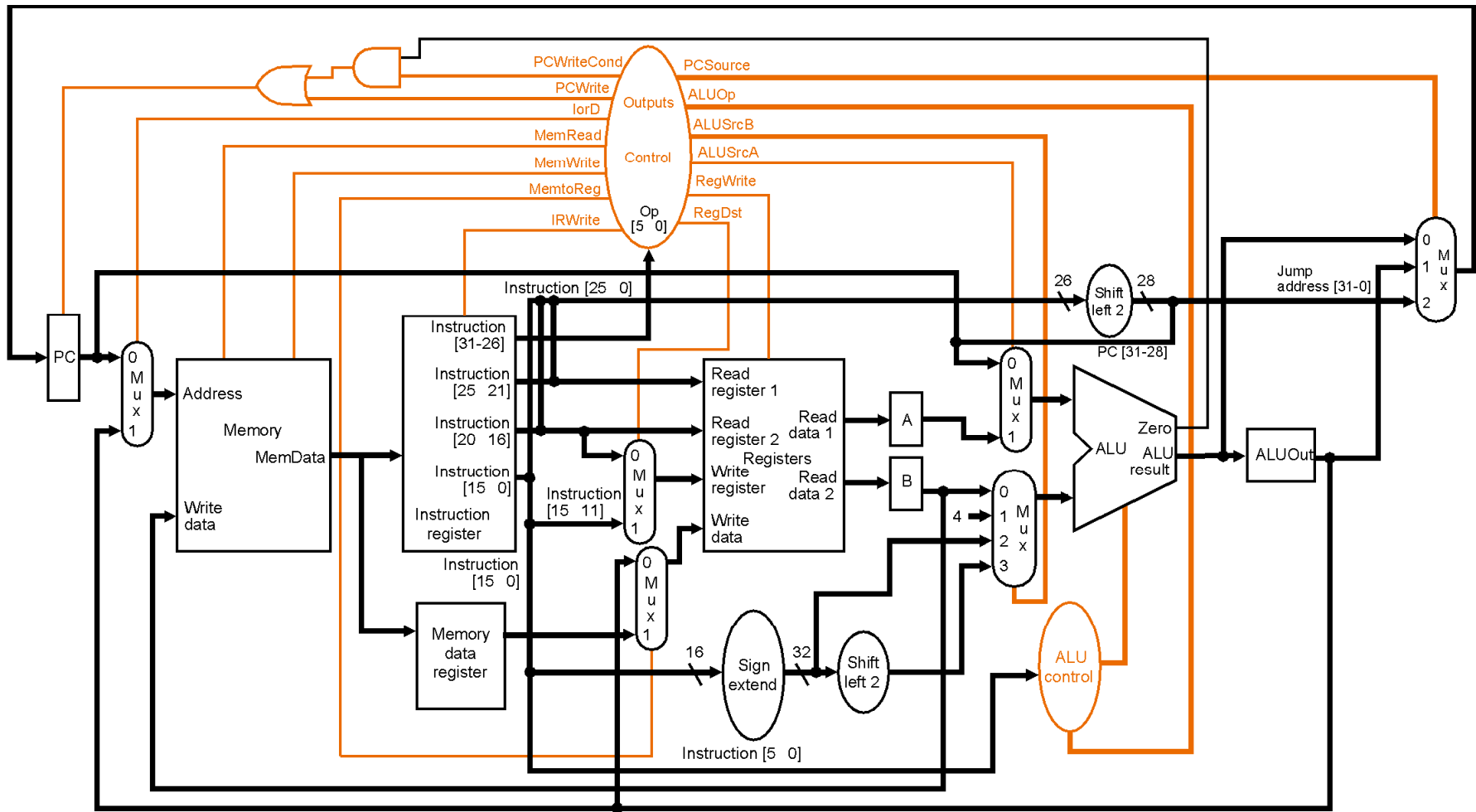
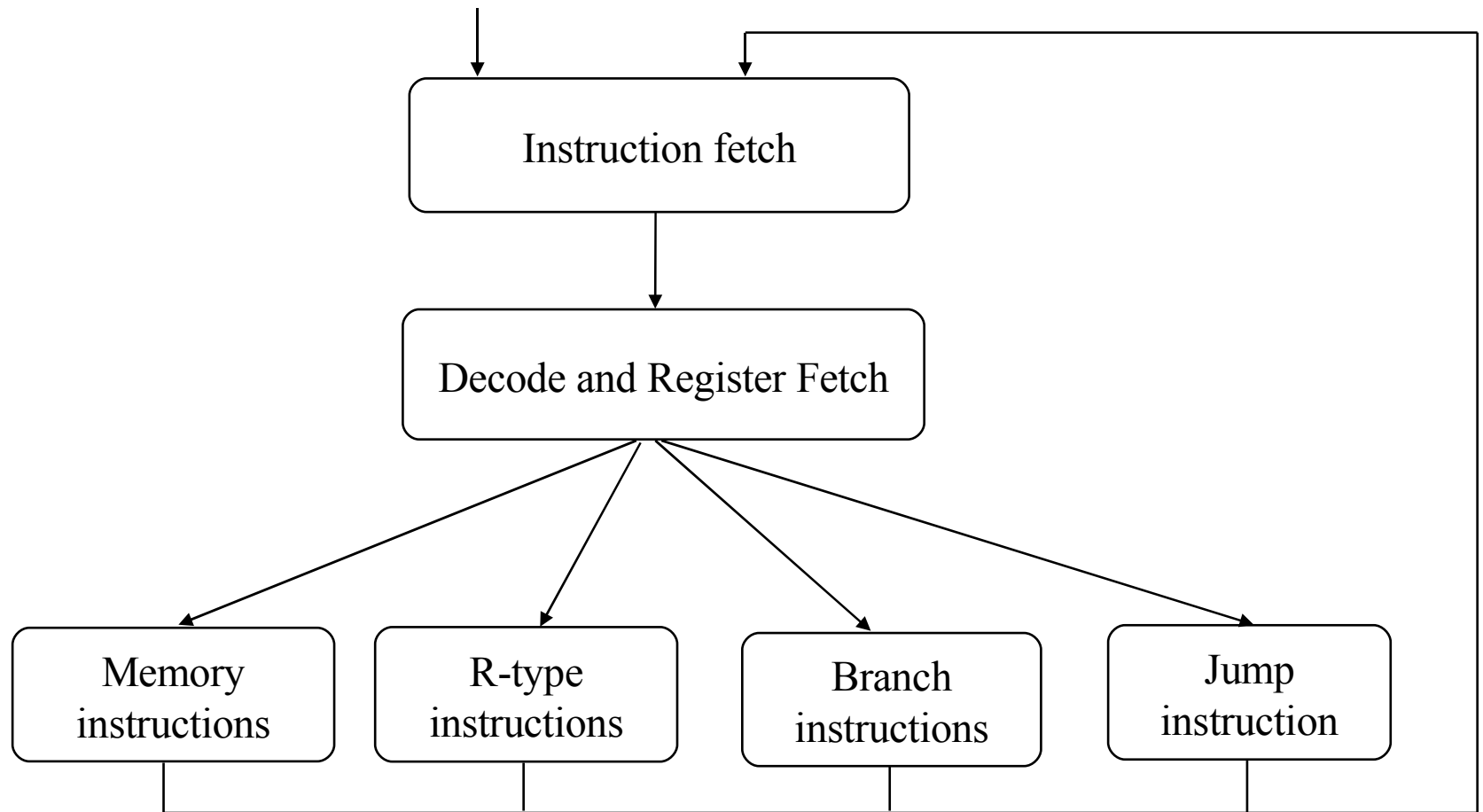


