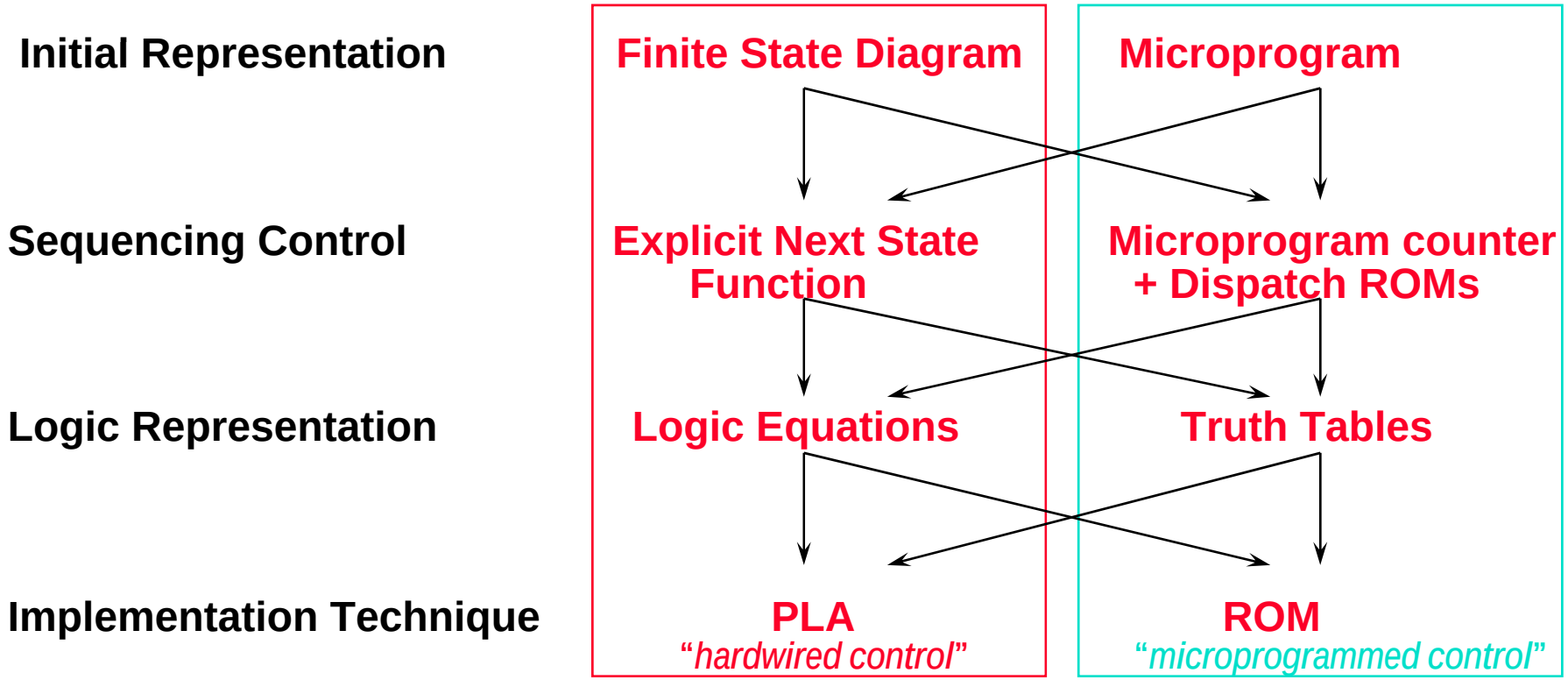
Computer Architecture
Ch. 5-5: Microprogramming and Exceptions

Spring, 2005

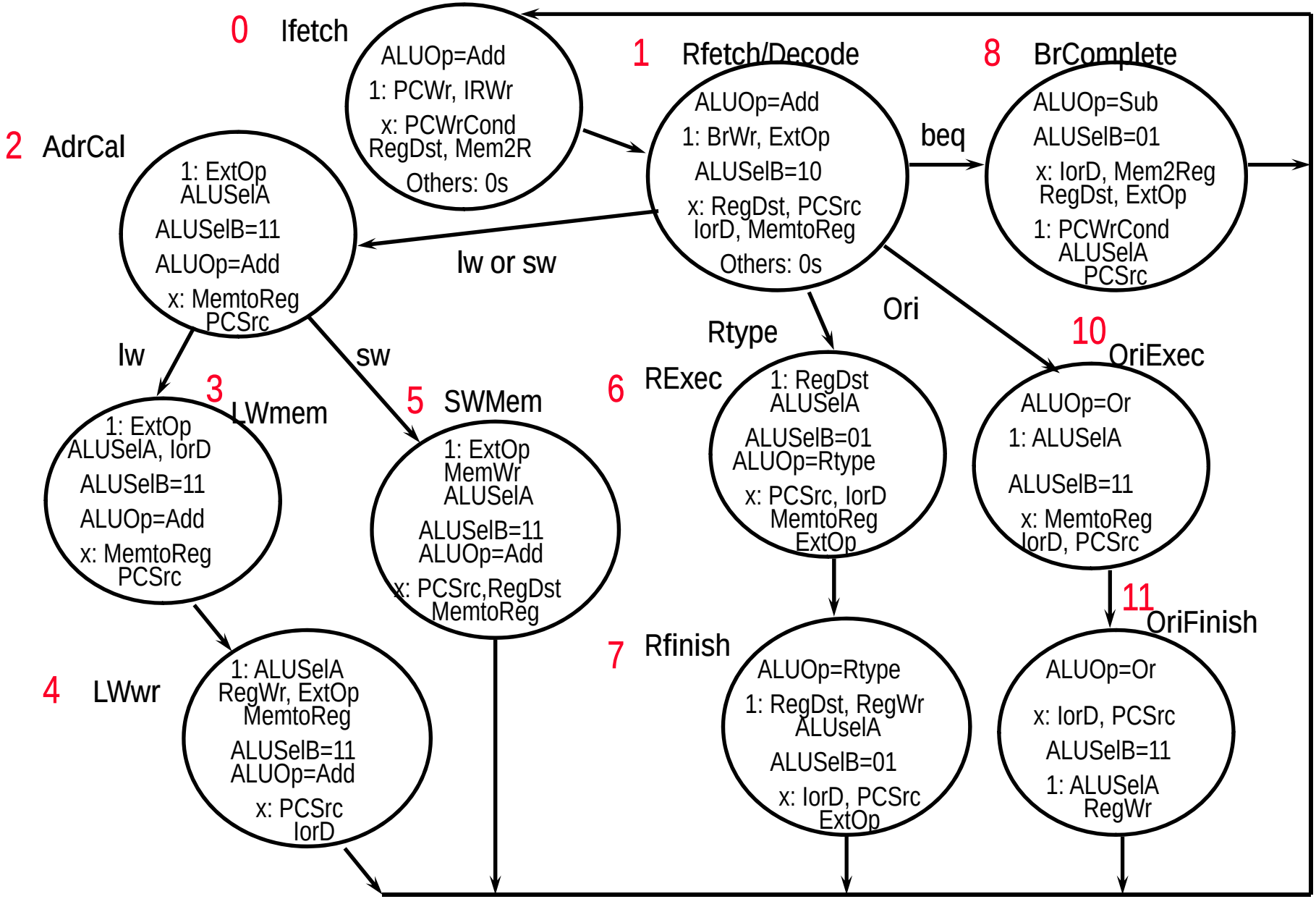
Sao-Jie Chen (csj@cc.ee.ntu.edu.tw)

