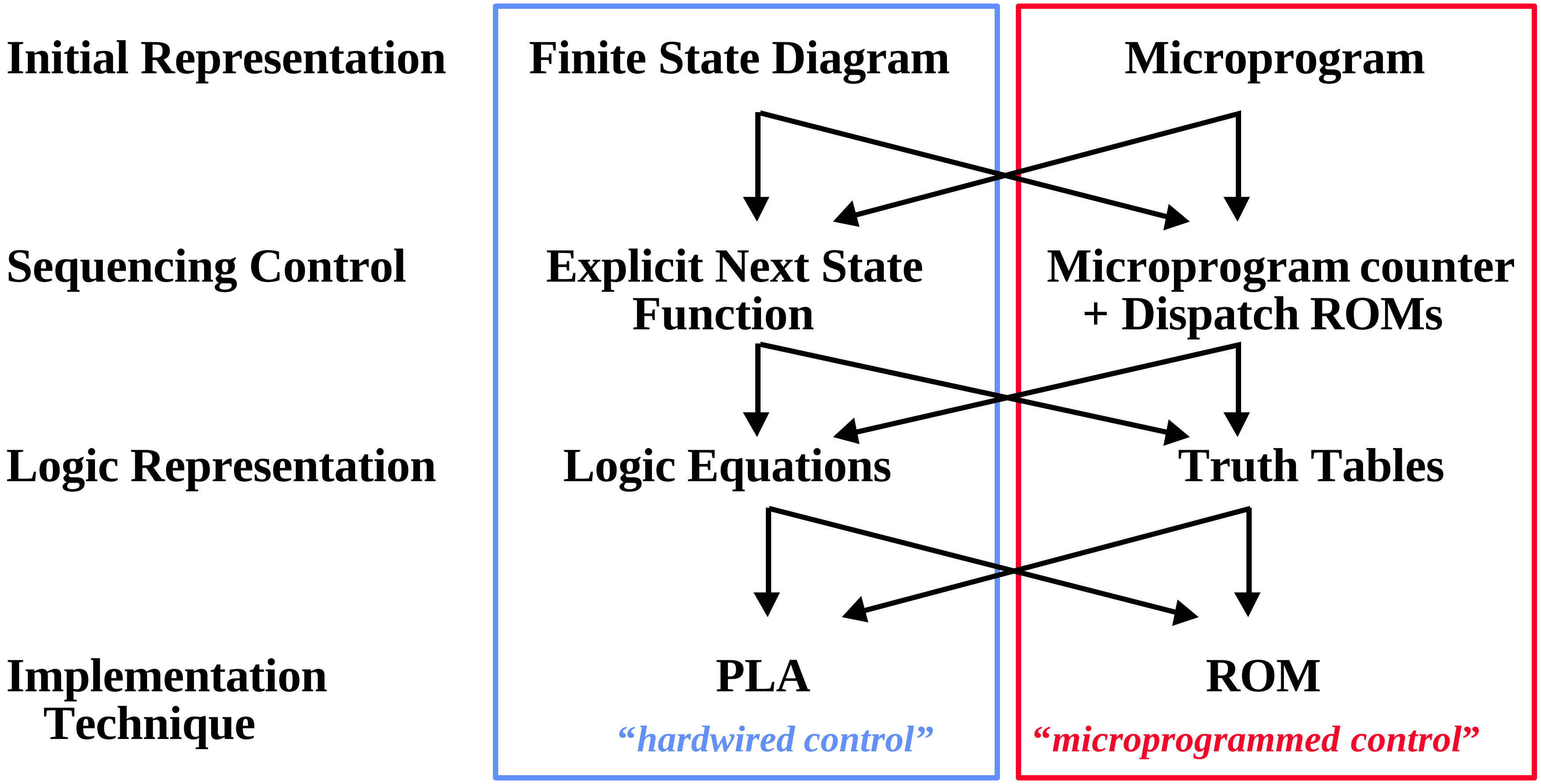
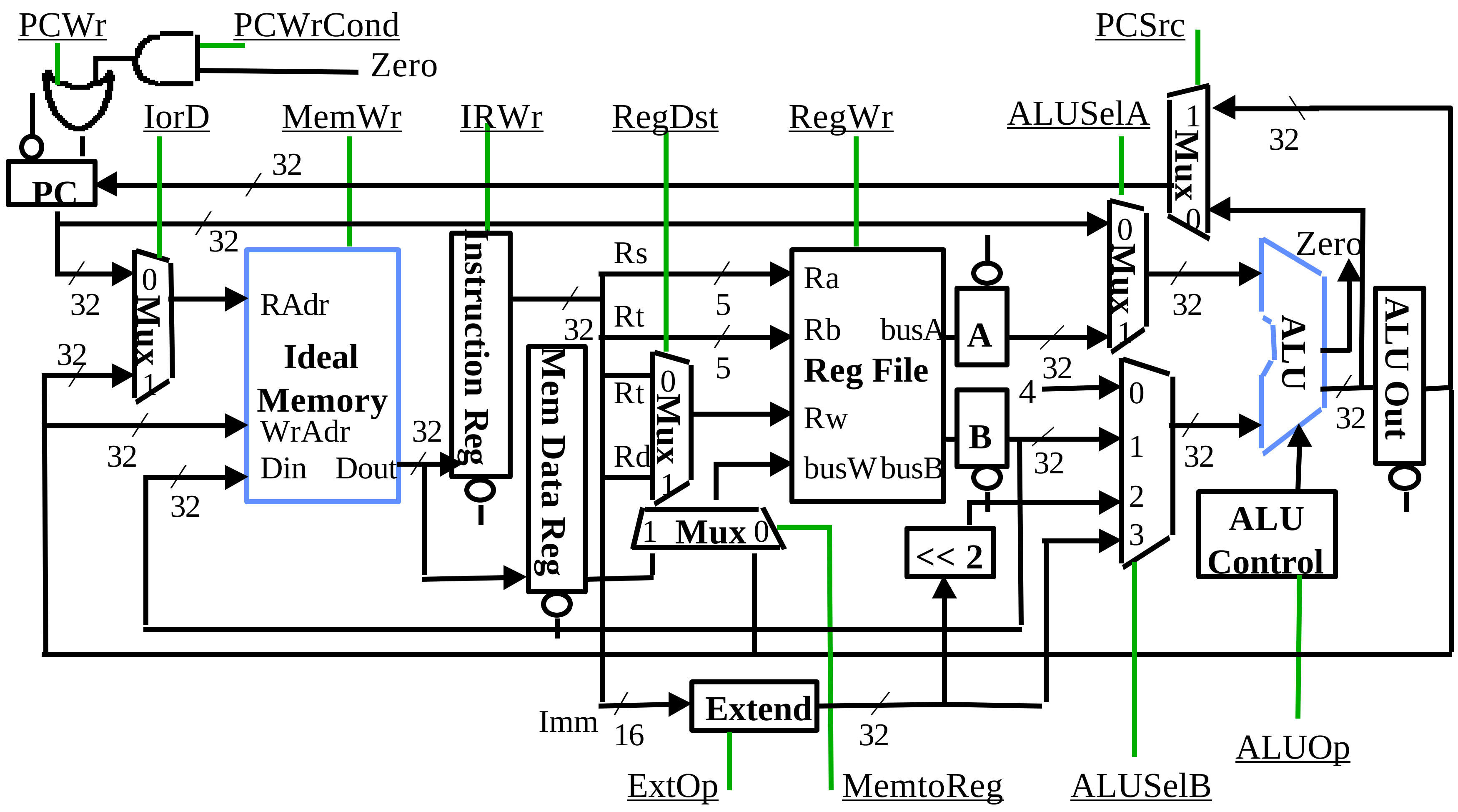
Control Implementation Alternatives

- Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.