Historical Context: Simplifying Control Design Through Microprogramming

Summary of First Multiple Cycle CPU Design



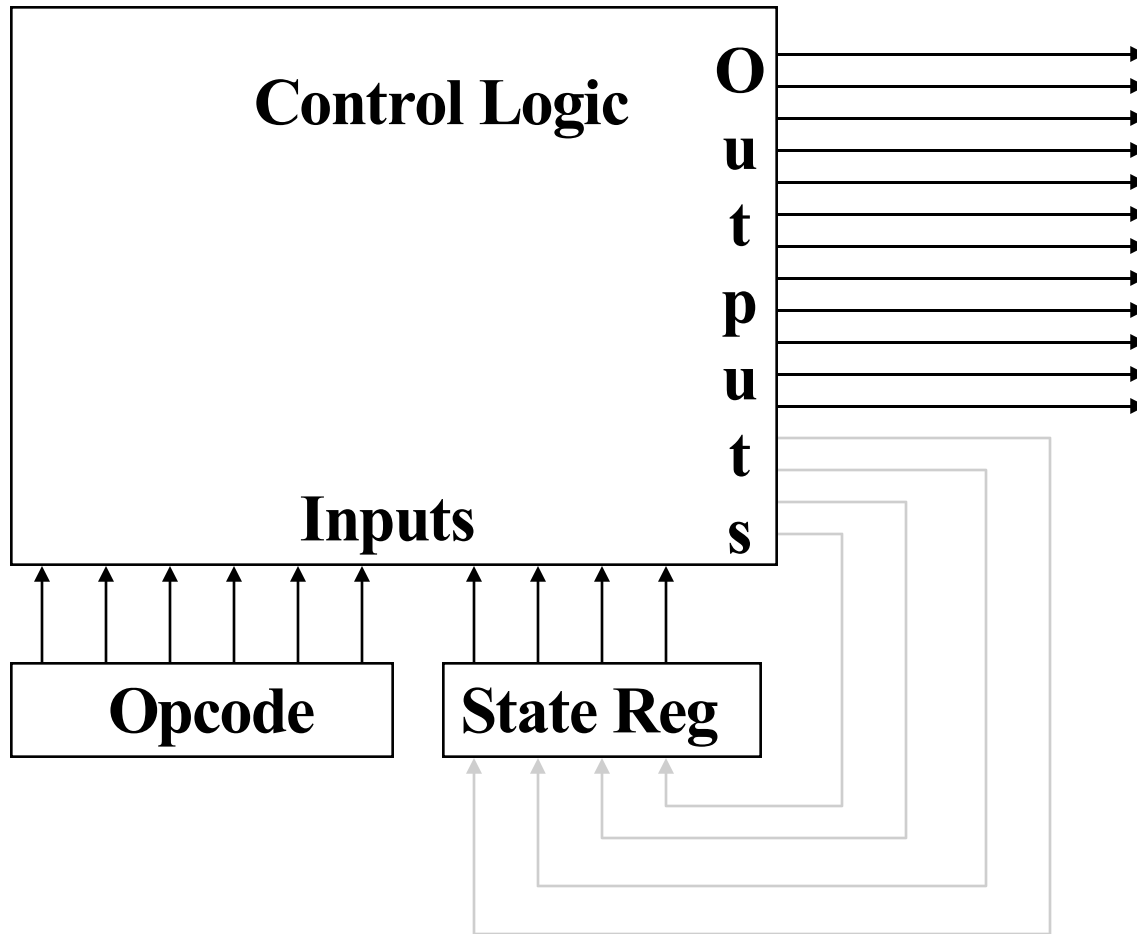
Summary of First Multiple Cycle CPU Design