Overview of the Previous Lectures

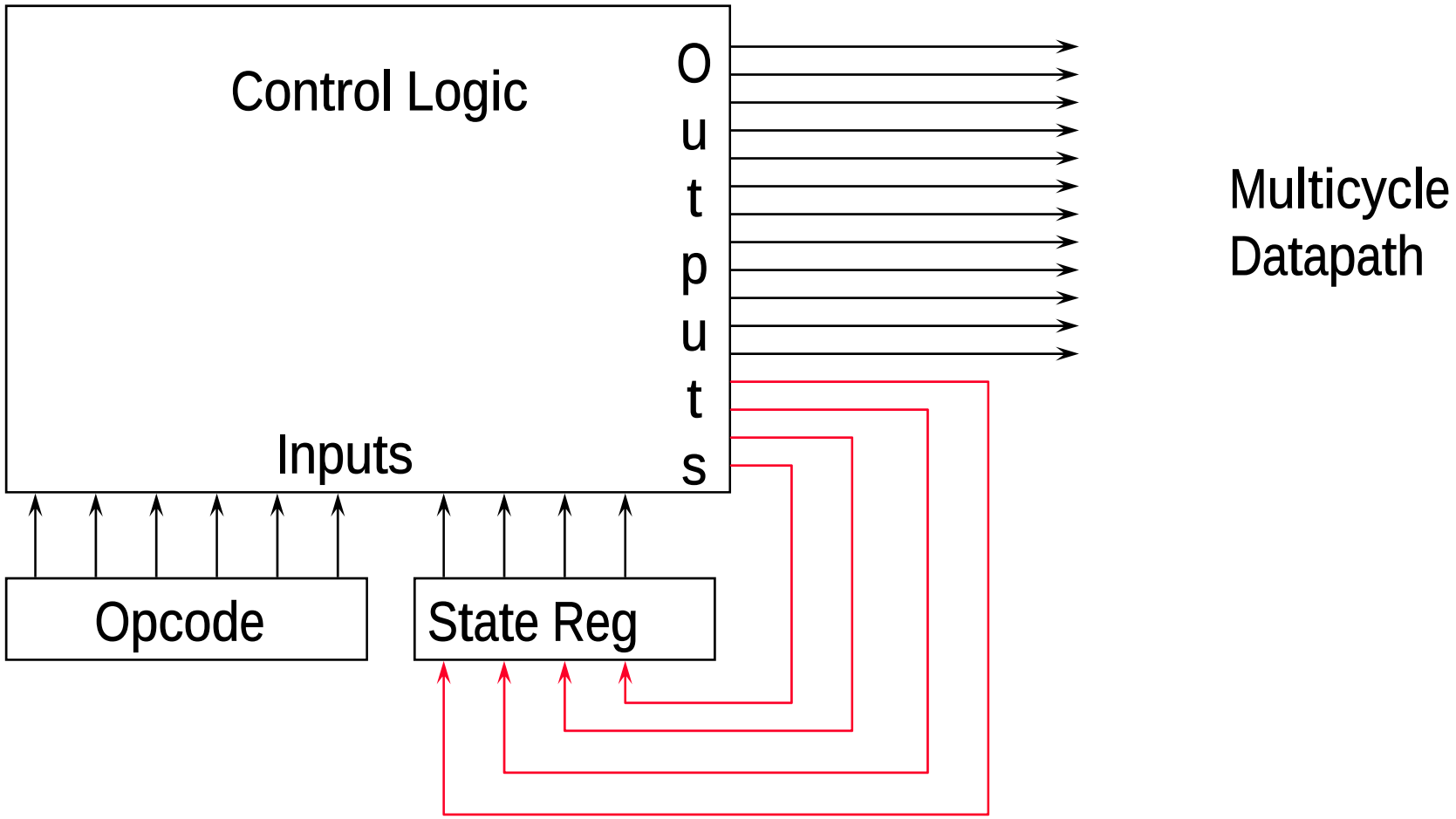
- Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.