EECC550 - Shaaban

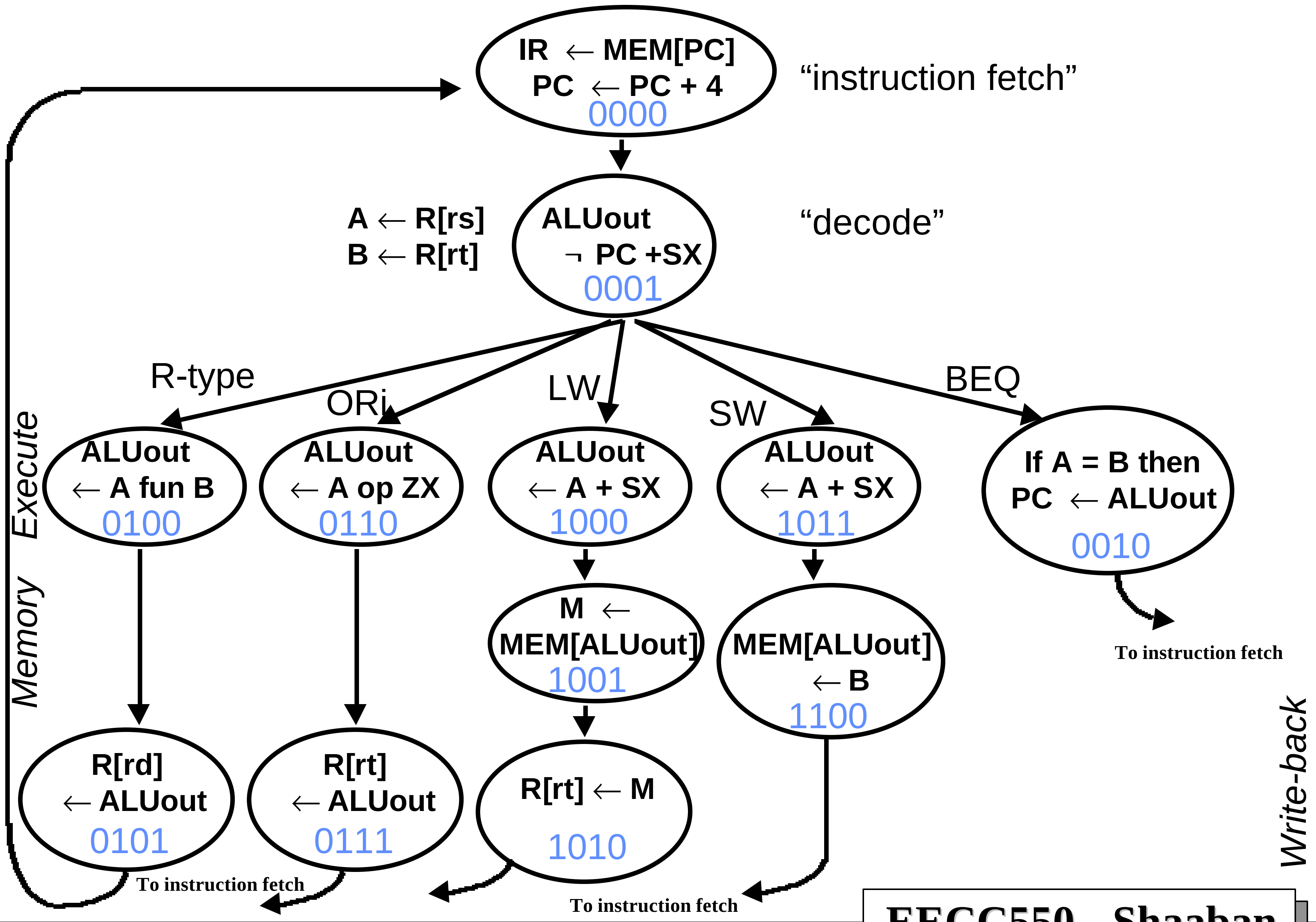
Alternative datapath (Textbook): Multiple Cycle Datapath



Operations In Each Cycle

	R-Type	Logic Immediate	Load	Store	Branch
Instruction Fetch	IR ¬ Mem[PC] PC ¬ PC + 4	IR ¬ Mem[PC] PC ¬ PC + 4	IR ¬ Mem[PC] PC ¬ PC + 4	IR ¬ Mem[PC] PC ¬ PC + 4	IR ¬ Mem[PC] PC ¬ PC + 4
Instruction Decode	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)	A ¬ R[rs] B ¬ R[rt] ALUout ¬ PC + (SignExt(imm16) x4)
Execution	ALUout ¬ A + B	ALUout ¬ A OR ZeroExt[imm16]	ALUout ¬ A + SignEx(Im16)	ALUout ¬ A + SignEx(Im16)	If Equal = 1 PC ¬ ALUout
Memory			M ¬ Mem[ALUout]	Mem[ALUout] ¬ B	
Write Back	R[rd] ¬ ALUout	R[rt] ¬ ALUout	R[rt] ¬ Mem		