The Problem with FSMs as control sequencers

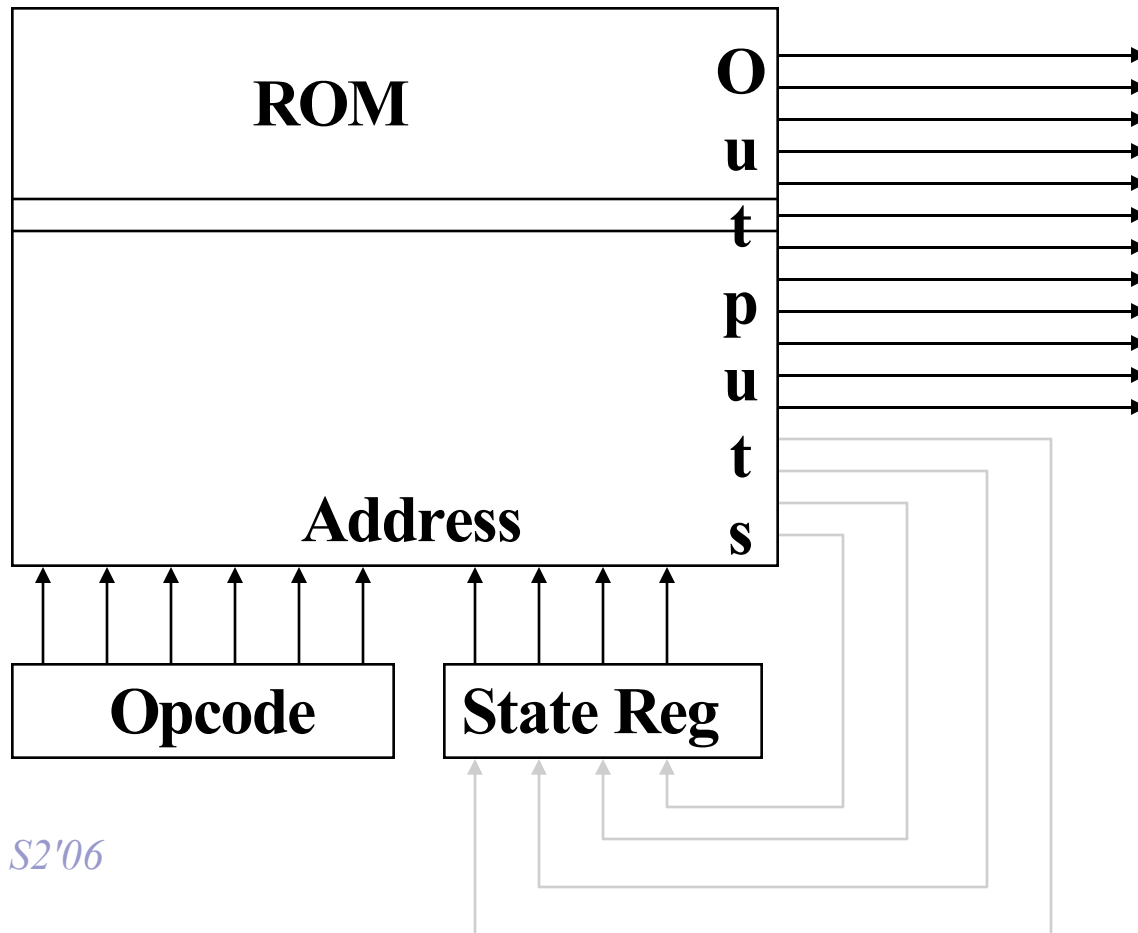
- They get unmanageable quickly as they grow.
 - hard to specify
 - hard to manipulate
 - error prone
 - hard to visualize

Implementing a control FSM

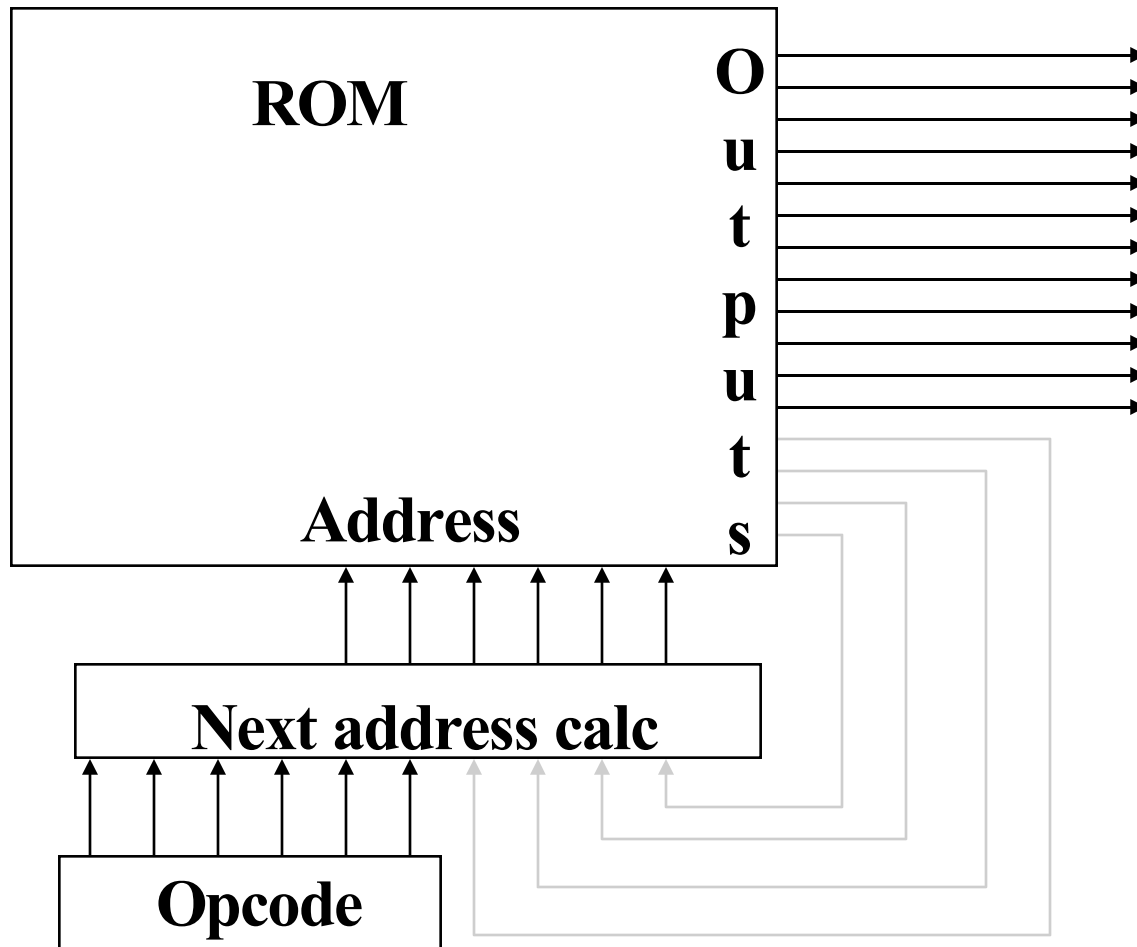


Implementing a control FSM with ROM

Each line in the ROM contains control signal outputs (an operation), and next-state outputs (branch destination)

