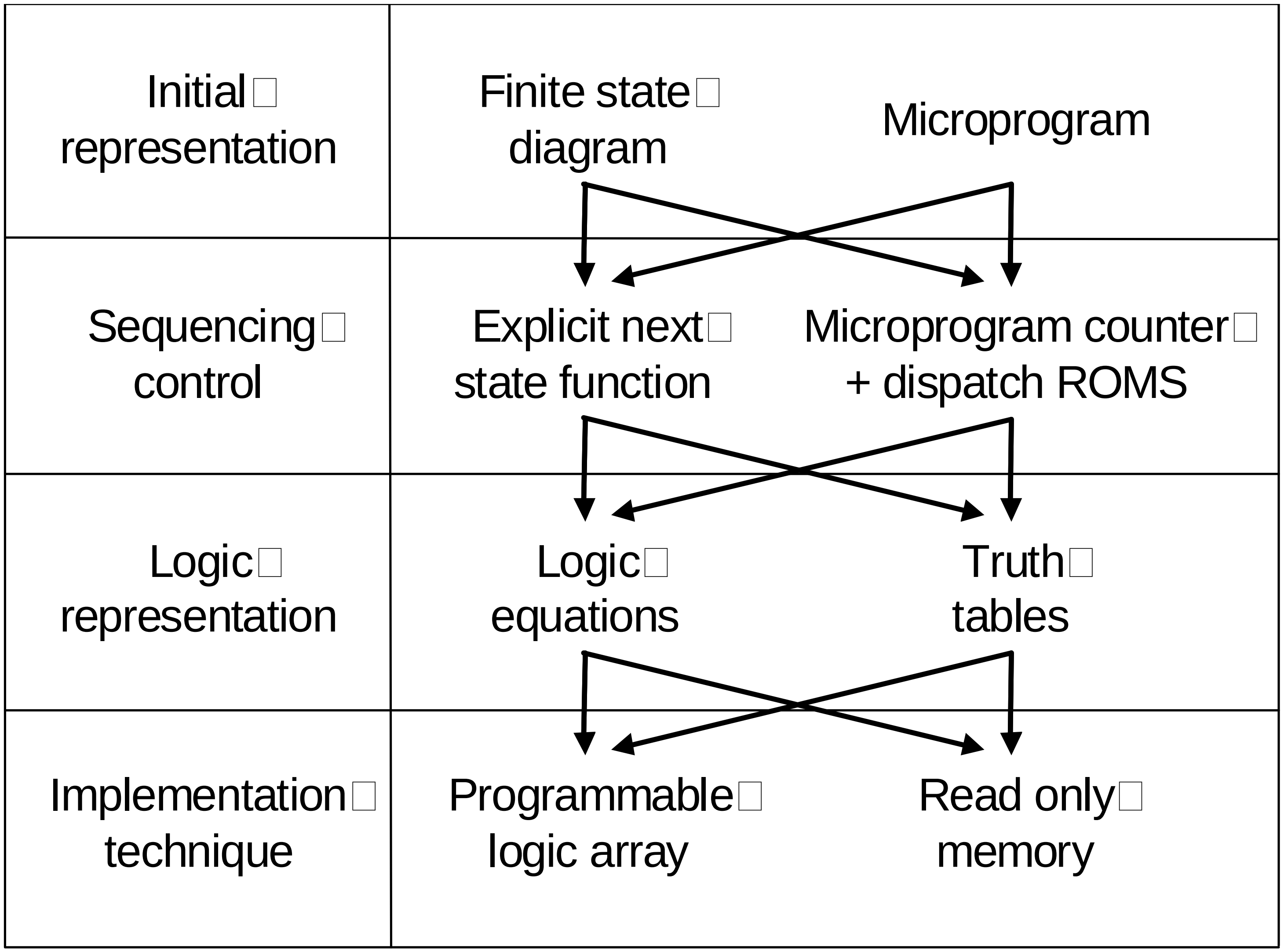
Maximally vs. Minimally Encoded

- No encoding:
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!
- Lots of encoding:
  - send the microinstructions through logic to get control signals
  - uses less memory, slower
- Historical context of CISC:
  - Too much logic to put on a single chip with everything else
  - Use a ROM (or even RAM) to hold the microcode
  - It’s easy to add new instructions

Microcode: Trade-offs

- **Distinction between specification and implementation is blurred**
- **Specification Advantages:**
  - **Easy to design and write**
  - **Design architecture and microcode in parallel**
- **Implementation (off-chip ROM) Advantages**
  - **Easy to change since values are in memory**
  - **Can emulate other architectures**
  - **Can make use of internal registers**
- **Implementation Disadvantages, SLOWER now that:**
  - **Control is implemented on same chip as processor**
  - **ROM is no longer faster than RAM**
  - **No need to go back and make changes**

The Big Picture



Initial representation	Finite state diagram	Microprogram
Sequencing control	Explicit next state function	Microprogram counter + dispatch ROMS
Logic representation	Logic equations	Truth tables
Implementation technique	Programmable logic array	Read only memory