Finite State Machine (FSM) Specification

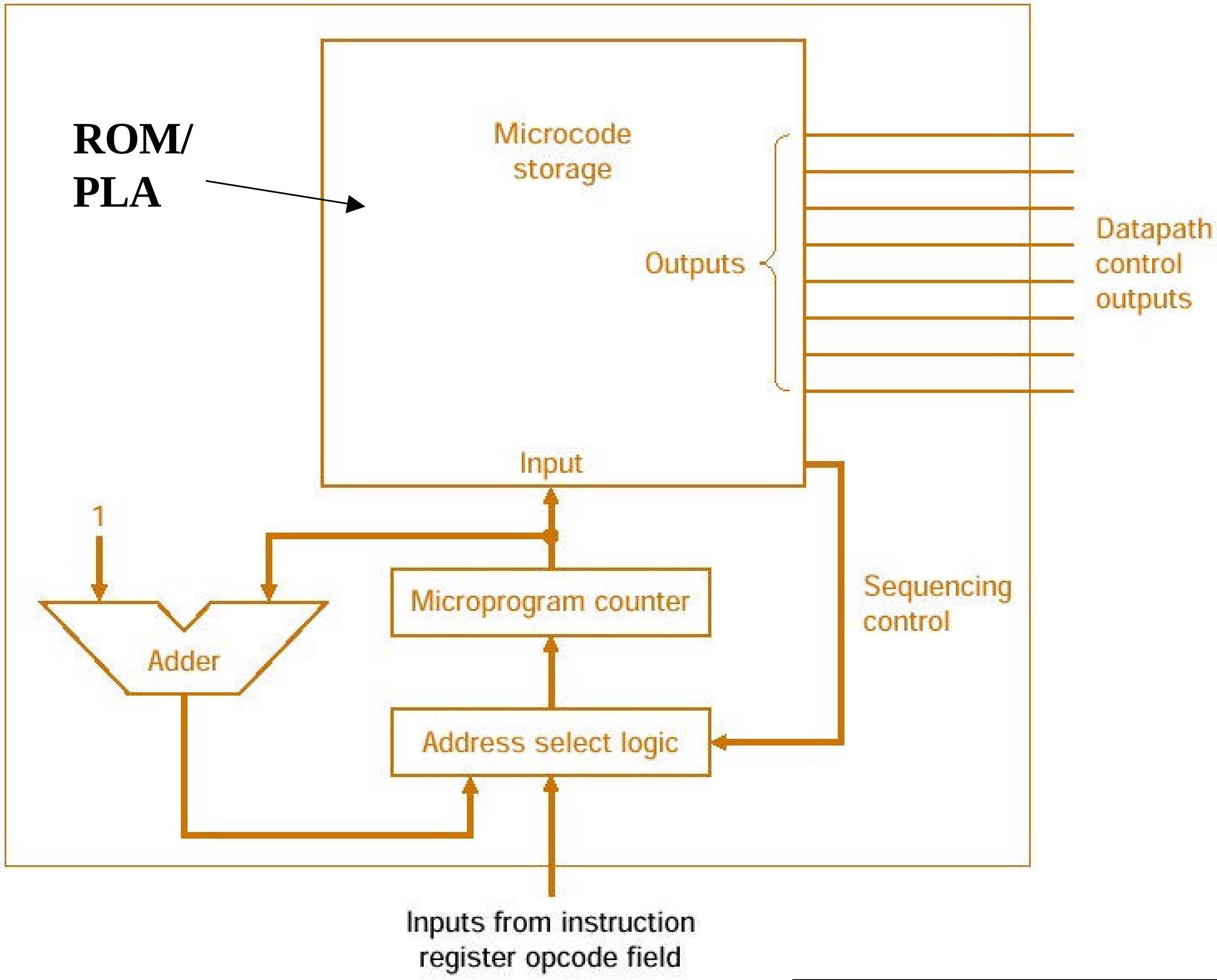


EECC550 - Shaaban

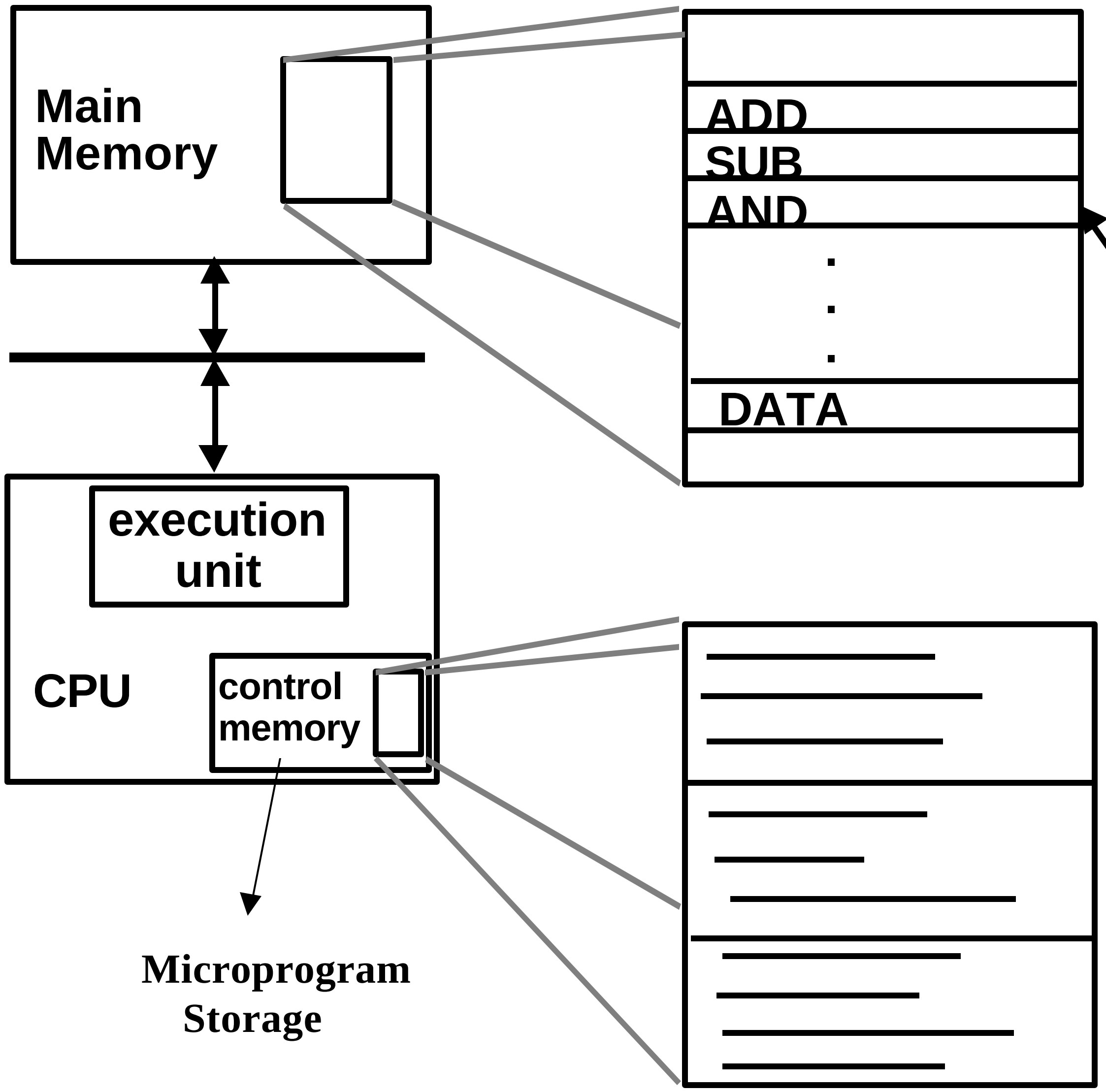
Microprogrammed Control

- Finite state machine control for a full set of instructions is very complex, and may involve a very large number of states:
  - Slight microoperation changes require new FSM controller.
- Microprogramming: Designing the control as a program that implements the machine instructions.
- A microprogam for a given machine instruction is a symbolic representation of the control involved in executing the instruction and is comprised of a sequence of microinstructions.
- Each microinstruction defines the set of datapath control signals that must asserted (active) in a given state or cycle.
- The format of the microinstructions is defined by a number of fields each responsible for asserting a set of control signals.
- Microarchitecture:
  - Logical structure and functional capabilities of the hardware as seen by the microprogrammer.

A Typical Microcode Controller Implementation



"Macroinstruction" Interpretation



User program plus Data

one of these is mapped into one of these

AND microsequence

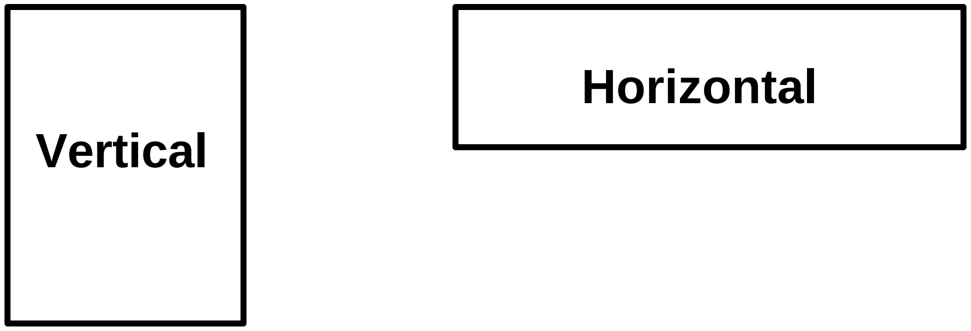
e.g., Fetch
Calc Operand Addr
Fetch Operand(s)
Calculate
Save Answer(s)

Variations on Microprogram Formats

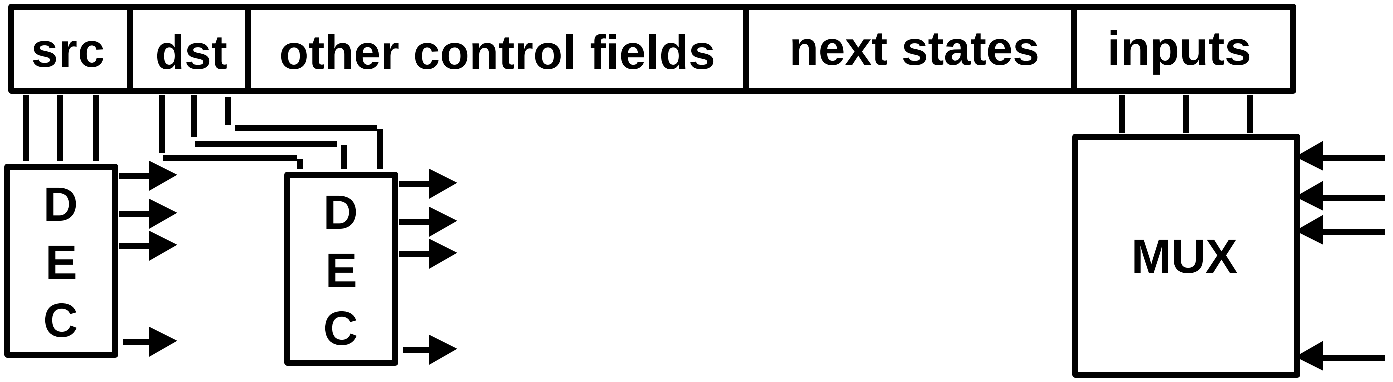
- "Horizontal" Microcode:
  - A control field for each control point in the machine.

µseq	µaddr	A-mux	B-mux	bus enables	register enables	

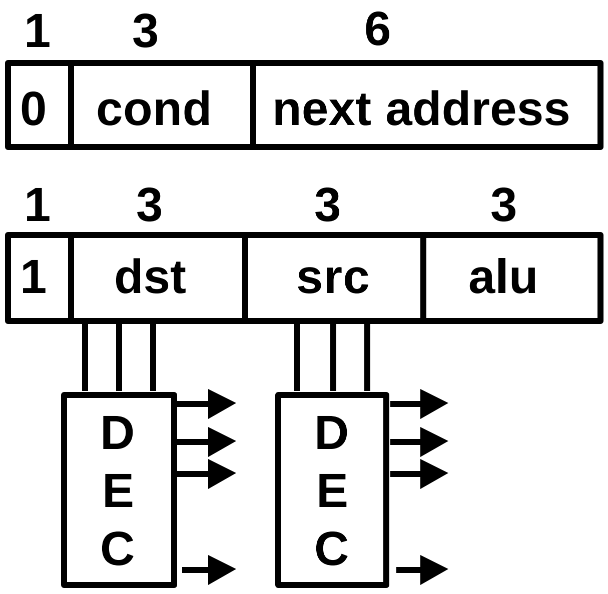
- "Vertical" Microcode:
  - A Compact microinstruction format for each class of control points.
  - Local decode is used to generate all control points.



More Vertical Microprogram Formats



Multiformat Microcode:



Branch Jump

Register Transfer Operation

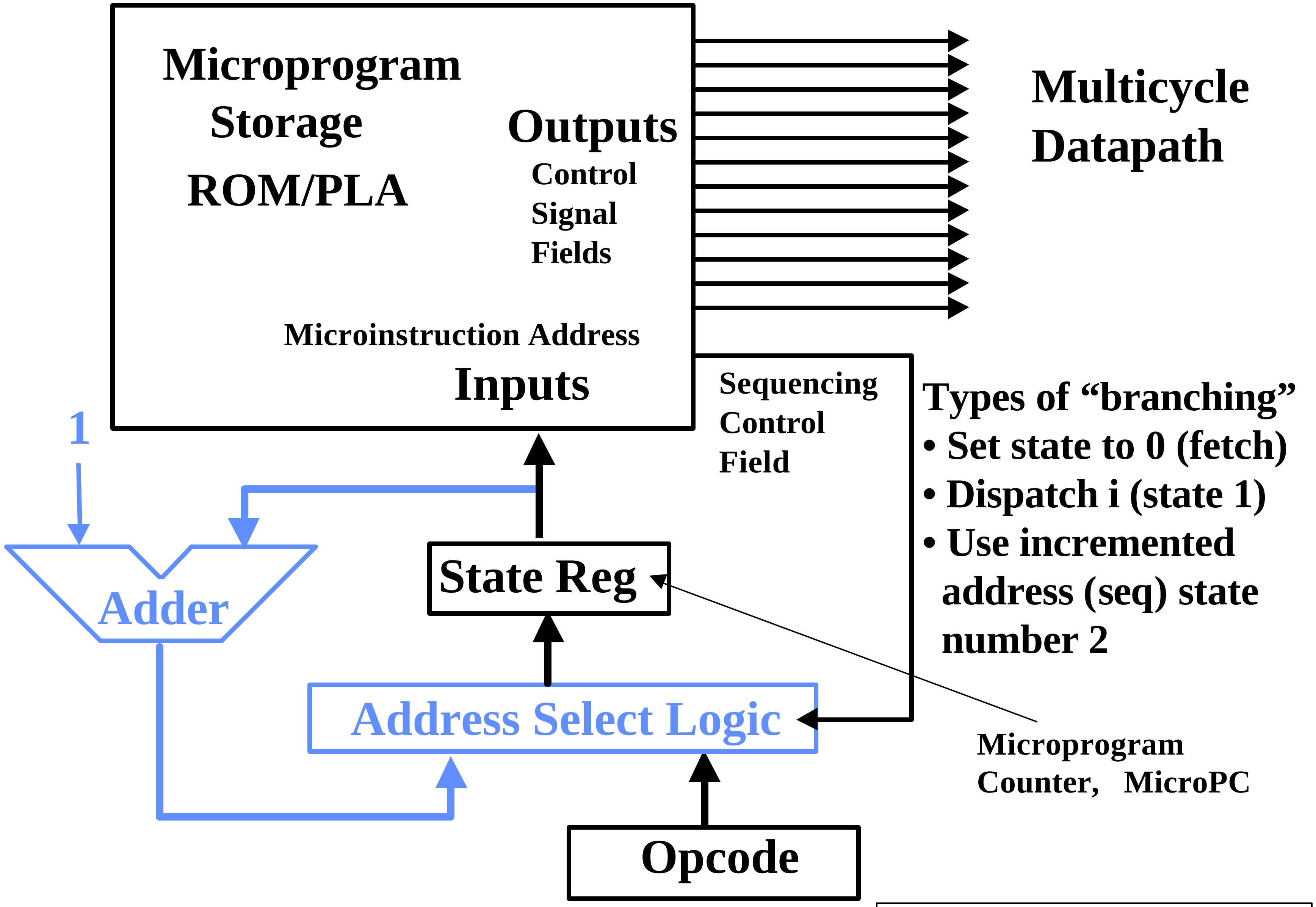
Microinstruction Format/Addressing

- Start with list of all control signals.
- Partition control signals with similar functions into a number of signal sets that share a single microinstruction field.
- A sequencing microinstruction field is used to indicate the next microinstruction to execute.
- Places fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last).
- Since microinstructions are placed in a ROM or PLA, addresses must be assigned to microinstructions, usually sequentially.
- Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals.
- To minimize microinstruction width, operations that will never be used at the same time may be encoded.

Next Microinstruction Selection

- **The next microinstruction to execute can be found by using the sequencing field:**
  - **Branch to a microinstruction that begins execution of the next MIPS instruction. "Fetch" is placed in the sequencing field.**
  - **Increment the address of the current instruction. Indicated in the microinstruction by putting "Seq" in the sequencing field.**
  - **Choose the next microinstruction based on the control unit input (a dispatch).**
    - **Dispatches are implemented by a look-up table stored in a ROM containing addresses of target microinstruction.**
    - **The table is indexed by the control unit input.**
    - **A dispatch operation is indicated by placing "Dispatch i" in the sequencing field; i is the dispatch table number.**

Microprogrammed Control Unit

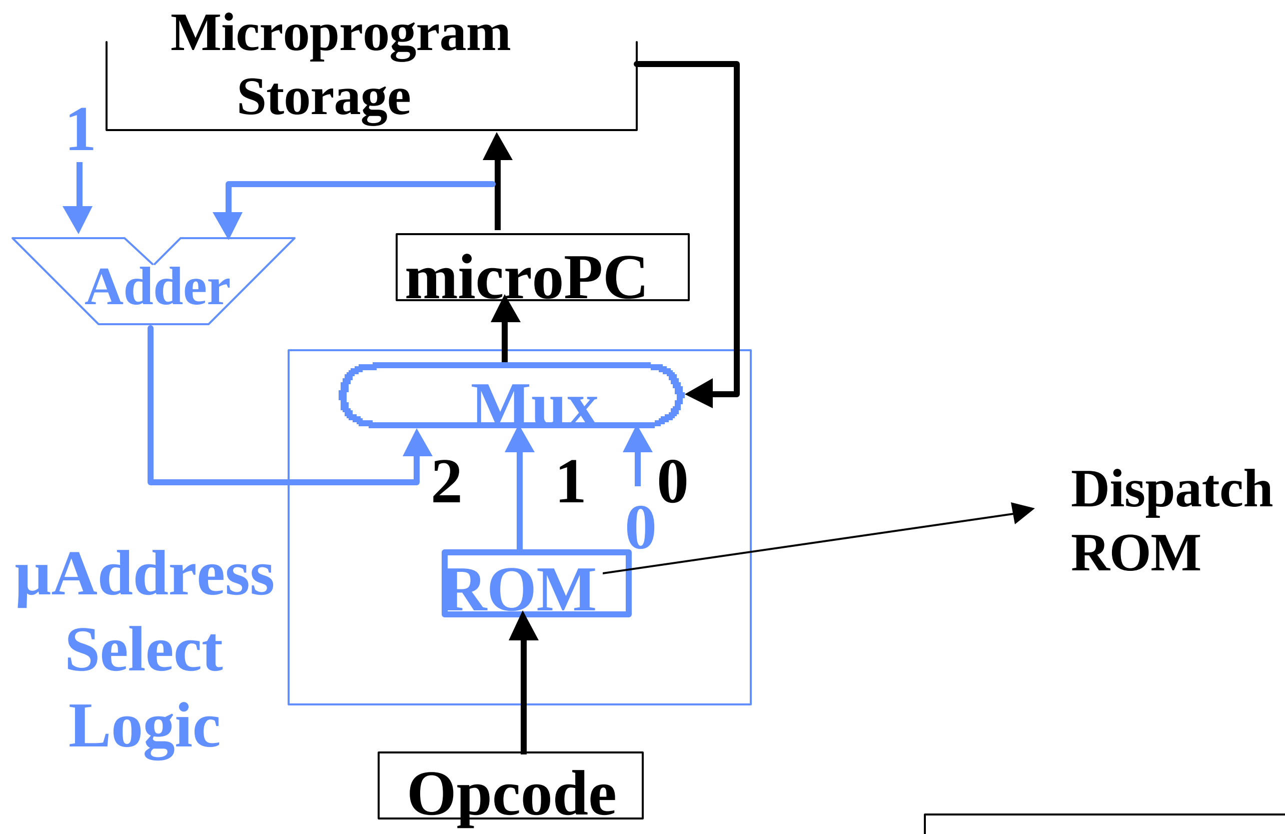


Types of "branching"
- Set state to 0 (fetch)
- Dispatch i (state 1)
- Use incremented address (seq) state number 2

Next State Function: Sequencing Field

- For next state function (next microinstruction address):

Signal	*Name*	*Value*	*Effect*
Sequencing	Fetch	00	Next µaddress = 0
	Dispatch i	01	Next µaddress = dispatch ROM
	Seq	10	Next µaddress = µaddress + 1



List of control Signals Grouped Into Fields

Single Bit Control

Signal name	*Effect when deasserted*	*Effect when asserted*
ALUSelA	1st ALU operand = PC	1st ALU operand = Reg[rs]
RegWrite	None	Reg. is written
MemtoReg	Reg. write data input = ALU	Reg. write data input = memory
RegDst	Reg. dest. no. = rt	Reg. dest. no. = rd
MemRead	None	Memory at address is read, MDR ¬ Mem[addr]
MemWrite	None	Memory at address is written
IorD	Memory address = PC	Memory address = S
IRWrite	None	IR ¬ Memory
PCWrite	None	PC ¬ PCSource
PCWriteCond	None	IF ALUzero then PC ¬ PCSource
PCSource	PCSource = ALU	PCSource = ALUout

Multiple Bit Control

Signal name	*Value*	*Effect*
ALUOp	00	ALU adds
	01	ALU subtracts
	10	ALU does function code
	11	ALU does logical OR
ALUSelB	000	2nd ALU input = Reg[rt]
	001	2nd ALU input = 4
	010	2nd ALU input = sign extended IR[15-0]
	011	2nd ALU input = sign extended, shift left 2 IR[15-0]
	100	2nd ALU input = zero extended IR[15-0]

