Overview of Control

- 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.

<table>
<thead>
<tr>
<th>Initial Representation</th>
<th>Finite State Diagram</th>
<th>Microprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequencing Control</td>
<td>Explicit Next State Function</td>
<td>Microprogram counter + Dispatch ROMs</td>
</tr>
<tr>
<td>Logic Representation</td>
<td>Logic Equations</td>
<td>Truth Tables</td>
</tr>
<tr>
<td>Implementation Technique</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recap: “Macroinstruction” Interpretation

- User program plus Data
- this can change!
- one of these is mapped into one of these

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer
  - Processor
    - Control
  - Datapath
  - Memory
  - Input
  - Output

- Today’s Topics:
  - Microprogrammed control
  - Administrivia; Courses
  - Microprogram it yourself
  - Exceptions
  - Intro to Pipelining (if time permits)
Recap: Horizontal vs. Vertical Microprogramming

NOTE: previous organization is not TRUE horizontal microprogramming; register decoders give flavor of encoded microoperations

Most microprogramming-based controllers vary between:
- **horizontal** organization (1 control bit per control point)
- **vertical** organization (fields encoded in the control memory and must be decoded to control something)

<table>
<thead>
<tr>
<th>Horizontal</th>
<th>Vertical</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ more control over the potential parallelism of operations in the datapath</td>
<td>+ easier to program, not very different from programming a RISC machine in assembly language</td>
</tr>
<tr>
<td>- uses up lots of control store</td>
<td>- extra level of decoding may slow the machine down</td>
</tr>
</tbody>
</table>

Recap: Designing a Microinstruction Set

1) Start with list of control signals
2) Group signals together that make sense (vs. random): called “fields”
3) Places fields in some logical order (e.g., 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
   - Use computers to design computers
5) To minimize the width, encode operations that will never be used at the same time

Alternative datapath (book): Multiple Cycle Datapath

° Minimizes Hardware: 1 memory, 1 adder

Finite State Machine (FSM) Spec

Q: How improve to do something in state 0001?
### Finite State Machine (FSM) Spec

**IR <= MEM[PC]**

```
0000
```

- **Instruction Fetch**

**PC <= PC + 4**

```
0001
```

- **Decode**

**R-type**

```
0100
```

- **ALUout <= A + SX**

```
1000
```

- **ALUout <= A or ZX**

```
1001
```

- **ALUout <= AfunB**

```
1010
```

- **PC <= A << B**

```
1100
```

**R[rd] <= ALUout**

```
0101
```

**R[rt] <= ALUout**

```
0111
```

**M <= MEM[ALUout]**

```
1011
```

**LW**

```
1101
```

**SW**

```
1111
```

**Execute**

**Write-back**

#### 1&2) Start with list of control signals, grouped into fields

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted</th>
<th>Effect when asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>Reg. is written</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Reg. write data input = ALU</td>
<td>Reg. write data input = memory</td>
</tr>
<tr>
<td>RegDest</td>
<td>Reg. dest. no. = rt</td>
<td>Reg. dest. no. = rd</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Memory at address is read,</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Memory at address is written,</td>
</tr>
<tr>
<td>IorD</td>
<td>Memory address = PC</td>
<td>Memory address = S</td>
</tr>
<tr>
<td>IRWrite</td>
<td>None</td>
<td>IR &lt;= Memory</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>PC &lt;= PCSource</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>IF ALUzero then PC &lt;= PCSource</td>
</tr>
<tr>
<td>PCSource</td>
<td>PCSource = ALU</td>
<td>PCSource = ALUout</td>
</tr>
<tr>
<td>ExtOp</td>
<td>Zero Extended</td>
<td>Sign Extended</td>
</tr>
</tbody>
</table>