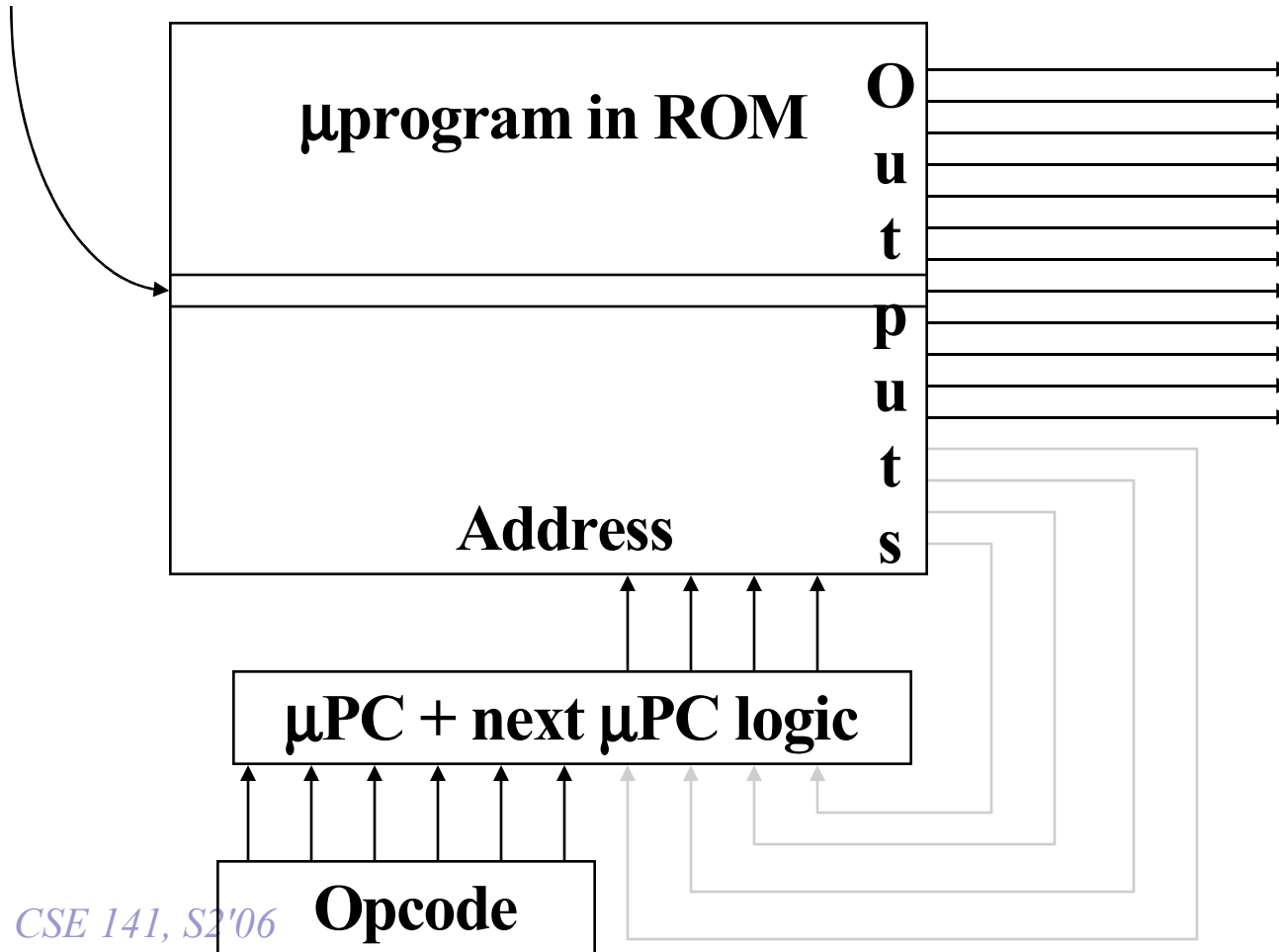


Implementing a control FSM with ???