Microinstruction Format

Field Name	***Width***		***Control Signals Set***
	wide	narrow	
ALU Control	4	2	ALUOp
SRC1	2	1	ALUSelA
SRC2	5	3	ALUSelB
Destination	3	2	RegWrite, MemtoReg, RegDst
Memory	4	3	MemRead, MemWrite, IorD
Memory Register	1	1	IRWrite
PCWrite Control	4	3	PCWrite, PCWriteCond, PCSource
Sequencing	3	2	AddrCtl
Total width	26	17	bits

Microinstruction Field Values

Field Name	***Values for Field***	***Function of Field with Specific Value***
ALU	**Add**	**ALU adds**
	Subt.	**ALU subtracts**
	Func code	**ALU does function code**
	Or	**ALU does logical OR**
SRC1	**PC**	**1st ALU input = PC**
	rs	**1st ALU input = Reg[rs]**
SRC2	**4**	**2nd ALU input = 4**
	Extend	**2nd ALU input = sign ext. IR[15-0]**
	Extend0	**2nd ALU input = zero ext. IR[15-0]**
	Extshft	**2nd ALU input = sign ex., sl IR[15-0]**
	rt	**2nd ALU input = Reg[rt]**
destination	**rd ALU**	**Reg[rd] ¬ ALUout**
	rt ALU	**Reg[rt] ¬ ALUout**
	rt Mem	**Reg[rt] ¬ Mem**
Memory	**Read PC**	**Read memory using PC**
	Read ALU	**Read memory using ALU output**
	Write ALU	**Write memory using ALU output, value B**
Memory register	**IR**	**IR ¬ Mem**
PC write	**ALU**	**PC ¬ ALU**
	ALUoutCond	**IF ALU Zero then PC ¬ ALUout**
Sequencing	**Seq**	**Go to sequential micoinstruction**
	Fetch	**Go to the first microinstruction**
	Dispatch i	**Dispatch using ROM.**

Instruction Fetch/decode Microcode Sequence

Label	ALU	SRC1	SRC2	Dest.	Memory	Mem. Reg.	PC Write	Sequencing
Fetch:	Add	PC	4		Read PC	IR	ALU	Seq
	Add	PC	Extshft					Dispatch

First microinstruction: Fetch, increment PC

Field Name	Value for Field	Function of Field
ALU	Add	ALU adds
SRC1	PC	1st ALU input = PC
SRC2	4	2nd ALU input = 4
Memory	Read PC	Read memory using PC
Memory register	IR	IR ¬ Mem
PC write	ALU	PC ¬ ALU
Sequencing	Seq	Go to sequential µinstruction

Second microinstruction: Decode, calculate branch address

Field Name	Value for Field	Function of Field
ALU	Add	ALU adds result in ALUout
SRC1	PC	1st ALU input = PC
SRC2	Extshft	2nd ALU input = sign ex., sl IR[15-0]
Sequencing	Dispatch	Dispatch using ROM according to opcode

LW Completion Microcode Sequence

Label	ALU	SRC1	SRC2	Dest.	Memory	Mem. Reg.	PC Write	Sequencing
Lw:	Add	rs	Extend					Seq
					Read ALU			Seq
				rt MEM				Fetch

First microinstruction: Execute, effective memory address calculation

Field Name	Value for Field	Function of Field
ALU	Add	ALU adds, result in ALUout
SRC1	rs	1st ALU input = Reg[rs]
SRC2	Extend	2nd ALU input = sign ext. IR[15-0]
Sequencing	Seq	Go to sequential µinstruction

Second microinstruction: Memory, read using ALUout

Field Name	Values for Field	Function of Field
Memory	Read ALU	Read memory using ALU output
Sequencing	Seq	Go to sequential µinstruction

Third microinstruction: Write Back, from memory to register rt

Field Name	Values for Field	Function of Field
destination	rt Mem	Reg[rt] $\leftarrow$ Mem
Sequencing	Fetch	Go to the first microinstruction (fetch)

SW Completion Microcode Sequence

Label	ALU	SRC1	SRC2	Dest.	Memory	Mem. Reg.	PC Write	Sequencing
Sw:	Add	rs	Extend					Seq
					Write ALU			Fetch

First microinstruction: Execute, effective memory address calculation

Field Name	Value for Field	Function of Field
ALU	Add	ALU adds result in ALUout
SRC1	rs	1st ALU input = Reg[rs]
SRC2	Extend	2nd ALU input = sign ext. IR[15-0]
Sequencing	Seq	Go to sequential µinstruction

Second microinstruction: Memory, write to memory

Field Name	Values for Field	Function of Field
Memory	Write ALU	Write memory using ALU output, value B
Sequencing	Fetch	Go to the first microinstruction (fetch)

R-Type Completion Microcode Sequence

Label	*ALU*	*SRC1*	*SRC2*	*Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequencing*
Rtype:	**Func**	**rs**	**rt**					**Seq**
				rd ALU				**Fetch**

First microinstruction: Execute, perform ALU function

Field Name	*Value for Field*	*Function of Field*
ALU	Func code	ALU does function code
SRC1	rs	1st ALU input = Reg[rs]
SRC2	rt	2nd ALU input = Reg[rt]
Sequencing	Seq	Go to sequential µinstruction

Second microinstruction: Write Back, ALU result in register rd

Field Name	*Values for Field*	*Function of Field*
destination	rd ALU	Reg[rd] ¬ ALUout
Sequencing	Fetch	Go to the first microinstruction (fetch)

BEQ Completion Microcode Sequence

Label	*ALU*	*SRC1*	*SRC2*	*Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequencing*
Beq:	**Subt.**	**rs**	**rt**				**ALUoutCond.**	**Fetch**

First microinstruction: Execute, compute condition, update PC

Field Name	***Values for Field***	***Function of Field***
ALU	**Subt.**	**ALU subtracts**
SRC1	**rs**	**1st ALU input = Reg[rs]**
SRC2	**rt**	**2nd ALU input = Reg[rt]**
PC write	**ALUoutCond**	**IF ALU Zero then PC ¬ ALUout**
Sequencing	**Fetch**	**Go to the first microinstruction (fetch)**

ORI Completion Microcode Sequence

Label	*ALU*	*SRC1*	*SRC2*	*Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequencing*
Ori:	**Or**	**rs**	**Extend0**					**Seq**
				rt ALU				**Fetch**

First microinstruction: Execute, rs OR immediate

Field Name	*Value for Field*	*Function of Field*
ALU	Or	ALU does logical OR result in ALUout
SRC1	rs	1st ALU input = Reg[rs]
SRC2	Extend0	2nd ALU input = zero ext. IR[15-0]
Sequencing	Seq	Go to sequential µinstruction

Second microinstruction: Write Back, ALU result in register rt

Field Name	*Values for Field*	*Function of Field*
destination	rt ALU	Reg[rt] ¬ ALUout
Sequencing	Fetch	Go to the first microinstruction (fetch)

Microprogram for The Control Unit

Label	*ALU*	*SRC1*	*SRC2*	*Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequencing*
Fetch:	Add	PC	4		Read PC	IR	ALU	Seq
	Add	PC	Extshft					Dispatch
Lw:	Add	rs	Extend					Seq
					Read ALU			Seq
				rt MEM				Fetch
Sw:	Add	rs	Extend					Seq
					Write ALU			Fetch
Rtype:	Func	rs	rt					Seq
				rd ALU				Fetch
Beq:	Subt.	rs	rt				ALUoutCond.	Fetch
Ori:	Or	rs	Extend0					Seq
				rt ALU				Fetch

Microprogramming Pros and Cons

- Ease of design.
- Flexibility:
  - Easy to adapt to changes in organization, timing, technology.
  - Can make changes late in design cycle, or even in the field.
- Can implement very powerful instruction sets (just more microprogram control memory is needed).
- Generality:
  - Can implement multiple instruction sets on the same machine.
  - Can tailor instruction set to application.
- Compatibility:
  - Many organizations, same instruction set.
- Possibly more costly to implement than FSM control.
- Usually slower than FSM control.