Initial Representation: Finite State Diagram



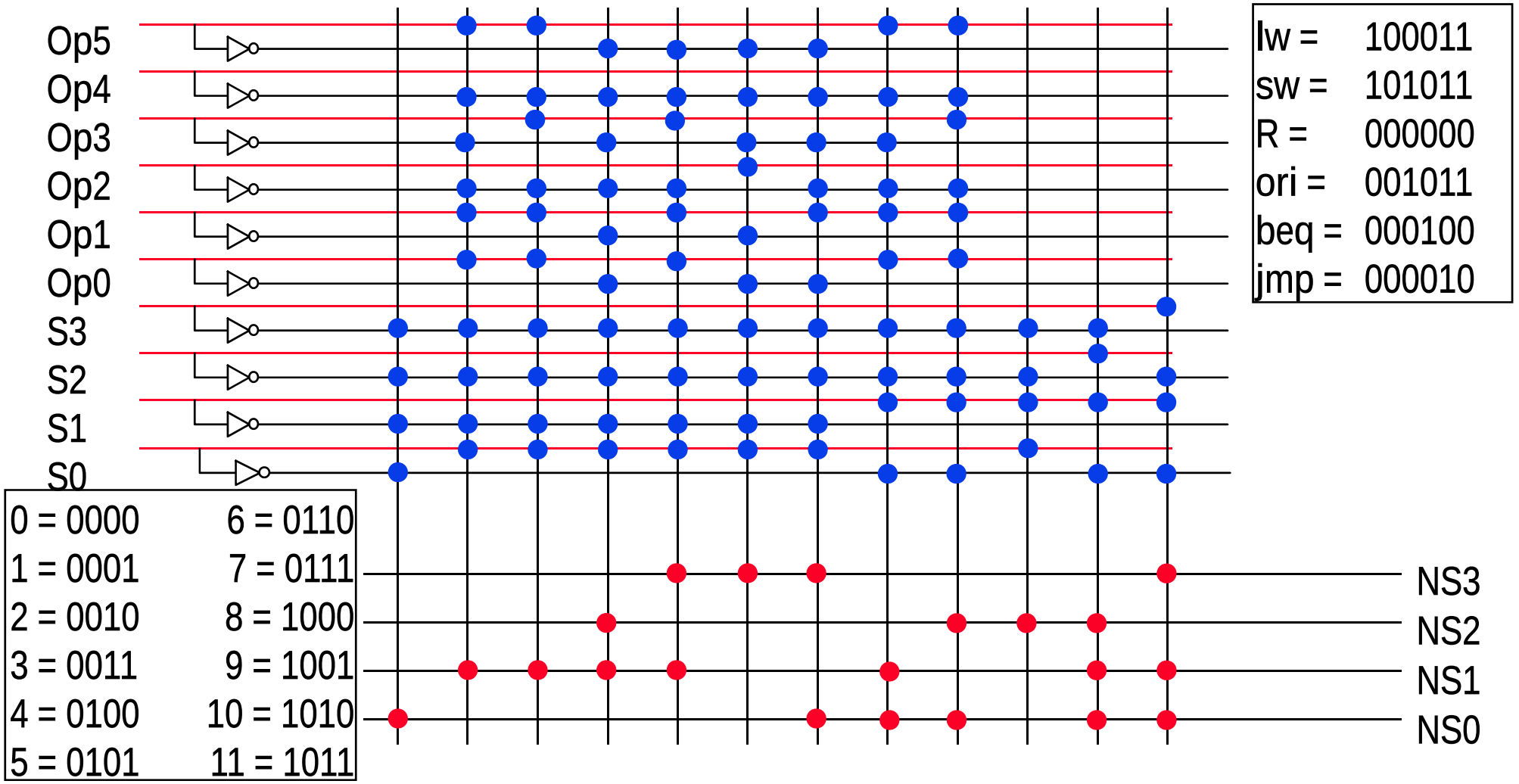
Sequencing Control: Explicit Next State Function



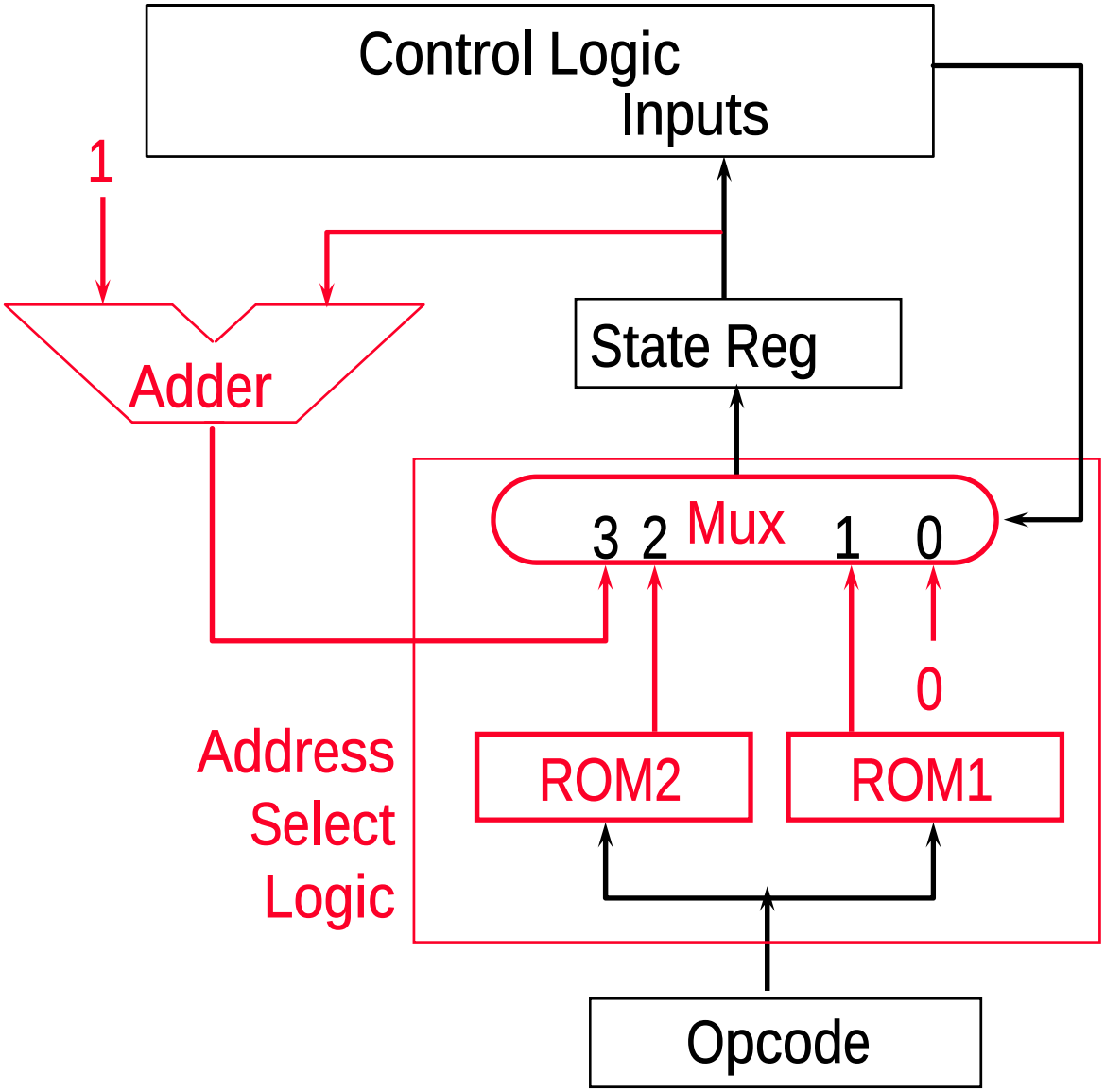
- **Next state number is encoded just like datapath controls**

