Recap

° Partition datapath into equal size chunks to minimize cycle time
  • ~10 levels of logic between latches

° Follow same 5-step method for designing “real” processor

° Control is specified by finite state digram
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>PLA</td>
<td>ROM</td>
</tr>
</tbody>
</table>

“hardwired control”

“microprogrammed control”
Recap: Controller Design

° The state digrams that arise define the controller for an instruction set processor are highly structured

° Use this structure to construct a simple “microsequencer”

° Control reduces to programming this very simple device
  • microprogramming
## Recap: Microprogram Control Specification

<table>
<thead>
<tr>
<th>µPC</th>
<th>Taken</th>
<th>Next</th>
<th>IR</th>
<th>PC en sel</th>
<th>Ops A B</th>
<th>Exec Ex Sr ALU S</th>
<th>Mem R W M</th>
<th>Write-Back M-R Wr Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>?</td>
<td>inc 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>load</td>
<td>inc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>x</td>
<td>zero</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>x</td>
<td>zero</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>x</td>
<td>inc</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1 fun</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>x</td>
<td>zero</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>x</td>
<td>inc</td>
<td></td>
<td>0</td>
<td>0 0 or 1</td>
<td>0</td>
<td>0 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0111</td>
<td>x</td>
<td>zero</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 1 0</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>x</td>
<td>inc</td>
<td></td>
<td>1</td>
<td>add 1</td>
<td>1</td>
<td>0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1001</td>
<td>x</td>
<td>inc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>x</td>
<td>zero</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1011</td>
<td>x</td>
<td>inc</td>
<td></td>
<td>1</td>
<td>add 1</td>
<td>1</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>x</td>
<td>zero</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0 1</td>
</tr>
</tbody>
</table>

### Instructions
- **BEQ**: Branch if equal
- **R**: Register
- **ORi**: Immediate operand
- **LW**: Load from memory
- **SW**: Store to memory
The Big Picture: Where are We Now?

° The Five Classic Components of a Computer

- Processor
- Control
- Datapath
- Memory
- Input
- Output

° Today’s Topics:
  - Microprogramed control
  - Administrivia
  - Microprogram it yourself
  - Exceptions
How Effectively are we utilizing our hardware?

- Example: memory is used twice, at different times
  - Ave mem access per inst = 1 + Flw + Fsw ~ 1.3
  - if CPI is 4.8, imem utilization = 1/4.8, dmem = 0.3/4.8

- We could reduce HW without hurting performance
  - extra control
Single memory for instruction and data access
  - memory utilization -> 1.3/4.8

In this case our state diagram does not change
  - several additional control signals
  - must ensure each bus is only driven by one source on each cycle
Alternative datapath (book): Multiple Cycle Datapath

- Minimizes Hardware: 1 memory, 1 adder
Our Controller FSM Spec

IR <= MEM[PC]  
PC <= PC + 4  

0000  

“A instruction fetch”

A <= R[rs]  
B <= R[rt]  

0001  

“decode”

R-type

S <= A fun B  
0100  

S <= A op ZX  
0110  

S <= A + SX  
1000  

S <= A + SX  
1011  

S <= A - B  
0010  

0011

“execute”

S <= A - B  
0010  

~Equal

PC <= PC +  SX || 00  
0000  

Write-back

Memory

R[rd] <= S  
0101  

R[rt] <= S  
0111  

R[rt] <= M  
1010  

MEM[S] <= B  
1100

0111

cs 152  L1 1 .10

DAP Fa97, © U.CB
Microprogramming

Control is the hard part of processor design
  - Datapath is fairly regular and well-organized
  - Memory is highly regular
  - Control is irregular and global

Microprogramming:
   -- A Particular Strategy for Implementing the Control Unit of a processor by "programming" at the level of register transfer operations

Microarchitecture:
   -- Logical structure and functional capabilities of the hardware as seen by the microprogrammer

Historical Note:

IBM 360 Series first to distinguish between architecture & organization
Same instruction set across wide range of implementations, each with different cost/performance
Sequencer-based control unit

Types of “branching”
- Set state to 0
- Dispatch (state 1)
- Use incremented state number

Adder

Address Select Logic

Opcode

State Reg

Inputs

Outputs

Control Logic

Multicycle Datapath
“Macroinstruction” Interpretation

Main Memory

execution unit

CPU

User program plus Data

ADD

SUB

AND

DATA

this can change!

one of these is mapped into one of these

AND microsequence

e.g., Fetch
Calc Operand Addr
Fetch Operand(s)
Calculate
Save Answer(s)
Variations on Microprogramming

° “Horizontal” Microcode
  – control field for each control point in the machine

<table>
<thead>
<tr>
<th>µseq</th>
<th>µaddr</th>
<th>A-mux</th>
<th>B-mux</th>
<th>bus enables</th>
<th>register enables</th>
</tr>
</thead>
</table>

° “Vertical” Microcode
  – compact microinstruction format for each class of microoperation
  – local decode to generate all control points

branch: µseq-op µadd
execute: ALU-op A,B,R
memory: mem-op S, D
Depending on bus organization, many potential control combinations simply wrong, i.e., implies transfers that can never happen at the same time.