Exceptions Handling in MIPS

- **Exceptions: Events Other than branches or jumps that change the normal flow of instruction execution.**
- **Two main types: Interrupts, Traps.**
  - **An interrupt usually comes from outside the processor (I/O devices) to get the CPU's attention to start a service routine.**
  - **A trap usually originates from an event within the CPU (Arithmetic overflow, undefined instruction) and initiates an exception handling routine usually by the operating system.**
- **The current MIPS implementation being considered can be extended to handle exceptions by adding two additional registers and the associated control lines:**
  - **EPC: A 32 bit register to hold the address of the affected instruction**
  - **Cause: A register used to record the cause of the exception.**

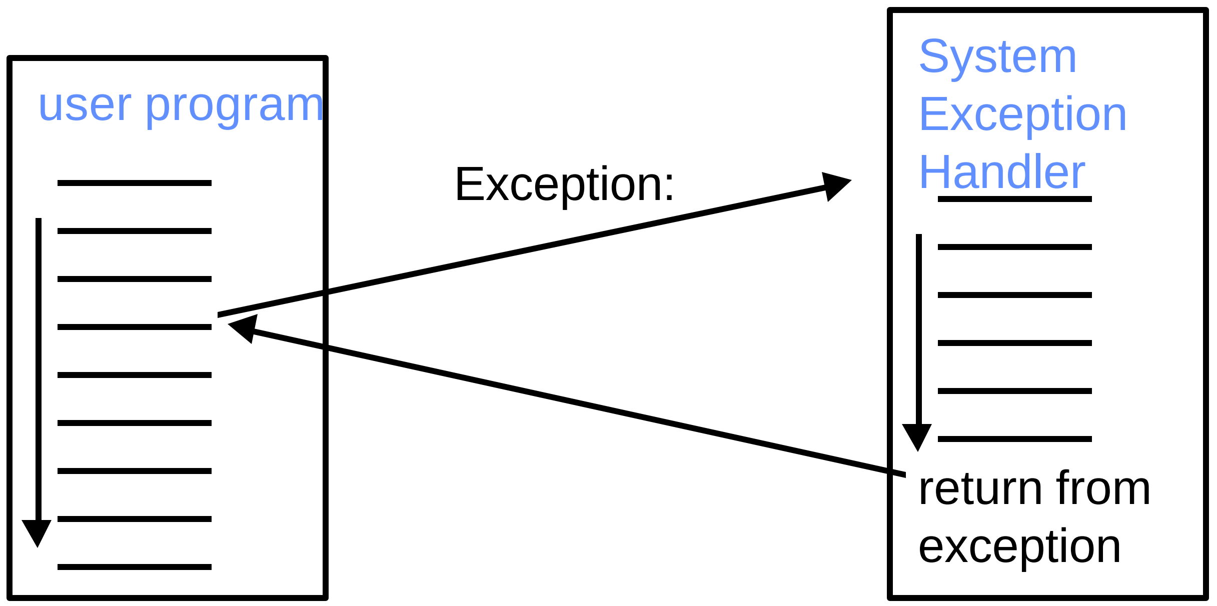
  In this implementation only the low-order bit is used to encode the two handled exceptions: undefined instruction = 0

  overflow = 1
- **Two additional states are added to the control finite state machine to handle these exceptions.**

Two Types of Exceptions

- **Interrupts:**
  - **Caused by external events (e.g. I/O device requests).**
  - **Asynchronous to program execution.**
  - **May be handled between instructions.**
  - **Simply suspend and resume user program.**
- **Traps:**
  - **Caused by internal events:**
    - **Exceptional conditions (e.g. overflow).**
    - **Errors (e.g memory parity error).**
    - **Faults (e.g. Page fault, non-resident page).**
  - **Synchronous to program execution.**
  - **Condition must be remedied by the system exception handler.**
  - **Instruction may be executed again and program continued or program may be aborted.**

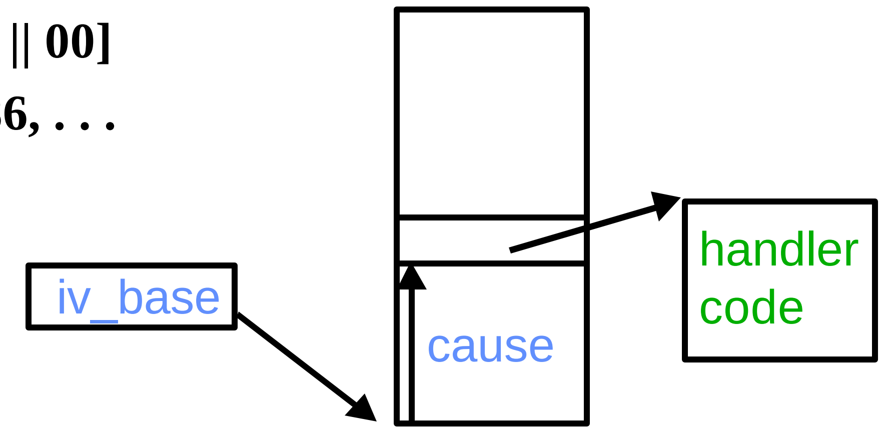
Exception Handling

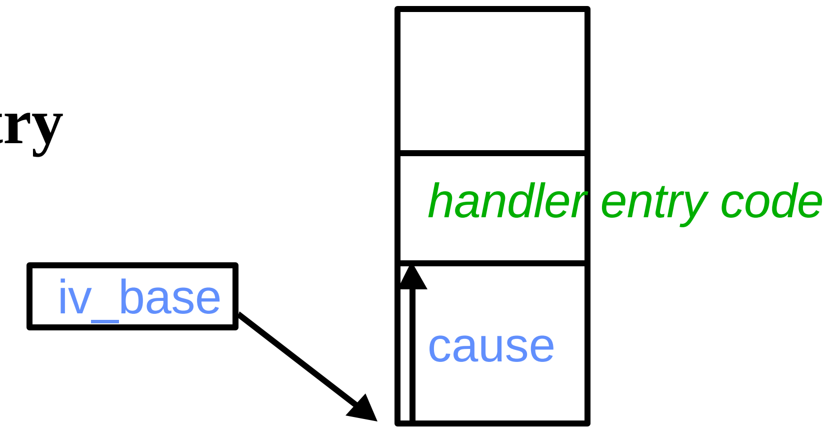


- **Exception = an unprogrammed control transfer**
  - **System takes action to handle the exception which include:**
    - **Recording the address of the offending instruction.**
    - **Saving & restoring user program state.**
    - **Returning control to user (unless user program is aborted).**

Addressing The Exception Handler

- **Traditional Approach, Interrupt Vector:**
  - **PC ¬ MEM[IV_base + cause || 00]**
  - **Used in: 370, 68000, Vax, 80x86, . . .**
- **RISC Handler Table:**
  - **PC ¬ IT_base + cause || 0000**
  - **saves state and jumps**
  - **Used in: Sparc, HP-PA, . . .**
- **MIPS Approach: Fixed entry**
  - **PC ¬ EXC_addr**
  - **Actually a very small table:**
    - **RESET entry**
    - **TLB**
    - **other**





Exception Handling: Saving The State

- Push it onto the stack:
  - Vax, 68k, 80x86

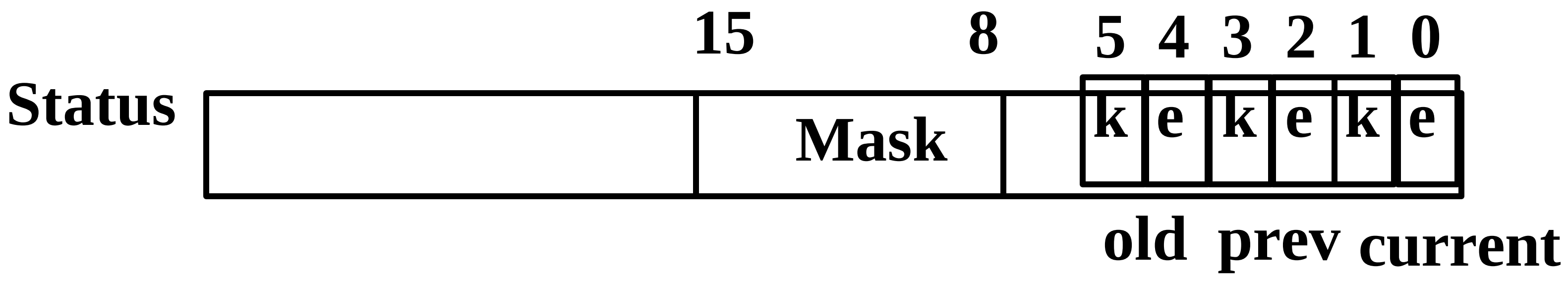
- Save it in special registers:
  - MIPS: EPC, BadVaddr, Status, Cause

- Shadow Registers:
  - M88k.
    - Save state in a shadow (a copy) of the internal CPU registers.