Implementation Technique: Programmed Logic Arrays

- **Each output line the logical OR of logical AND of input lines or their complement: AND minterms specified in top AND plane, OR sums specified in bottom OR plane**



Sequencer-based control unit details



Dispatch ROM 1

Op	*Name*	*State*
000000	Rtype	0110
000010	jmp	1001
000100	beq	1000
001011	ori	1010
100011	lw	0010
101011	sw	0010

Dispatch ROM 2

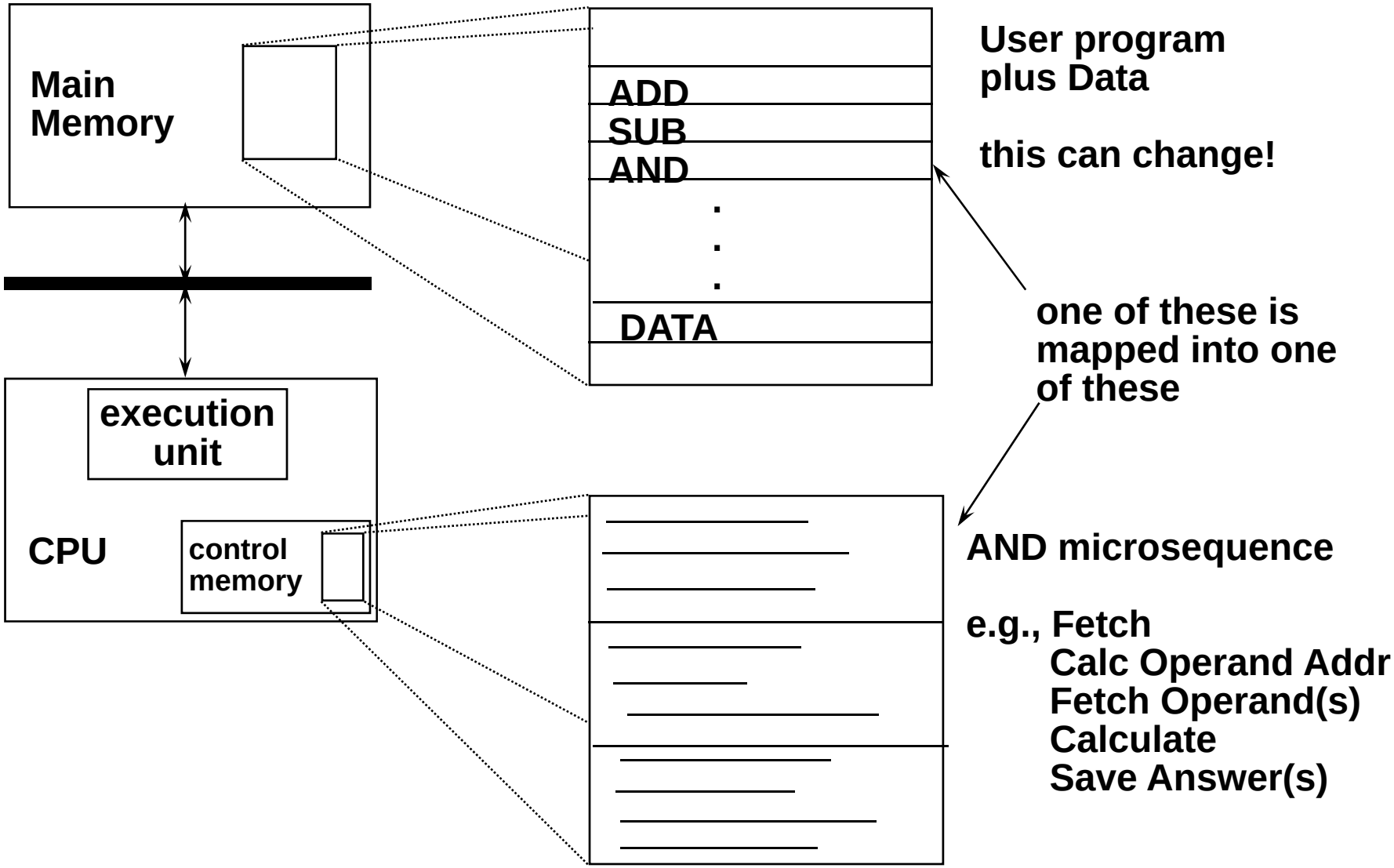
Op	*Name*	*State*
100011	lw	0011
101011	sw	0101

Implementing Control with a ROM

- Instead of a PLA, use a ROM with one word per state ("control word")?

State number	*Control Word Bits 18-2*	*Control Word Bits 1-0*
0	10010100000001000	11
1	00000000010011000	01
2	00000000000010100	10
3	00110000000010100	11
4	00110010000010110	00
5	00101000000010100	00
6	00000000001000100	11
7	00000000001000111	00
8	01000000100100100	00
9	10000001000000000	00
10		11
11		00

Macroinstruction Interpretation



Designing a Microinstruction Set

1. Start with list of control signals
2. Group signals together that make sense: called "fields"
3. Places fields in some logical order (ALU operation & ALU operands first and microinstruction sequencing last)
4. Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals
5. To minimize the width, encode operations that will never be used at the same time