Makes sense to encode fields to save ROM space

Example: mem_to_reg and ALU_to_reg should never happen simultaneously;
=> encode in single bit which is decoded rather than two separate bits

NOTE: encoding should be just sufficient that parallel actions that the datapath supports should still be specifiable in a single microinstruction
More Vertical Format

Some of these may have nothing to do with registers!

Multiformat Microcode:

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cond</td>
<td>next address</td>
</tr>
</tbody>
</table>

Branch Jump

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dst</td>
<td>src</td>
<td>alu</td>
</tr>
</tbody>
</table>

Register Xfer Operation
Hybrid Control

Not all critical control information is derived from control logic

E.g., Instruction Register (IR) contains useful control information, such as register sources, destinations, opcodes, etc.
Vax Microinstructions

VAX Microarchitecture:

96 bit control store, 30 fields, 4096 μinstructions for VAX ISA encodes concurrently executable "microoperations"

<table>
<thead>
<tr>
<th>95</th>
<th>87</th>
<th>84</th>
<th>68</th>
<th>65</th>
<th>63</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>USHF</td>
<td>UALU</td>
<td>USUB</td>
<td></td>
<td></td>
<td></td>
<td>UJMP</td>
</tr>
</tbody>
</table>

- 001 = left
- 010 = right
- 100 = A+B+1
- 101 = left3
- 010 = A-B-1
- 00 = Nop
- 01 = CALL
- 10 = RTN

ALU Shifter Control

ALU Control

Subroutine Control

Jump Address
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)

**Horizontal**

+ more control over the potential parallelism of operations in the datapath

- uses up lots of control store

**Vertical**

+ easier to program, not very different from programming a RISC machine in assembly language

- extra level of decoding may slow the machine down
Administration

° Schedule change: Delay Lab 4 until Tuesday after midterm (10/14)?
  • => Delay Lab 5 until 10/28
         => Delay Lab 6 until 11/11
         => Delay Midterm II until 11/18

° Midterm Wed 10/8 at 5:30 in 306 Soda; no lecture that day

° Review Session 1-3 Sunday 10/5 in 306 Soda
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
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>ALUSelA</td>
<td>1st ALU operand = PC</td>
<td>1st ALU operand = Reg[rs]</td>
</tr>
<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>RegDst</td>
<td>Reg. dest. no. = rt</td>
<td>Reg. dest. no. = rd</td>
</tr>
<tr>
<td>TargetWrite</td>
<td>None</td>
<td>Target reg. = ALU</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 = ALU</td>
</tr>
<tr>
<td>IRWrite</td>
<td>None</td>
<td>IR = Memory</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>PC = PCSource</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>IF ALUzero then PC = PCSource</td>
</tr>
</tbody>
</table>

### Single Bit Control

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp</td>
<td>00</td>
<td>ALU adds</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>ALU subtracts</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>ALU does function code</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>ALU does logical OR</td>
</tr>
<tr>
<td>ALUSelB</td>
<td>000</td>
<td>2nd ALU input = Reg[rt]</td>
</tr>
<tr>
<td></td>
<td>001</td>
<td>2nd ALU input = 4</td>
</tr>
<tr>
<td></td>
<td>010</td>
<td>2nd ALU input = sign extended IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>011</td>
<td>2nd ALU input = sign extended, shift left 2 IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2nd ALU input = zero extended IR[15-0]</td>
</tr>
<tr>
<td>PCSOURCE</td>
<td>00</td>
<td>PC = ALU</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>PC = Target</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>PC = PC+4[29-26] : IR[25–0] &lt;&lt; 2</td>
</tr>
</tbody>
</table>
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</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequen</td>
<td>00</td>
<td>Next µaddress = 0</td>
</tr>
<tr>
<td>-cing</td>
<td>01</td>
<td>Next µaddress = dispatch ROM</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Next µaddress = µaddress + 1</td>
</tr>
</tbody>
</table>