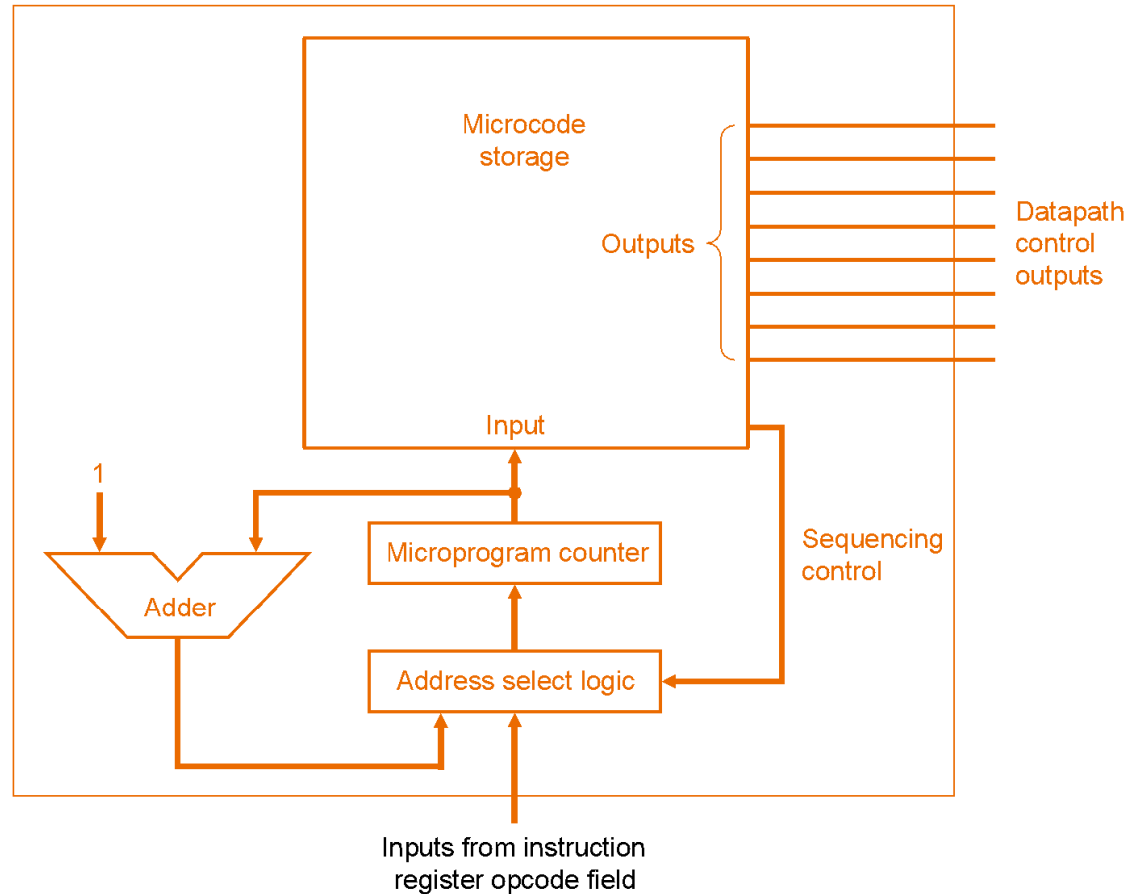


Implementing a control FSM with a microprogram

Each line in the ROM is now a microprogram instruction, corresponding to a FSM state, with an operation (control signals) and branch destination (next state info).



Microprogram Implementation



Microprogramming

- Being able to specify sequences of signals doesn't necessarily make it “easy”.
- Groups of signals are combined and given **symbolic** names.
E.g.,
 - MemRead, ALUSrcA = 0, IorD = 0, ALUSrcB = 01... = “fetch”
 - RegDst=1, MemtoReg=0, regwrite=1 => “rd = ALUout”
 - RegDst=0, MemtoReg=1, regwrite=1 => “rt = MDR”
- So a microprogram might be:

```
start:    fetch
          decode; goto “opcode”
          ...
add:      src1=A; src2=B; add
          rd=ALUout; goto start
```

Microprogramming

- If a microprogram is fundamentally the same as the FSM, what's the big deal?
 - Easier to specify (program), visualize, and manipulate.
 - allows us to think about the control symbolically
- Each microinstruction typically specifies (1) **control** information and (2) **sequencing** information (which microinstruction to execute next).
- There would typically be a one-one correspondence between FSM states and microprogram instructions.
- Microprogramming allowed architectures to change/adapt/be fixed easily. It is also to blame for the extreme CISC architectures. Why?

Exceptions

Exceptions

- There are two sources of non-sequential control flow in a processor
 - explicit branch and jump instructions
 - exceptions
- *Branches* are synchronous and deterministic
- *Exceptions* are typically asynchronous and non-deterministic
- Guess which is more difficult to handle?

(*control flow* refers to the movement of the program counter through memory)

Exceptions and Interrupts

the terminology is not consistent, but we'll refer to

- *exceptions* as any unexpected change in control flow
- *interrupts* as any externally-caused exception

So then, what is:

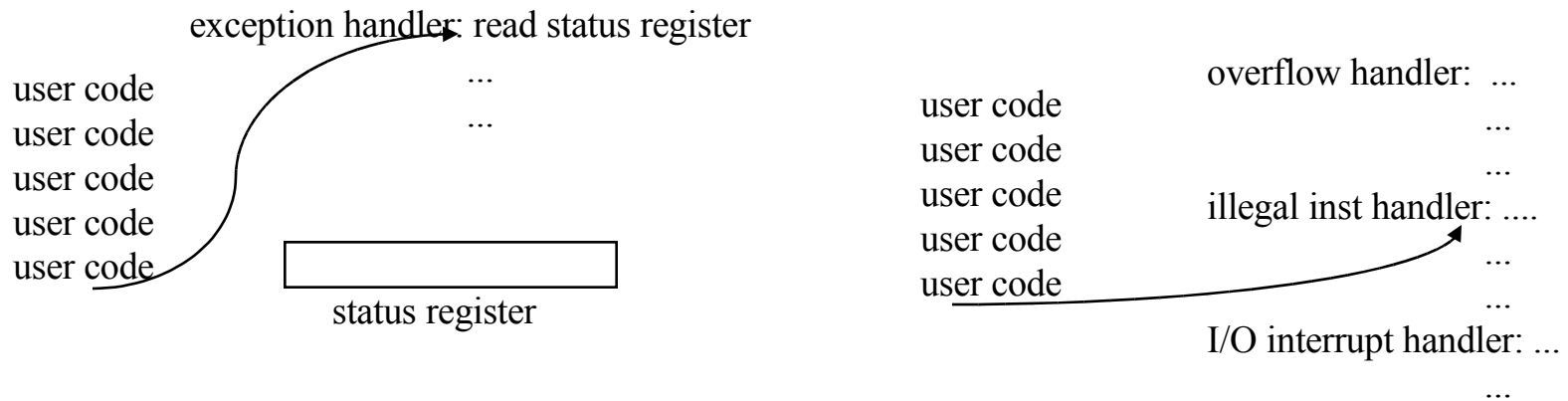
- arithmetic overflow
- divide by zero
- I/O device signals completion to CPU
- user program invokes the OS
- memory parity error
- illegal instruction
- timer signal

For now...