1. Start with list of control signals, 2. Grouped into fields

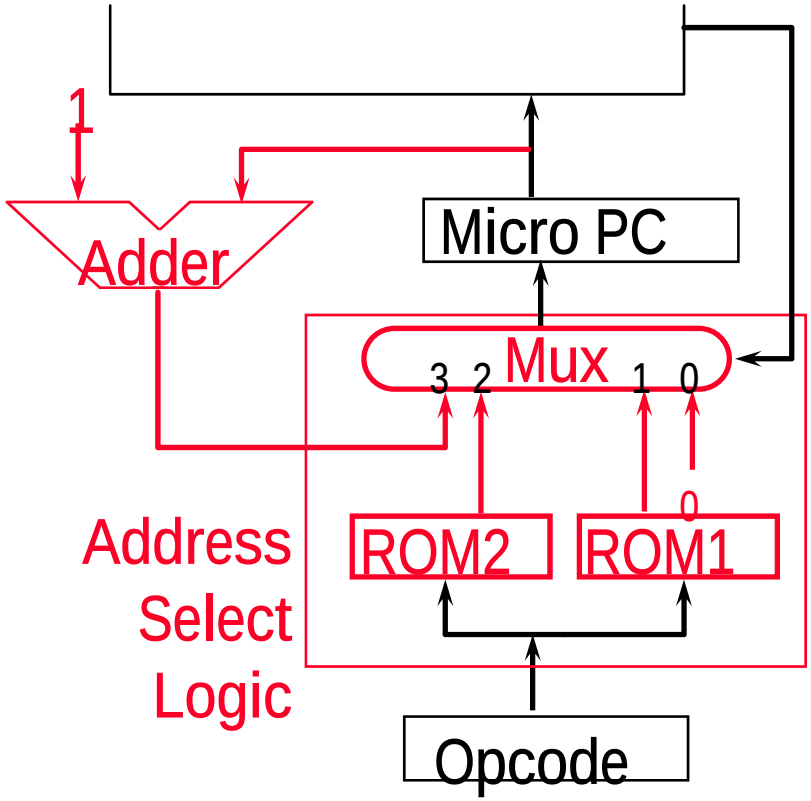
Signal name	*Effect when deasserted*	*Effect when asserted*
ALUSelA	1st ALU operand = PC	1st ALU operand = Reg[rs]
RegWrite	None	Reg. is written
MemtoReg	Reg. write data input = ALU	Reg. write data input = memory
RegDst	Reg. dest. no. = rt	Reg. dest. no. = rd
TargetWrite	None	Target reg. = ALU
MemRead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

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]
PCSource	00	PC = ALU
	01	PC = Target
	10	PC = PC+4[29-26] : IR[25-0] << 2

Start with list of control signals (1 & 2 cont'd)

- For next state function (next microinstruction address), use Sequencer-based control unit , called Micro-PC or μPC (vs. State Reg.)

Signal	*Value*	*Effect*
Sequen	00	Next μaddress = 0
-cing	01	Next μaddress = dispatch ROM 1
	10	Next μaddress = dispatch ROM 2
	11	Next μaddress = μaddress + 1



3. Microinstruction Format

Field Name	*Width*	*Control Signals Set*
ALU Control	2	ALUOp
SRC1	1	ALUSelA
SRC2	3	ALUSelB
ALU Destination	4	RegWrite, MemtoReg, RegDst, TargetWrite
Memory	3	MemRead, MemWrite, IorD
Memory Register	1	IRWrite
PCWrite Control	4	PCWrite, PCWriteCond, PCSource
Sequencing	2	AddrCtl
Total	20	

4. Legend of Fields and Symbolic Names

Field Name	*Values for Field*	*Function of Field with Specific Value*
ALU	Add	ALU adds
	Subt.	ALU subtracts
	Funct 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 ext. sl2 IR[15-0]
	rt	2nd ALU input = Reg[rt]
ALU destination	Target	Target = ALU
	rd	Reg[rd] = ALU
Memory	Read PC	Read memory using PC
	Read ALU	Read memory using ALU output
	Write ALU	Write memory using ALU output
Memory register	IR	IR = Mem
	Write rt	Reg[rt] = Mem
	Read rt	Mem = Reg[rt]
PC write	ALU	PC = ALU output
	Target-cond.	IF ALU Zero then PC = Target
	jump addr.	PC = PCSource
Sequencing	Seq	Go to sequential μinstruction
	Fetch	Go to the first microinstruction
	Dispatch i	Dispatch using ROMi (1 or 2).

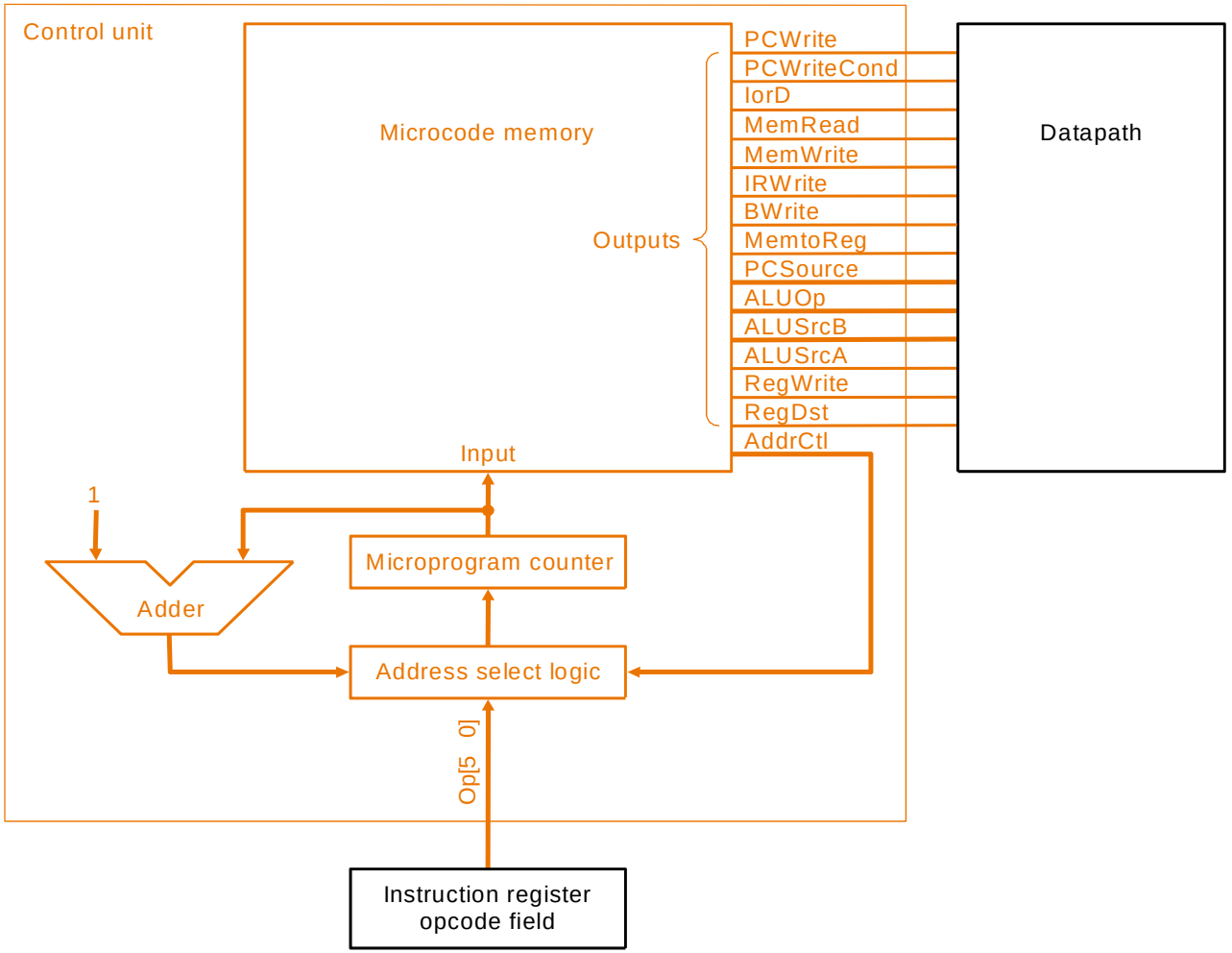
Microprogram it yourself!