Additions to MIPS to Support Exceptions

- **EPC: A 32-bit register used to hold the address of the affected instruction (in reality register 14 of coprocessor 0).**
- **Cause: A register used to record the cause of the exception. In the MIPS architecture this register is 32 bits, though some bits are currently unused. Assume that bits 5 to 2 of this register encode the two possible exception sources mentioned above:**
  - **Undefined instruction = 0**
  - **Arithmetic overflow = 1 (in reality, register 13 of coprocessor 0).**
- **BadVAddr: Register contains memory address at which memory reference occurred (register 8 of coprocessor 0).**
- **Status: Interrupt mask and enable bits (register 12 of coprocessor 0).**
- **Control signals to write EPC , Cause, BadVAddr, and Status.**
- **Be able to write exception address into PC, increase mux to add as input 01000000 00000000 00000000 01000000_{two} (8000 0080_{hex}).**
- **May have to undo PC = PC + 4, since we want EPC to point to offending instruction (not its successor); PC = PC - 4**

Details of MIPS Status Register



- Mask = 1 bit for each of 5 hardware and 3 software interrupt levels

    1 → enables interrupts

    0 → disables interrupts

- k = kernel/user

    0 → was in the kernel when interrupt occurred

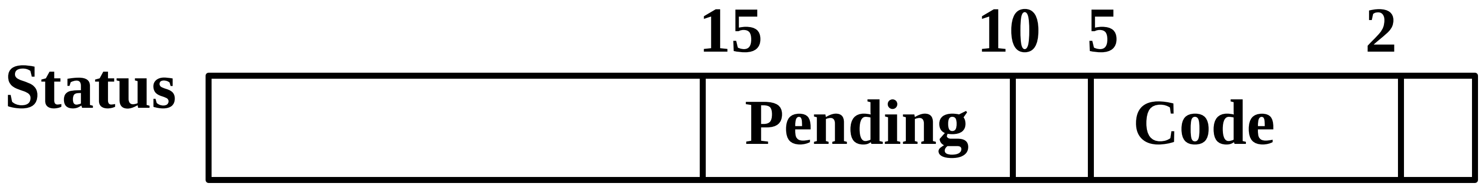
    1 → was running user mode

- e = interrupt enable

    0 → interrupts were disabled

    1 → interrupts were enabled

Details of MIPS Cause register



- **<u>Pending interrupt:</u> 5 hardware levels: bit set if interrupt occurs but not yet serviced:**
  - **Handles cases when more than one interrupt occurs at same time, or while records interrupt requests when interrupts disabled.**
- **<u>Exception Code:</u> Encodes reasons for interrupt:**

0 (INT) $\rightarrow$ external interrupt

4 (ADDRL) $\rightarrow$ Address error exception (load or instr fetch).

5 (ADDRS) $\rightarrow$ Address error exception (store).

6 (IBUS) $\rightarrow$ Bus error on instruction fetch.

7 (DBUS) $\rightarrow$ Bus error on data fetch.

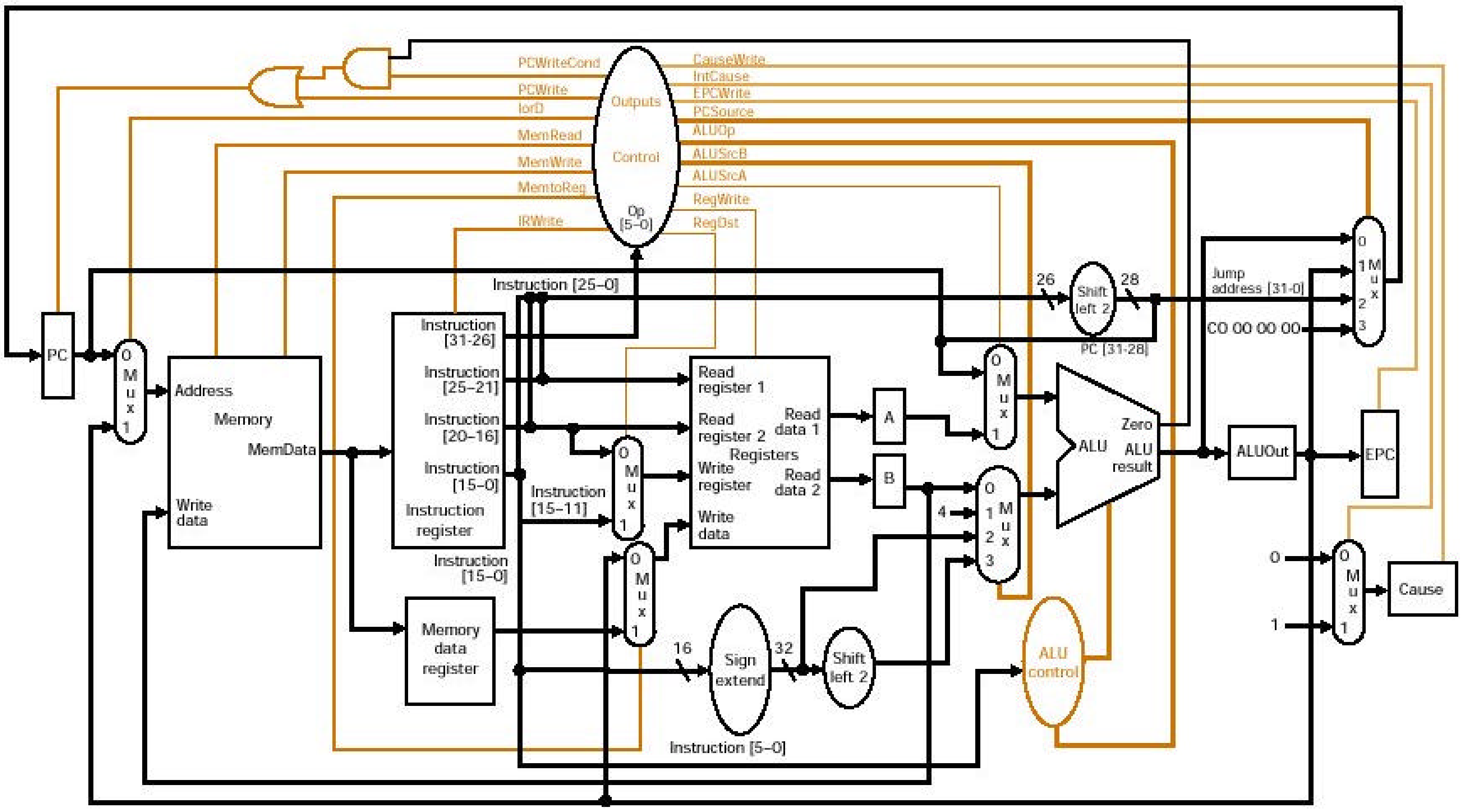
8 (Syscall) $\rightarrow$ Syscall exception.

9 (BKPT) $\rightarrow$ Breakpoint exception.

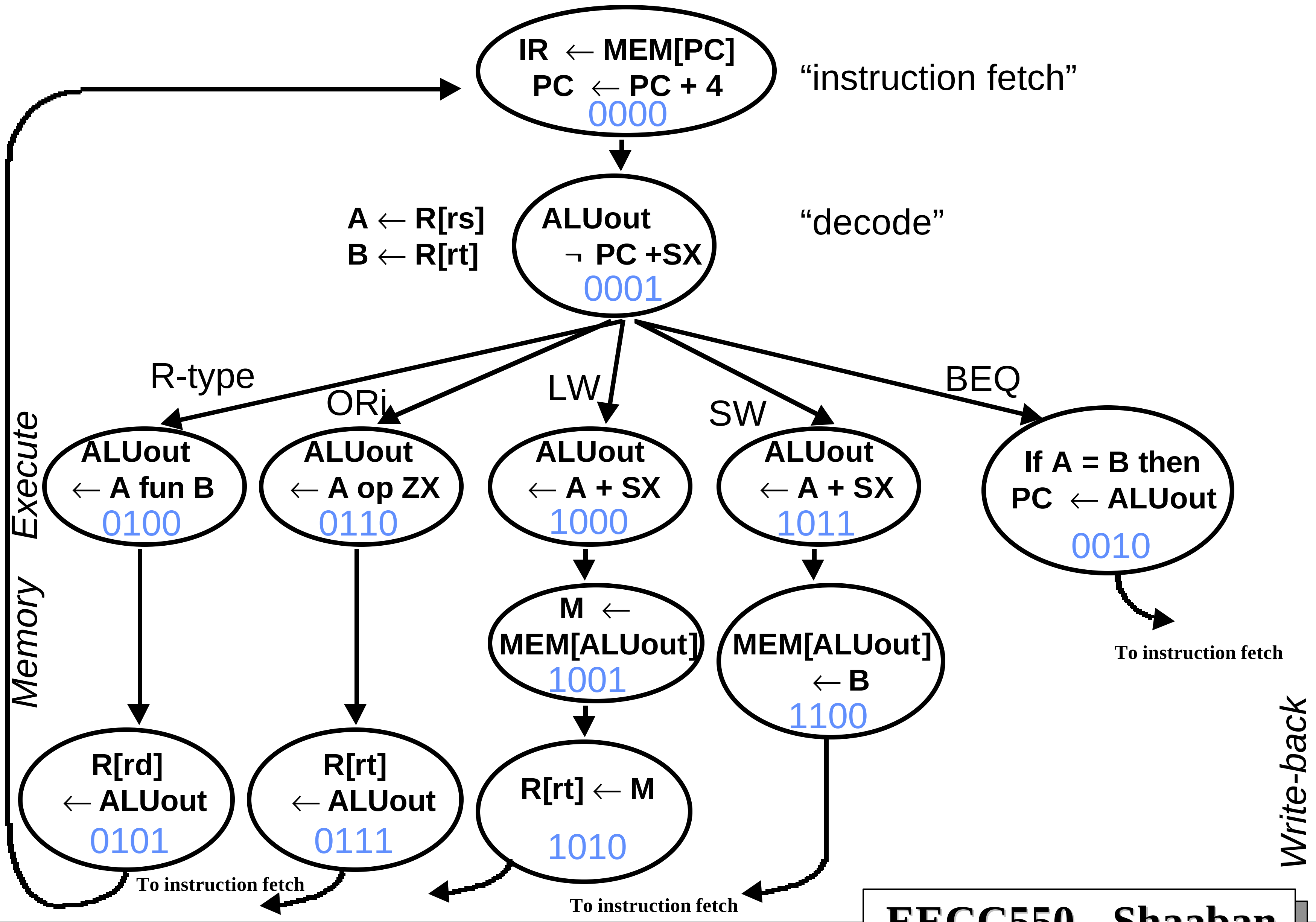
10 (RI) $\rightarrow$ Reserved Instruction exception.