- The machine we've been designing in class can generate two types of exceptions.
 - arithmetic overflow
 - illegal instruction
- On an exception, we need to
 - save the PC (invisible to user code)
 - record the nature of the exception/interrupt
 - transfer control to OS

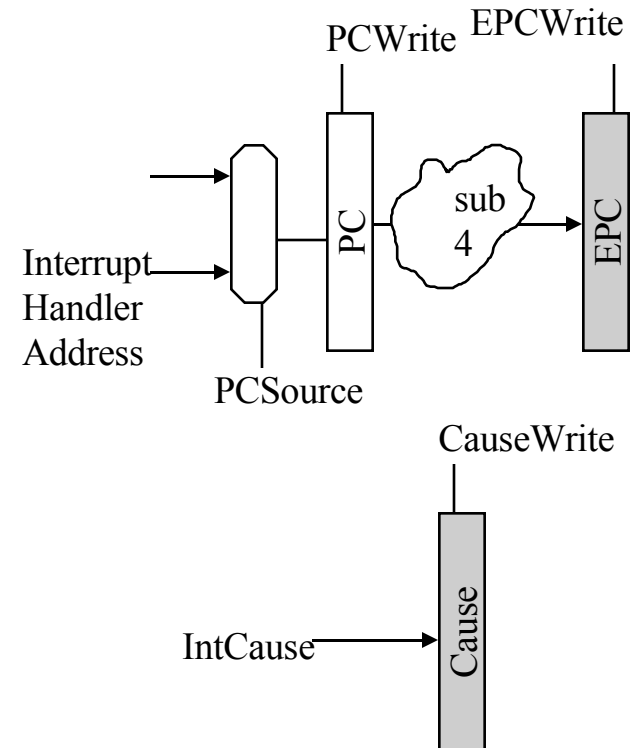
Handling exceptions

- PC saved in EPC (exception program counter), which the OS may read and store in kernel memory
- A status register, and a single exception handler may be used to record the exception and transfer control, or
- A *vectored interrupt* transfers control to a different location for each possible type of interrupt/exception

