Summary of First Multiple Cycle CPU Design

Shift left 2

PC
MUX
0
1

Registers
Write
register
Write data
Read data 1
Read data 2
Read register 1
Read register 2

Instruction[15–11]
MUX
0
1
MUX
0
1

Instruction[15–0]

Instruction[25–21]
Instruction[20–16]
Instruction[15–0]

Instruction
register

ALU
control

ALU
result

ALU
Zero

Memory
data

register

A
B

IorD

MemRead
MemWrite
MemtoReg
PCWriteCond
PCWrite
IRWrite

ALUOp

ALUSrcB
ALUSrcA

RegDst

PCSource

RegWrite

Control

Outputs

Op

[5–0]

Instruction

31-26

Instruction [5–0]

Jump
address [31-0]

Instruction [25–0]

0
2

2

1

Shift left 2

PC [31-28]

1

M

MUX
0
1

M

MUX
0
1

ALUOut

Memory
MemData
Write

data

Address

Historical Context:
Simplifying Control Design Through
Microprogramming

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

Control Logic

Inputs

Opcode
State Reg

Implementing a control FSM with ROM

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

ROM

Address

Outputs

Implementing a control FSM with ???

ROM

Address

Next address calc

Opcode
State Reg

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).

µ-program in ROM

Address

Outputs

µPC + next µPC logic

Opcode
Microprogram Implementation

Microprogram Implementation

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.

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

Instruction Fetch, state 0
Instruction Decode/ Register Fetch, state 1

Memory Inst FSM
R-type Inst FSM
Branch Inst FSM
Jump Inst 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

Illegal instruction

Interrupt Handler Address

PCWrite

EPCWrite

IntCause=1
CauseWrite

state 11

IntCause=0
CauseWrite

state 12

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

fetch
Key Points

• microprogramming can simplify (conceptually) CPU control generation
• a microprogram is a small program 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 unpredictable.