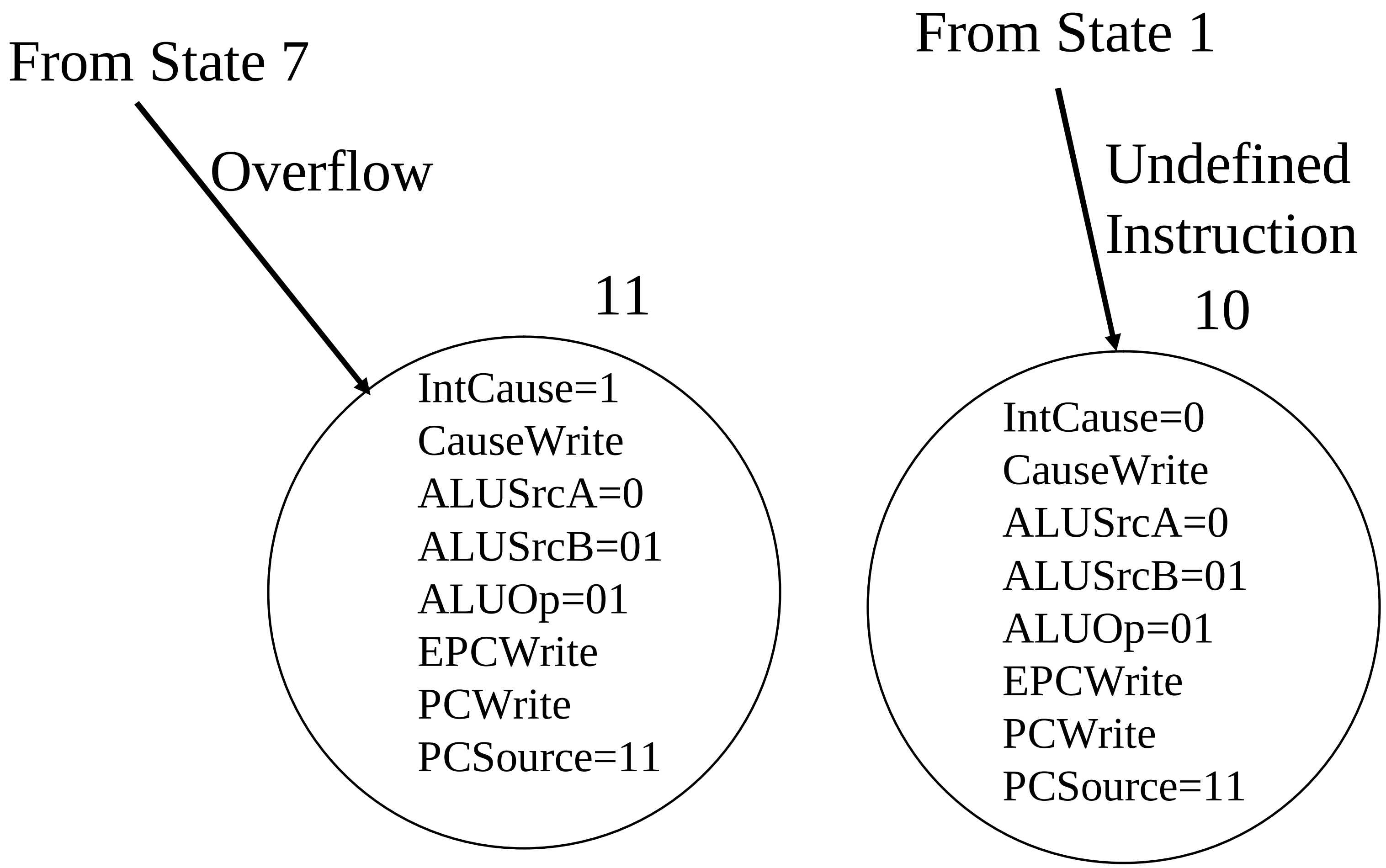
Exceptions

- What should the machine do if there is a problem
- Exceptions are just that
  - Changes in the normal execution of a program
- Two types of exceptions
  - External Condition: I/O interrupt, power failure, user termination signal (Ctrl-C)
  - Internal Condition: Bad memory read address (not a multiple of 4), illegal instructions, overflow/underflow.
- Interrupts – external
- Exceptions – internal
- Usually we refer to both by the general term “Exception”
- In either case, we need some mechanism by which we can handle the exception generated.
- Control is transferred to an exception handling mechanism, stored at a pre-specified location
- Address of instruction is saved in a register called EPC

How Exceptions are Handled

- We need two special registers
  - EPC: 32 bit register to hold address of current instruction
  - Cause: 32 bit register to hold information about the type of exception that has occurred.
- Simple Exception Types
  - Undefined Instruction
  - Arithmetic Overflow
- Another type is Vectored Interrupts
  - Do not need cause register
  - Appropriate exception handler jumped to from a vector table

Two new states for the Multi-cycle CPU



Vectored Interrupts/Exceptions

- Address of exception handler depends on the problem
  - Undefined Instruction C0 00 00 00
  - Arithmetic OverflowC0 00 00 20
  - Addresses are separated by a fixed amount, 32 bytes in MIPS
- PC is transferred to a register called EPC
- If interrupts are not vectored, then we need another register to store the cause of problem
- In what state what exception can occur?

Final Words on Single and Multi-Cycle Systems

- Single cycle implementation
  - Simpler but slowest
  - Require more hardware
- Multi-cycle
  - Faster clock
  - Amount of time it takes depends on instruction mix
  - Control more complicated
- Exceptions and Other conditions add a lot of complexity
- Other techniques to make it faster

Conclusions on Chapter 5

- **Control is the most complex part**
- **Can be hard-wired, ROM-based, or micro-programmed**
- **Simpler instructions also lead to simple control**
- **Just because machine is micro-programmed, we should not add complicated instructions**
- **Sometimes simple instructions are more effective than a single complex instruction**
- **More complex instructions may have to be maintained for compatibility reasons**