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

Microprogram it yourself!

Label	*ALU*	*SRC1*	*SRC2*	*ALU Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequencing*
Fetch	Add	PC	4		Read PC	IR	ALU	Seq
	Add	PC	Extshft	Target				Dispatch 1
LWSW1	Add	rs	Extend					Dispatch 2
LW2	Add	rs	Extend		Read ALU			Seq
	Add	rs	Extend		Read ALU	Write rt		Fetch
SW2	Add	rs	Extend		Write ALU	Read rt		Fetch
Rtype	Funct	rs	rt					Seq
	Funct	rs	rt	rd				Fetch
BEQ1	Subt.	rs	rt				Target-cond.	Fetch
JUMP1							jump address	Fetch
ORI	Or	rs	Extend0					Seq
	Or	rs	Extend0	rd				Fetch

Microprogramming



- **What are the “microinstructions” ?**

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 microassembler do?*

Microinstruction format (Fig. C.5.1)

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, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 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

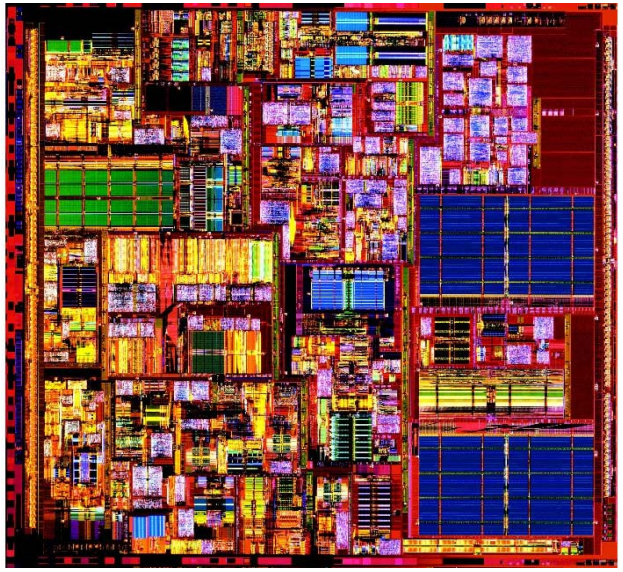
- Distinction between specification and implementation is sometimes blurred
- Specification Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel
- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - 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

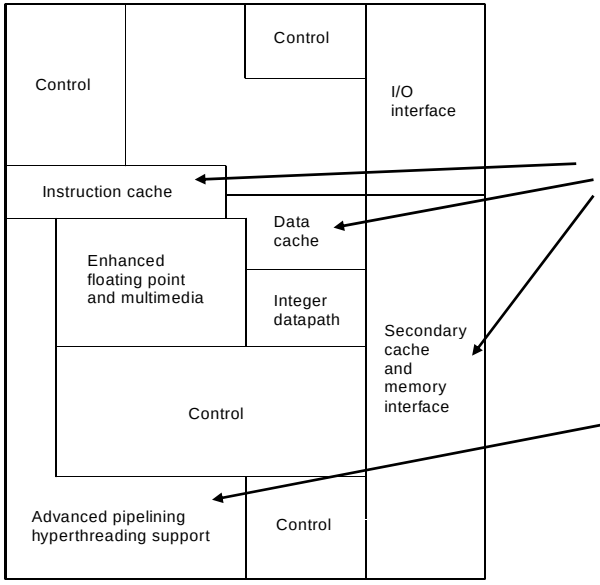
Historical Perspective

- In the ‘60s and ‘70s microprogramming was very important for implementing machines
- This led to more sophisticated ISAs and the VAX
- In the ‘80s RISC processors based on pipelining became popular
- Pipelining the microinstructions is also possible!
- Implementations of IA-32 architecture processors since 486 use:
  - “hardwired control” for simpler instructions
    (few cycles, FSM control implemented using PLA or random logic)
  - “microcoded control” for more complex instructions
    (large numbers of cycles, central control store)
- The IA-64 architecture uses a RISC-style ISA and can be implemented without a large central control store

Pentium 4

- **Pipelining is important (last IA-32 without it was 80386 in 1985)**



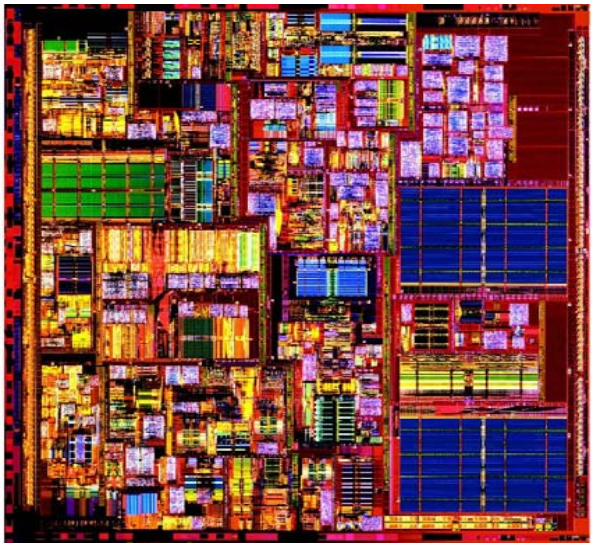


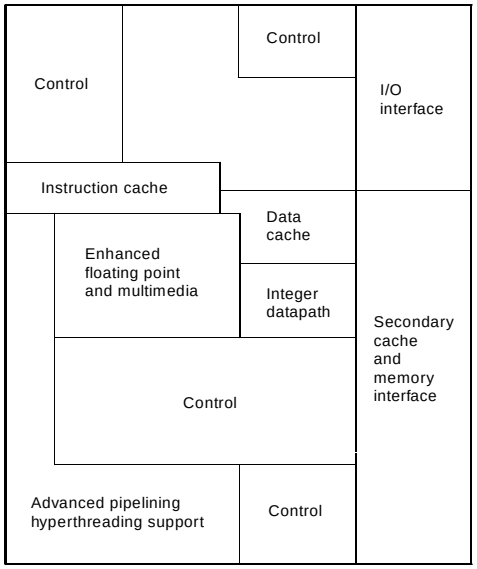
- **Pipelining is used for the simple instructions favored by compilers**