starting µaddress

1 1 0

µAddress Select Logic

1 0

µAddress

Adder

1 0

microPC

Mux

2 1 0

Opcode

ROM
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></td>
<td>wide</td>
<td>narrow</td>
</tr>
<tr>
<td>ALU Control</td>
<td>4</td>
<td>2 ALUOp</td>
</tr>
<tr>
<td>SRC1</td>
<td>2</td>
<td>1 ALUSelA</td>
</tr>
<tr>
<td>SRC2</td>
<td>5</td>
<td>3 ALUSelB</td>
</tr>
<tr>
<td>ALU Destination</td>
<td>6</td>
<td>4 RegWrite, MemtoReg, RegDst, TargetWr.</td>
</tr>
<tr>
<td>Memory</td>
<td>4</td>
<td>3 MemRead, MemWrite, IorD</td>
</tr>
<tr>
<td>Memory Register</td>
<td>1</td>
<td>1 IRWrite</td>
</tr>
<tr>
<td>PCWrite Control</td>
<td>5</td>
<td>4 PCWrite, PCWriteCond, PCSource</td>
</tr>
<tr>
<td>Sequencing</td>
<td>3</td>
<td>2 AddrCtl</td>
</tr>
<tr>
<td>Total width</td>
<td>30</td>
<td>20 bits</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>ALU destination</td>
<td>Target</td>
<td>Target = ALUout</td>
</tr>
<tr>
<td></td>
<td>rd</td>
<td>Reg[rd] = ALUout</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 ALU output</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>Write memory using ALU output</td>
</tr>
<tr>
<td>Memory register</td>
<td>IR</td>
<td>IR = Mem</td>
</tr>
<tr>
<td></td>
<td>Write rt</td>
<td>Reg[rt] = Mem</td>
</tr>
<tr>
<td></td>
<td>Read rt</td>
<td>Mem = Reg[rt]</td>
</tr>
<tr>
<td>PC write</td>
<td>ALU</td>
<td>PC = ALU output</td>
</tr>
<tr>
<td></td>
<td>Target-cond.</td>
<td>IF ALU Zero then PC = Target</td>
</tr>
<tr>
<td></td>
<td>jump addr.</td>
<td>PC = PCSource</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>Go to sequential µinstruction</td>
</tr>
<tr>
<td></td>
<td>Fetch</td>
<td>Go to the first microinstruction</td>
</tr>
<tr>
<td></td>
<td>Dispatch</td>
<td>Dispatch using ROM</td>
</tr>
</tbody>
</table>
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>

DAP Fa97, © U.CB
**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</th>
<th>Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Add</td>
<td>PC</td>
<td>PC</td>
<td>4 Extshft Target</td>
<td>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
</tr>
<tr>
<td>LW</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Read ALU</td>
<td>Write rt</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Write ALU</td>
<td>Read rt</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>Rtype</td>
<td>Func</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ1</td>
<td>Subt.</td>
<td>rs</td>
<td>rt</td>
<td>Target– cond.</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td>jump address</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORI</td>
<td>Or</td>
<td>rs</td>
<td>Extend0</td>
<td>rd</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
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)?
What about 1 bus? 1 adder? 1 Register port?
Exceptions

Exception = unprogrammed control transfer
- system takes action to handle the exception
  - must record the address of the offending instruction
- returns control to user
- must save & restore user state

° Allows construction of a “user virtual machine”

normal control flow:
- sequential, jumps, branches, calls, returns

user program

Exception:

System Exception Handler

return from exception
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 => why it is hard
Two Types of Exceptions

° **Interrupts**
  - caused by external events
  - asynchronous to program execution
  - 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>
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, M88K, . . .

° 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?

° **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

° Be able to write exception address into PC, increase mux to add as input $01000000\ 00000000\ 00000000\ 01000000_{two}$ ($8000\ 0080_{hex}$)

° May have to undo $PC = PC + 4$, since want EPC to point to offending instruction (not its successor); $PC = PC - 4$
Recap: Details of Status register

<table>
<thead>
<tr>
<th>Status</th>
<th>Mask</th>
<th>k</th>
<th>e</th>
<th>k</th>
<th>e</th>
<th>k</th>
<th>e</th>
<th>k</th>
<th>e</th>
<th>k</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>k</td>
<td>e</td>
<td>k</td>
<td>e</td>
<td>k</td>
<td>e</td>
<td>k</td>
<td>e</td>
<td>k</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>old</td>
<td>prev</td>
<td>current</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

° **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
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 that the physical resources
    - files vs. disk sectors
    - virtual memory vs physical memory
  • protects each user program from others

° Exceptions allow the system to taken action in response to events that occur while user program is executing
  • O/S begins at the handler
Recap: Details of Cause register

- **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

<table>
<thead>
<tr>
<th>Status</th>
<th>15</th>
<th>10</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pending</td>
<td>Code</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DAP Fa97, © U.CB
Precise Interrupts

- Precise => state of the machine is preserved as if program executed up to the offending instruction
  - Same system code will work on different implementations of the architecture
  - Position clearly established by IBM
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position

- Imprecise => 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
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 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 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.
Modification to the Control Specification

IR <= MEM[PC]
PC <= PC + 4

A <= R[rs]
B <= R[rt]

R[rd] <= S

EPC <= PC - 4
PC <= exp_addr
cause <= 12 (Ovf)

unknown instruction

EPC <= PC - 4
PC <= exp_addr
cause <= 10 (RI)

undefined instruction

S <= A fun B

S <= A op ZX

S <= A + SX

S <= A + SX

S <= A - B

~Equal

BEQ

M <= MEM[S]

MEM[S] <= B

PC <= PC + SX || 00

0011

0010

overflow

Equal

R[rt] <= S

R[rt] <= S

R[rt] <= M

Additional condition from Datapath
Summary

- 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: 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?