#### Multiple Bit Control

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ALUOp</td>
<td>0</td>
<td>ALU adds</td>
</tr>
<tr>
<td>01</td>
<td>ALUOp</td>
<td>1</td>
<td>ALU subtracts</td>
</tr>
<tr>
<td>10</td>
<td>ALUOp</td>
<td>10</td>
<td>ALU does function code</td>
</tr>
<tr>
<td>11</td>
<td>ALUOp</td>
<td>11</td>
<td>ALU does logical OR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ALUSelB</td>
<td>00</td>
<td>2nd ALU input = 4</td>
</tr>
<tr>
<td>01</td>
<td>ALUSelB</td>
<td>01</td>
<td>2nd ALU input = Reg[rt]</td>
</tr>
<tr>
<td>10</td>
<td>ALUSelB</td>
<td>10</td>
<td>2nd ALU input = extended,shift left 2</td>
</tr>
<tr>
<td>11</td>
<td>ALUSelB</td>
<td>11</td>
<td>2nd ALU input = extended</td>
</tr>
</tbody>
</table>

**4) Legend of Fields and Symbolic Names**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field with Specific Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Add</td>
<td>ALU adds</td>
</tr>
<tr>
<td></td>
<td>Subt.</td>
<td>ALU subtracts</td>
</tr>
<tr>
<td></td>
<td>Func code</td>
<td>ALU does function code</td>
</tr>
<tr>
<td></td>
<td>Or</td>
<td>ALU does logical OR</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>1st ALU input = PC</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>1st ALU input = Reg[rs]</td>
</tr>
<tr>
<td>SRC2</td>
<td>4</td>
<td>2nd ALU input = 4</td>
</tr>
<tr>
<td></td>
<td>Extend</td>
<td>2nd ALU input = sign ext. IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>Extend0</td>
<td>2nd ALU input = zero ext. IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>Extshft</td>
<td>2nd ALU input = sign ex., sl IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>rt</td>
<td>2nd ALU input = Reg[rt]</td>
</tr>
<tr>
<td>destination</td>
<td>rd ALU</td>
<td>Reg[rd] = ALUout</td>
</tr>
<tr>
<td></td>
<td>rt ALU</td>
<td>Reg[rt] = ALUout</td>
</tr>
<tr>
<td></td>
<td>rt Mem</td>
<td>Reg[rt] = Mem</td>
</tr>
<tr>
<td>Memory</td>
<td>Read PC</td>
<td>Read memory using PC</td>
</tr>
<tr>
<td></td>
<td>Read ALU</td>
<td>Read memory using ALUout for addr</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>Write memory using ALUout for addr</td>
</tr>
<tr>
<td>Memory register</td>
<td>IR</td>
<td>IR = Mem</td>
</tr>
<tr>
<td>PC write</td>
<td>&lt;= ALU</td>
<td>PC = ALU</td>
</tr>
<tr>
<td>Seq</td>
<td>IF ALUZero then PC = ALUout</td>
<td></td>
</tr>
<tr>
<td>Fetch</td>
<td>Go to the first microinstruction</td>
<td></td>
</tr>
<tr>
<td>Dispatch</td>
<td>Dispatch using ROM.</td>
<td></td>
</tr>
</tbody>
</table>
Alternative datapath (book): Multiple Cycle Datapath

- Minimizes Hardware: 1 memory, 1 adder

![Datapath Diagram]

3) Microinstruction Format: unencoded vs. encoded fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Width</th>
<th>Control Signals Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU Control</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>SRC1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SRC2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>ALU Destination</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Memory</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Memory Register</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PCWrite Control</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Sequencing</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Total width</td>
<td>24</td>
<td>15</td>
</tr>
</tbody>
</table>

Start with list of control signals, cont’d

- For next state function (next microinstruction address), use Sequencer-based control unit from last lecture
  - Called “microPC” or “μPC” vs. state register

<table>
<thead>
<tr>
<th>Signal Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Next μaddress = 0</td>
</tr>
<tr>
<td>01</td>
<td>μaddress = dispatch ROM</td>
</tr>
<tr>
<td>10</td>
<td>μaddress = μaddress + 1</td>
</tr>
</tbody>
</table>

- Could even include “branch” option which changes microPC by adding offset when certain control signals are true.