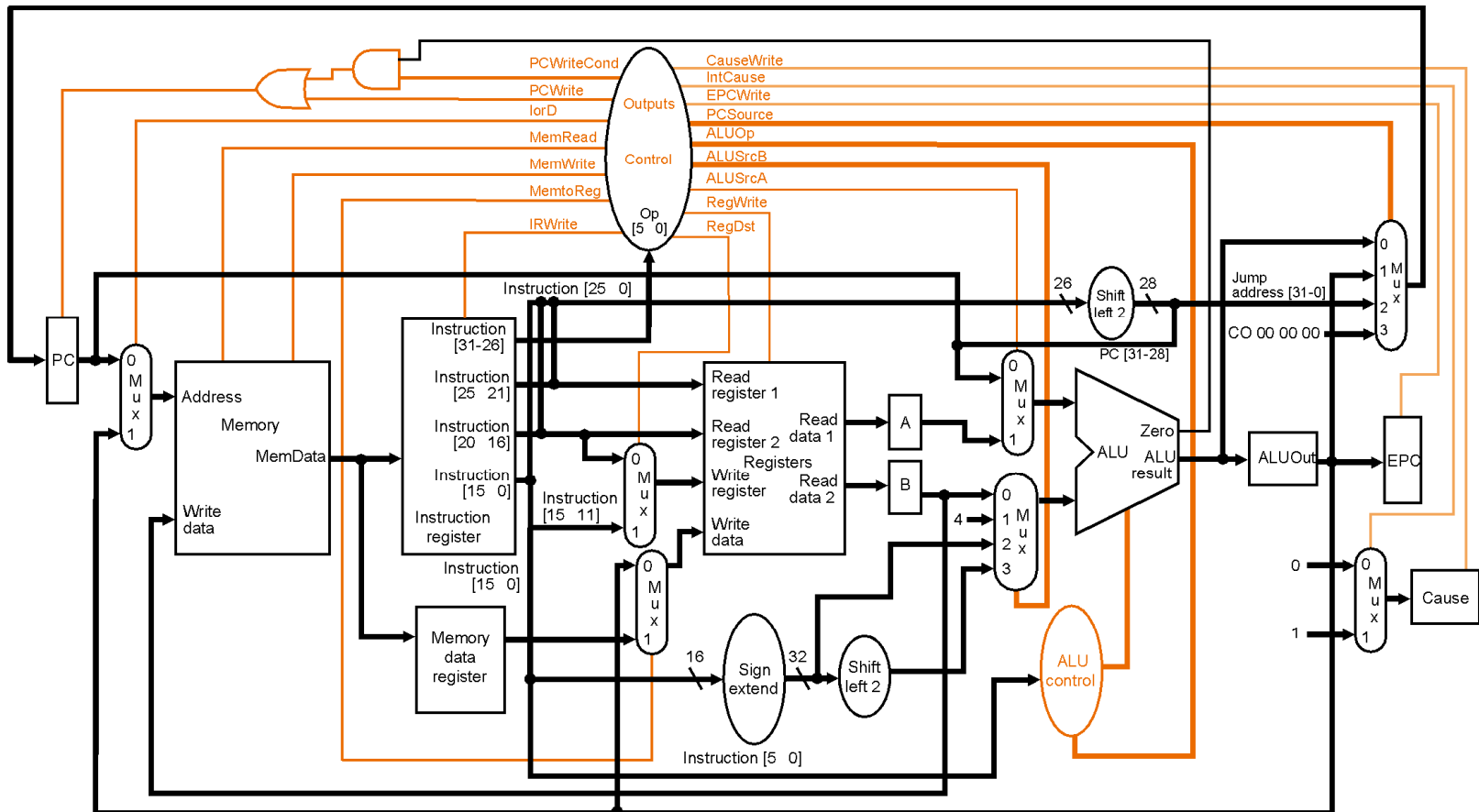


Supporting exceptions

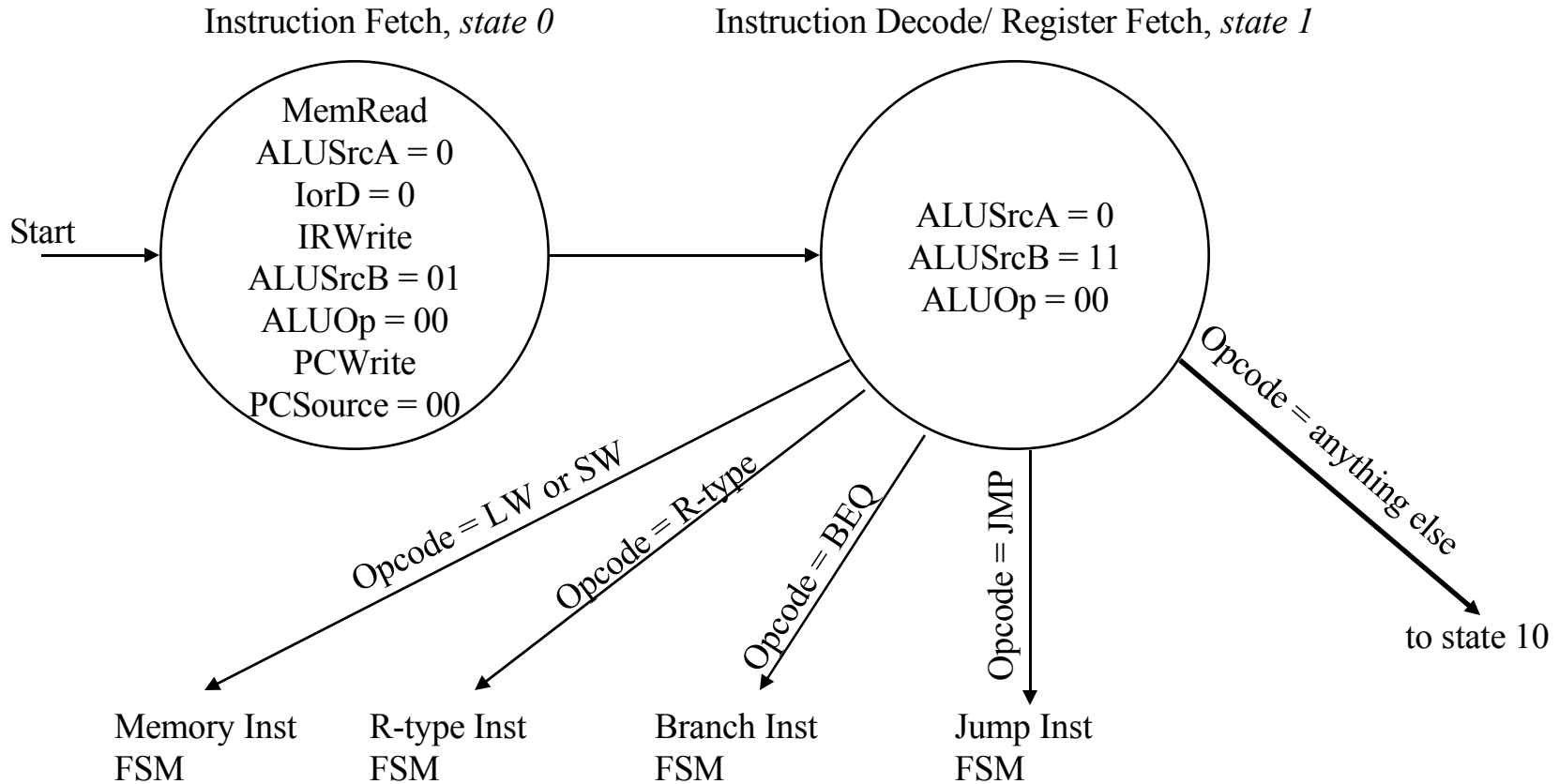
- For our MIPS-subset architecture, we will add two registers:
 - EPC: a 32-bit register to hold the user's PC
 - Cause: A register to record the cause of the exception
 - we'll assume undefined inst = 0, overflow = 1
- We will also add three control signals:
 - EPCWrite (will need to be able to subtract 4 from PC)
 - CauseWrite
 - IntCause
- We will extend PCSource multiplexor to be able to latch the interrupt handler address into the PC.