"Simply put, a high performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize the complex, less frequently used, instructions"

Pentium 4

- Somewhere in all that “control we must handle complex instructions





- Processor executes simple microinstructions, 70 bits wide (hardwired)
- 120 control lines for integer datapath (400 for floating point)
- If an instruction requires more than 4 microinstructions to implement, control from microcode ROM (8000 microinstructions)
- Its complicated!

Microprogram Summary

- If we understand the instructions…

  We can build a simple processor!

- If instructions take different amounts of time, multi-cycle is better

- Datapath implemented using:
  - Combinational logic for arithmetic
  - State holding elements to remember bits

- Control implemented using:
  - Combinational logic for single-cycle implementation
  - Finite state machine for multi-cycle implementation

Exceptions and Interrupts

- Control is hardest part of the design
- Hardest part of control is exceptions and interrupts
  - events other than branches or jumps that change the normal flow of instruction execution
  - exception is an unexpected event from within the processor; e.g., arithmetic overflow
  - interrupt is an unexpected event from outside the processor; e.g., I/O
- MIPS convention: exception means any unexpected change in control flow, without distinguishing internal or external; use the term interrupt only when the event is externally caused.

Type of event	*From where?*	*MIPS terminology*
I/O device request	External	Interrupt
Invoke OS from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or Interrupt

How are Exceptions Handled?

- Machine must save the address of the offending instruction in the EPC (exception program counter)
- Then transfer control to the OS at some specified address
  - OS performs some action in response, then terminates or returns using EPC
- Two types of exceptions in our current implementation: undefined instruction and an arithmetic overflow
- Which Event caused Exception?
  - Option 1 (used by MIPS): a Cause register contains reason
  - Option 2 Vectored interrupts: address determines cause.
    - ⇒ addresses separated by 32 instructions, e.g.,

Exception Type	*Exception Vector Address (in Binary)*
Undefined instruction	01000000 00000000 00000000 0**0**000000$_{two}$
Arithmetic overflow	01000000 00000000 00000000 0**1**000000$_{two}$

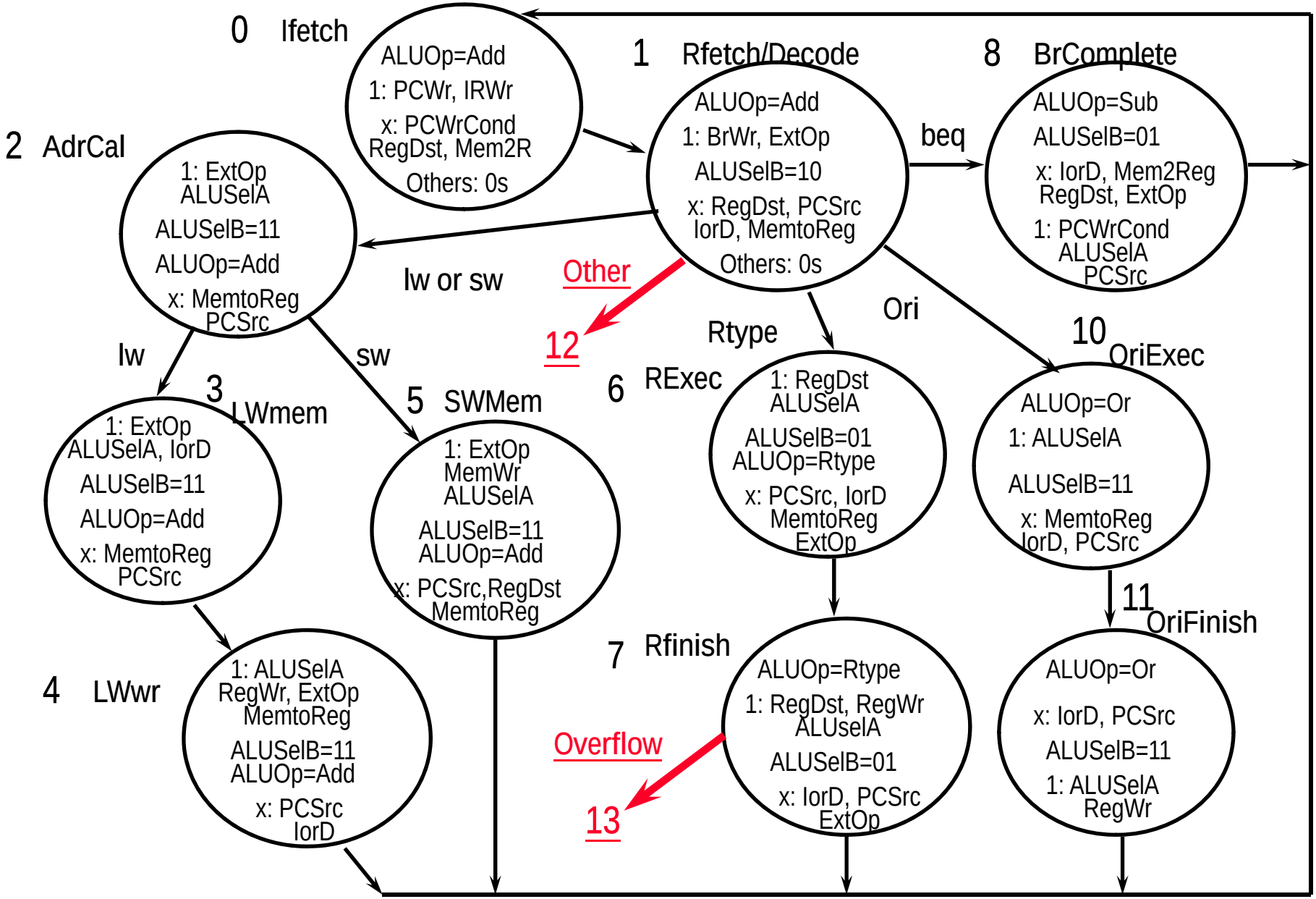
Additions to MIPS ISA to support Exceptions