Microprogram it yourself!

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>ALU Dest.</th>
<th>Memory</th>
<th>Mem. Reg. PC Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
</tr>
</tbody>
</table>
Microprogram it yourself!

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Dest.</th>
<th>Memory</th>
<th>Mem. Reg.</th>
<th>PC</th>
<th>Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch:</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Extshft</td>
<td>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
<td>Dispatch</td>
</tr>
<tr>
<td>Rtype:</td>
<td>Func</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>rd ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ori:</td>
<td>Or</td>
<td>rs</td>
<td>Extend0</td>
<td>rt ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Read ALU</td>
<td>Seq</td>
<td>rt MEM</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Write ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beq:</td>
<td>Subt.</td>
<td>rs</td>
<td>rt</td>
<td>ALUOutCond.</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Administrivia

° Midterm I on Wednesday
  • 5:30 - 8:30 in 277 Cory
  • Bring a Calculator!
  • One 8 1/2 by 11 page (both sides) of notes
  • Materials through Section 5.5 (Microprogramming)

° Afterwards: Pizza and refreshments at LaVals

° Lab 4 breakdown due by 5pm tonight
  • EMail to your TA
  • Get moving on it! This is a complicated lab.

° Now, start reading Chapter 6

Administrivia: Courses to consider during Telebears

° General Philosophy
  • Take courses from great teachers (HKN ratings helps find them)
    - http://www-hkn.eecs.berkeley.edu/toplevel/coursesurveys.html
  • Take variety of undergrad courses now to get introduction to areas; can learn advanced material on own later once know vocabulary
  • Who knows what you will work on over a 40 year career?

° CS169 Software Engineering
  • Everyone writes programs, even hardware designers
  • Often programs are written in groups ⇒ learn skill in school

° EE122 Introduction to Communication Networks
  • World is getting connected; communications must play major role

° CS162 Operating Systems
  • All special-purpose hardware will run a layer of software that uses processes and concurrent programming; CS162 is the closest thing

Lab4: start using test benches

° Idea: wrap testing infrastructure around devices under test.

° Include test vectors that are supposed to detect errors in implementation. Even strange ones...

° Can (and probably should in later labs) include assert statements to check for “things that should never happen”
**An Alternative MultiCycle DataPath**

- In each clock cycle, each Bus can be used to transfer from one source
- *μ*-instruction can simply contain B-Bus and W-Dst fields

**What about a 2-Bus Microarchitecture (datapath)?**

- Instruction Fetch
- Decode / Operand Fetch

**Legacy Software and Microprogramming**

- IBM bet company on 360 Instruction Set Architecture (ISA): single instruction set for many classes of machines
  - (8-bit to 64-bit)
- Stewart Tucker stuck with job of what to do about software compatibility
  - If microprogramming could easily do same instruction set on many different microarchitectures, then why couldn’t multiple microprograms do multiple instruction sets on the same microarchitecture?
  - Coined term “emulation”: instruction set interpreter in microcode for non-native instruction set
  - Very successful: in early years of IBM 360 it was hard to know whether old instruction set or new instruction set was more frequently used
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 control memory)
° Generality
  • Can implement multiple instruction sets on same machine.
  • Can tailor instruction set to application.
° Compatibility
  • Many organizations, same instruction set
° Costly to implement
° Slow

Exceptions

° Exception = unprogrammed control transfer
  • system takes action to handle the exception
    - must record the address of the offending instruction
    - record any other information necessary to return afterwards
  • returns control to user
    • must save & restore user state
° Allows construction of a “user virtual machine”

Two Types of Exceptions: Interrupts and Traps

° Interrupts
  • caused by external events:
    - Network, Keyboard, Disk I/O, Timer
  • asynchronous to program execution
    - Most interrupts can be disabled for brief periods of time
    - Some (like “Power Failing”) are non-maskable (NMI)
  • may be handled between instructions
  • simply suspend and resume user program

° Traps
  • caused by internal events
    - exceptional conditions (overflow)
    - errors (parity)
    - faults (non-resident page)
  • synchronous to program execution
  • condition must be remedied by the handler
  • instruction may be retried or simulated and program continued or program may be aborted

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.