12 (OVF) $\rightarrow$ Arithmetic overflow exception.

The MIPS Multicycle Datapath With Exception Handling Added

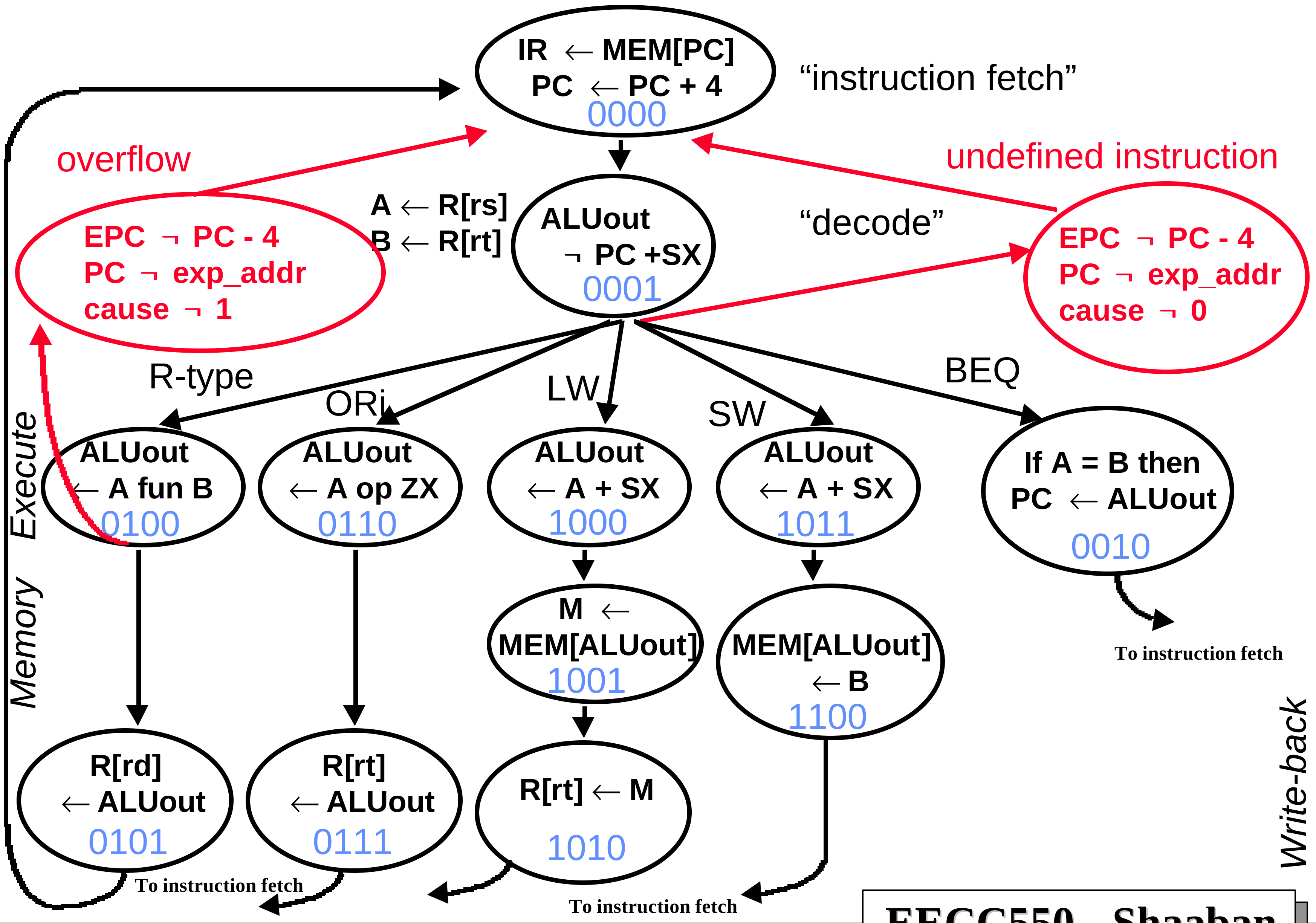


Finite State Machine (FSM) Specification



EECC550 - Shaaban

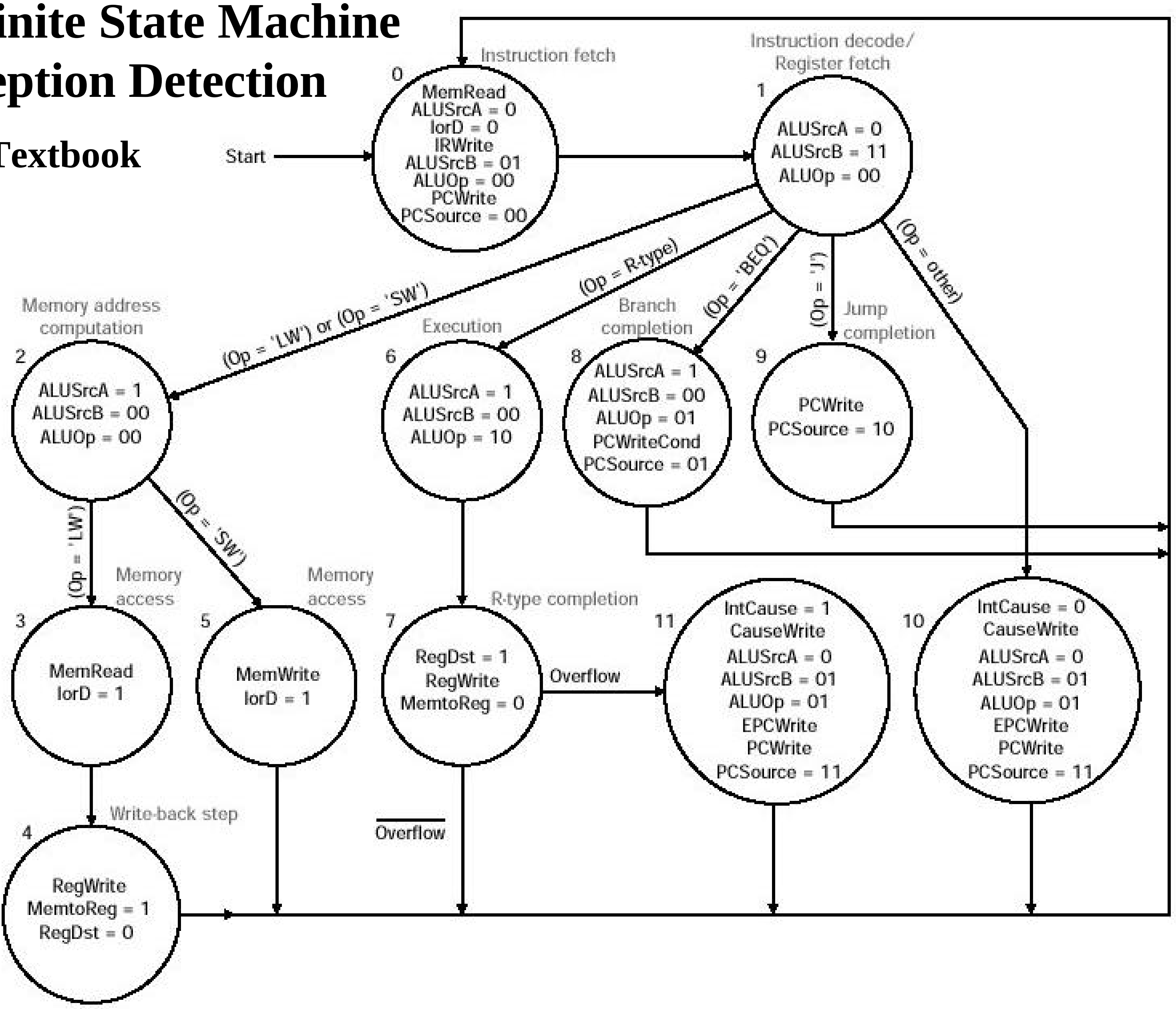
FSM Control Specification To Handle Exceptions



EECC550 - Shaaban

Control Finite State Machine With Exception Detection

Version In Textbook Figure 5.50


Start
0
Instruction fetch
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
1
Instruction decode/ Register fetch
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
(Op = other)
2
Memory address computation
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 00
6
Execution
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
8
Branch completion
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
Jump completion
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
3
Memory access
MemRead
IorD = 1
5
Memory access
MemWrite
IorD = 1
7
R-type completion
RegDst = 1
RegWrite
MemtoReg = 0
Overflow
11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
4
Write-back step
RegWrite
MemtoReg = 1
RegDst = 0
$\overline{\text{Overflow}}$