- **EPC: a 32-bit register used to hold the address of the affected instruction.**
- **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 the low-order bit of this register encodes the two possible exception sources mentioned above: undefined instruction=0 and arithmetic overflow=1.**
- **2 control signals to write EPC and Cause**
- **Be able to write exception address into PC, increase mux to add as input $01000000\ 00000000\ 00000000\ 00000000_{two}$**
- **May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor); PC = PC - 4**

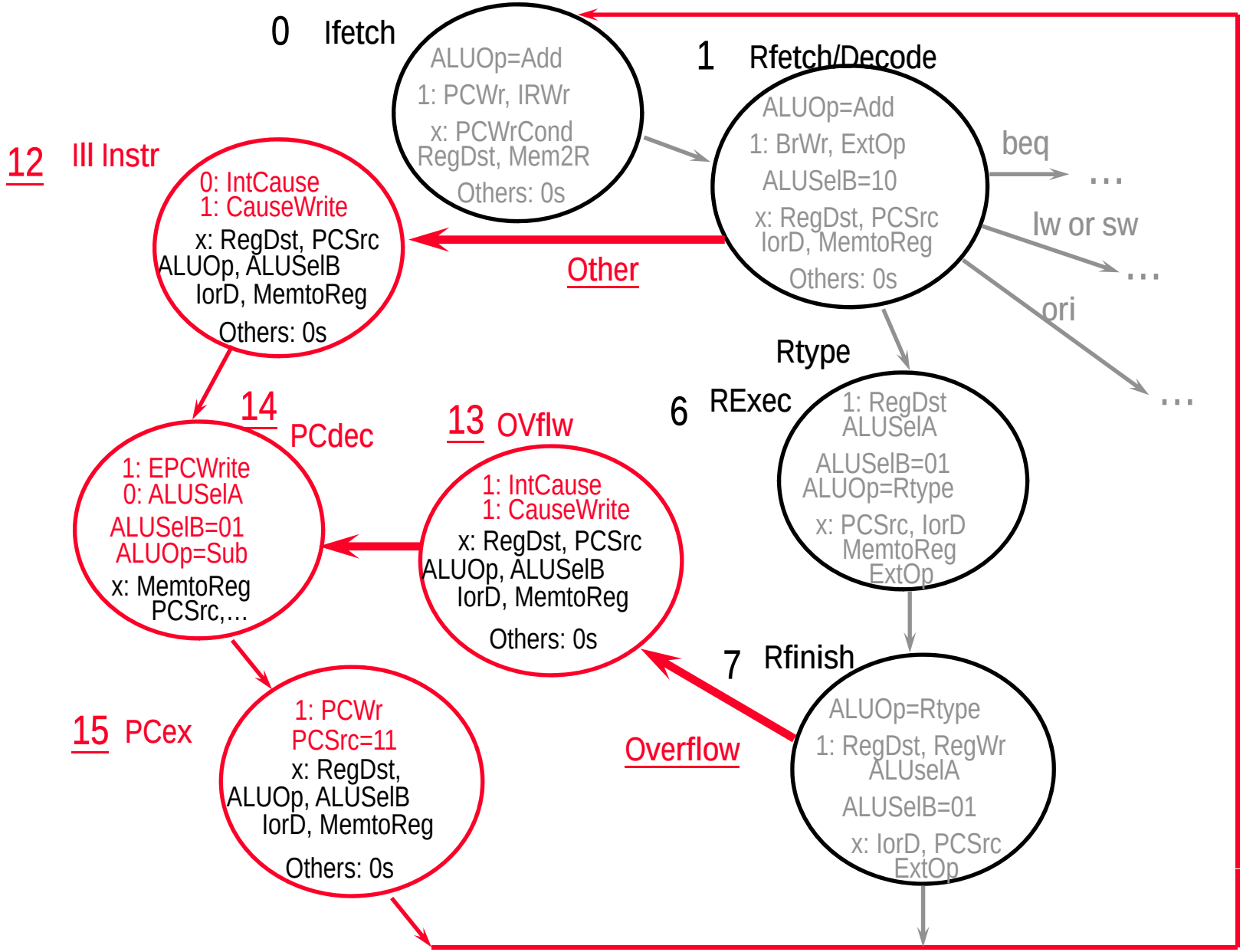
How Control Detects Exceptions

- Undefined Instruction detected when no next state is defined from state 1 for the op value.
  - We handle this exception by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, beq, and ori as new state 12.
  - Shown symbolically using “other” to indicate that the op field does not match any of the opcodes that label arcs out of state 1.
- Arithmetic overflow—Chapter 3 included logic in the ALU to detect overflow, and a signal called Overflow is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state for state 7
- Note: Challenge in designing control of a real machine is to handle different interactions between instructions and other exception-causing events such that control logic remains small and fast.
  - Complex interactions makes the control unit the most challenging aspect of hardware design

Changes to Finite State Diagram to Detect Exceptions



Extra States to Handle Exceptions



What happens to Instruction with Exception?

- Some problems could occur in the way the exceptions are handled.
- For example, in the case of arithmetic overflow, the instruction causing the overflow completes writing its result, because the overflow branch is in the state when the write completes.
- However, the architecture may define the instruction as having no effect if the instruction causes an exception; MIPS specifies this.
- When get to virtual memory we will see that certain classes of exceptions prevent the instruction from changing the machine state.
- This aspect of handling exceptions becomes complex and potentially limits performance.

Summary

- Control is hard part of computer design
- Microprogramming specifies control like assembly language programming instead of finite state diagram
- Next State function, Logic representation, and implementation technique can be the same as finite state diagram, and vice versa
- Exceptions are the hard part of control
- Need to find convenient place to detect exceptions and to branch to state or microinstruction that saves PC and invokes the operating system
- As we get pipelined CPUs that support page faults on memory accesses which means that the instruction cannot complete AND you must be able to restart the program at exactly the instruction with the exception, it gets even harder