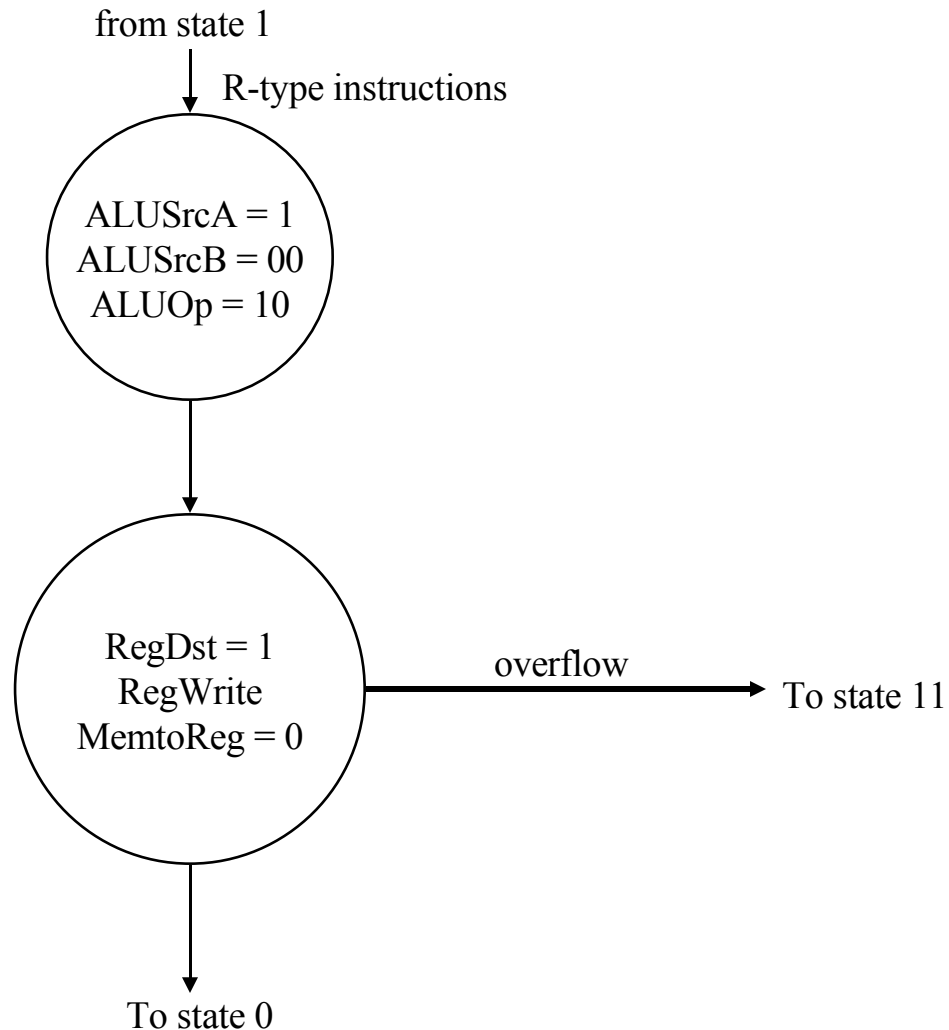
Supporting exceptions in our DataPath



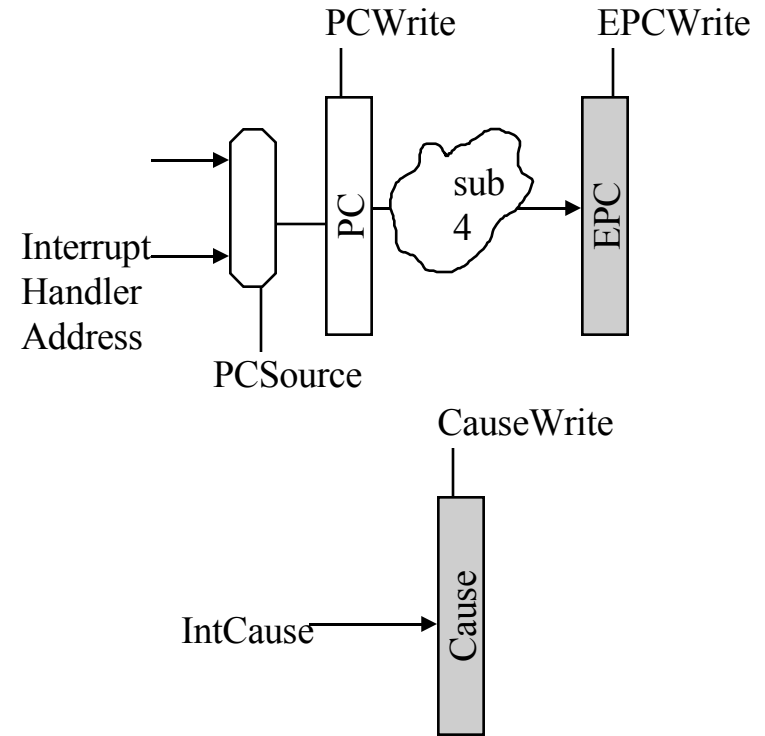
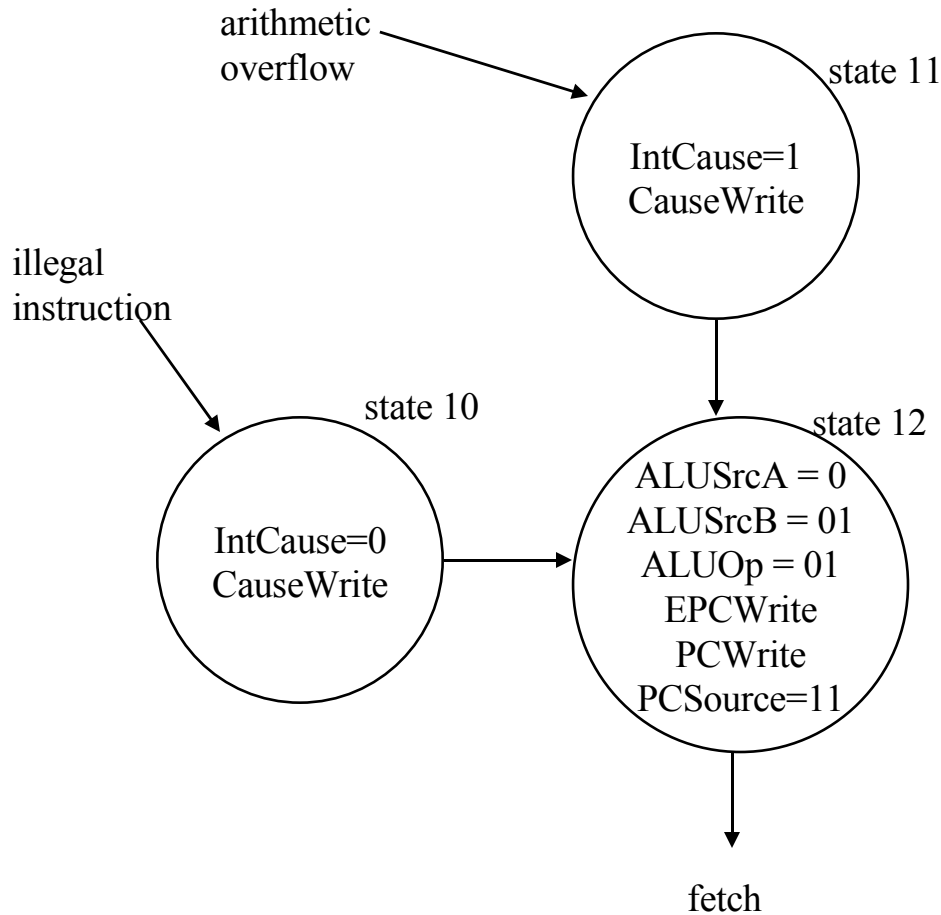
Supporting exceptions in our FSM



Supporting exceptions in our FSM.



Supporting exceptions in our FSM



Key Points

- microprogramming can simplify (conceptually) CPU control generation
- a microprogram is a small sequencer/processor inside the CPU that executes the individual instructions of the “real” program.
- Exception-handling is difficult in the CPU, because the interactions between the executing instructions and the interrupt are complex and sometimes unpredictable.