Historical Context:
Simplifying Control Design Through Microprogramming
Summary of First Multiple Cycle CPU Design

Instruction fetch

Decode and Register Fetch

Memory instructions
R-type instructions
Branch instructions
Jump instruction
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 ???

- ROM
- Address
- Opcode
- Next address calc
- Outputs
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

```plaintext
exception handler: read status register

overflow handler: ...

illegal inst handler: ...

I/O interrupt handler: ...

status register
```

user code
user code
user code
user code
user code
user code

...
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

Instruction Fetch, \textit{state 0}

- MemRead
- ALUSrcA = 0
- IorD = 0
- IRWrite
- ALUSrcB = 01
- ALUOp = 00
- PCWrite
- PCSource = 00

Instruction Decode/ Register Fetch, \textit{state 1}

- ALUSrcA = 0
- ALUSrcB = 11
- ALUOp = 00

- Opcode = LW or SW
- Opcode = R-type
- Opcode = BEQ
- Opcode = JMP
- Opcode = anything else to state 10

Flowchart:
- Start
- Instruction Fetch, state 0
- Instruction Decode/ Register Fetch, state 1
- Opcode
- Memory Inst FSM
- R-type Inst FSM
- Branch Inst FSM
- Jump Inst FSM
Supporting exceptions in our FSM.

from state 1

R-type instructions

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

RegDst = 1
RegWrite
MemtoReg = 0

overflow

To state 11

To state 0

CSE 141, S2'06  Jeff Brown
Supporting exceptions in our FSM

IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite

IntCause = 0
CauseWrite

ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11

illegal instruction

arithmetic overflow

fetch

EPCWrite
PCWrite
Interrupt Handler Address
PCSource
sub 4
EPC
CauseWrite
IntCause

CSE 141, S2'06
Jeff Brown
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.