<table>
<thead>
<tr>
<th>Type of event</th>
<th>From where?</th>
<th>MIPS terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>External</td>
<td>Interrupt</td>
</tr>
<tr>
<td>Invoke OS from user program</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Arithmetic overflow</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Using an undefined instruction</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Hardware malfunctions</td>
<td>Either</td>
<td>Exception or Interrupt</td>
</tr>
</tbody>
</table>
What happens to Instruction with Exception?

- MIPS architecture defines the instruction as having **no effect** if the instruction causes an exception.
- When get to virtual memory we will see that certain classes of exceptions must prevent the instruction from changing the machine state.
- This aspect of handling exceptions becomes complex and potentially limits performance \( \Rightarrow \) why it is hard

Precise Interrupts

- **Precise** \( \Rightarrow \) state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions **completed**
  - Offending instruction and all following instructions act **as if they have not even started**
  - Same system code will work on different implementations
  - Position clearly established by IBM
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position
- **Imprecise** \( \Rightarrow \) system software has to figure out what is where and put it all back together
- Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Big Picture: user / system modes

- By providing two modes of execution (user/system) it is possible for the computer to manage itself
  - operating system is a special program that runs in the privileged mode and has access to all of the resources of the computer
  - presents “virtual resources” to each user that are more convenient than the physical resources
    - files vs. disk sectors
    - virtual memory vs physical memory
  - protects each user program from others
  - protects system from malicious users.
  - OS is assumed to “know best”, and is trusted code, so enter system mode on exception.
- Exceptions allow the system to taken action in response to events that occur while user program is executing:
  - Might provide supplemental behavior (dealing with denormal floating-point numbers for instance).
  - “Unimplemented instruction” used to emulate instructions that were not included in hardware (i.e. MicroVax)

Addressing the Exception Handler

- **Traditional Approach:** Interrupt Vector
  - PC <- MEM[ IV_base + cause || 00]
  - 370, 68000, Vax, 80x86, ...
- **RISC Handler Table**
  - PC <- IT_base + cause || 0000
  - saves state and jumps
  - Sparc, PA, M68K, ...
- **MIPS Approach:** fixed entry
  - PC <- EXC_addr
  - Actually very small table
    - RESET entry
    - TLB
    - other
Saving 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 of the internal pipeline registers

---

Additions to MIPS ISA to support Exceptions?

- **Exception state is kept in “coprocessor 0”**.

- **EPC** – a 32-bit register used to hold the address of the affected instruction (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 encodes the two possible exception sources mentioned above: undefined instruction=0 and arithmetic overflow=1 (register 13 of coprocessor 0).

- **BadVAddr** – register contained 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**

- **EPC**

  - a 32-bit register used to hold the address of the affected instruction (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 encodes the two possible exception sources mentioned above: undefined instruction=0 and arithmetic overflow=1 (register 13 of coprocessor 0).

- **BadVAddr**

  - register contained 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**

- **EPC**

  - a 32-bit register used to hold the address of the affected instruction (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 encodes the two possible exception sources mentioned above: undefined instruction=0 and arithmetic overflow=1 (register 13 of coprocessor 0).

- **BadVAddr**

  - register contained 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**

- **EPC**

  - a 32-bit register used to hold the address of the affected instruction (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 encodes the two possible exception sources mentioned above: undefined instruction=0 and arithmetic overflow=1 (register 13 of coprocessor 0).

- **BadVAddr**

  - register contained 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**

---

Recap: Details of Status register

| Status | 15 8 5 4 3 2 1 0 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Mask | k e k e k e | old | prev | current |

- **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

- When interrupt occurs, 6 LSB shifted left 2 bits, setting 2 LSB to 0
  - run in kernel mode with interrupts disabled

---

Recap: Details of Cause register

| Status | 15 10 5 2 |
| --- | --- | --- | --- |
| Pending | Code |

- **Pending interrupt** 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

- **Exception Code** encodes reasons for interrupt
  - 0 (INT) => external interrupt
  - 4 (ADDRL) => address error exception (load or instr fetch)
  - 5 (ADDRS) => address error exception (store)
  - 6 (IBUS) => bus error on instruction fetch
  - 7 (DBUS) => bus error on data fetch
  - 8 (Syscall) => Syscall exception
  - 9 (BKPT) => Breakpoint exception
  - 10 (RI) => Reserved Instruction exception
  - 12 (OVF) => Arithmetic overflow exception
How Control Detects Exceptions in our FSD

- 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.
  - 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 4 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.

- 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.

How add Exceptions for Overflow and Unimplemented?

Pipelining is Natural!

- Laundry Example
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

Sequential Laundry

- Sequential laundry takes 6 hours for 4 loads
  - If they learned pipelining, how long would laundry take?
**Pipelined Laundry: Start work ASAP**

Pipelined laundry takes 3.5 hours for 4 loads

- Pipelined laundry takes 3.5 hours for 4 loads

**Pipelining Lessons**

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced speedup = Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

**Single Cycle, Multiple Cycle, vs. Pipeline**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Cycle Implementation:</td>
<td>Multiple Cycle Implementation:</td>
</tr>
<tr>
<td>Load</td>
<td>Store</td>
</tr>
<tr>
<td>Cycle 1</td>
<td>Cycle 2</td>
</tr>
<tr>
<td>Clk</td>
<td>Clk</td>
</tr>
</tbody>
</table>

**Pipelined Execution**

- **Utilization?**
- **Now we just have to make it work**
Why Pipeline?

- Suppose we execute 100 instructions
  - Single Cycle Machine
    - 45 ns/cycle \times 1 \text{ CPI} \times 100 \text{ inst} = 4500 \text{ ns}
  - Multicycle Machine
    - 10 ns/cycle \times 4.6 \text{ CPI (due to inst mix)} \times 100 \text{ inst} = 4600 \text{ ns}
  - Ideal pipelined machine
    - 10 ns/cycle \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1040 \text{ ns}

Why Pipeline? Because the resources are there!

Can pipelining get us into trouble?

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - data hazards: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can’t fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline
  - control hazards: attempt to make a decision before condition is evaluated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
    - branch instructions

  - Can always resolve hazards by waiting
    - pipeline control must detect the hazard
    - take action (or delay action) to resolve hazards

Summary 1/3

- Specialize state-diagrams easily captured by microsequencer
  - simple increment & “branch” fields
  - datapath control fields
- Control design reduces to Microprogramming
- 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
Summary 2/3

- Microprogramming is a fundamental concept
  - implement an instruction set by building a very simple processor and interpreting the instructions
  - essential for very complex instructions and when few register transfers are possible

- Pipelining is a fundamental concept
  - multiple steps using distinct resources

- Utilize capabilities of the Datapath by pipelined instruction processing
  - start next instruction while working on the current one
  - limited by length of longest stage (plus fill/flush)
  - detect and resolve hazards

Summary: Microprogramming one inspiration for RISC

- If simple instruction could execute at very high clock rate...
- If you could even write compilers to produce microinstructions...
- If most programs use simple instructions and addressing modes...
- If microcode is kept in RAM instead of ROM so as to fix bugs ...
- If same memory used for control memory could be used instead as cache for “macroinstructions”...
- Then why not skip instruction interpretation by a microprogram and simply compile directly into lowest language of machine? (microprogramming is overkill when ISA matches datapath 1-1)