Review: High Level Design

- **Design Process**
  - Design Entry: Schematics, HDL, Compilers
  - High Level Analysis: Simulation, Testing, Assertions
  - Technology Mapping: Turn design into physical implementation
  - Low Level Analysis: Check out Timing, Setup/Hold, etc

- **Verilog – Three programming styles**
  - Structural: Like a Nettlist
    - Instantiation of modules + wires between them
  - Dataflow: Higher Level
    - Expressions instead of gates
  - Behavioral: Hardware programming
    - Full flow-control mechanisms
    - Registers, variables
    - File I/O, console display, etc

Review: Testing: Make sure that things work

- **Testing methodologies**
  - Understand what correct behavior is when you design things
    - Collect vectors for later use
  - Build monitor modules to check assertions of correct values
  - Produce a regression test
    - Set of tests to run each time something changes

- **Types of test (Doug Clark):**
  - Directed Vectors – test explicit behavior
  - Random Vectors – apply random values or orderings to device
  - Daemons – continuous error insertion

- **Monitor modules:**
  - Check to see if invariants are maintained during long running simulations

Review: Monitor Modules: Passthrough testing

```
module monitorms32(carry,sum,A,B);
  input [31:0] A,B;
  output [31:0] sum;
  output carry;
  reg [31:0] predsum;
  reg precarry;
  // The "real" adder
  sum32 mysum (carry,sum,A,B);
  // Ifndef synthesis // This checker code only for simulation
  always @(A or B)
    begin
      #100 //wait for output to settle (don’t make too long!)
      predsum = A + B;
      if ((carry != precarry) || (sum != predsum))
        $display(">>> Mismatch: 0x%x+0x%x->0x%x carry %x", A,B,sum,carry);
    end
  endmodule
```

Alewife Numbers

- Initial set of "nifty" bugs
- John Kubiatowicz at ISCA ’92 (Debugging provides to a half)
Lab4 version: monitors and benches

- Idea: wrap testing infrastructure around devices under test (DUT)
- 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”

Test Bench

Complete Top-Level Design

Inline Monitor

Output in readable format (disassembly)

Incomplete Top-Level Design

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

Today’s Topics:

- Microprogrammed control
- Administrivia; Courses
- Microprogram it yourself
- Exceptions
- Intro to Pipelining (if time permits)

Alternative multicycle datapath (book)

- Minimizes Hardware: 1 memory, 1 adder

New Finite State Machine (FSM) Spec

IR <= MEM[PC]
PC <= PC + 4
R-type
ALUout <= A fun B
M <= MEM[ALUout]
R[r] <= M
1100

“instruction fetch”
“decode”

If A = B then
PC <= ALUout
0001

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

```
IR <= MEM[PC] 0000
PC <= PC + 4

ALUout <= A + B
R[rd] <= ALUout

if A = B then
PC <= ALUout 0010

M <= MEM[ALUout] 1100
R[rt] <= M
M <= MEM[ALUout] 1001

if A = Z then
PC <= PC + 4
```

**Recap: Microprogramming**

- 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
  - overkill when ISA matches datapath 1-1

**Recap: “Macroinstruction” Interpretation**

- User program plus Data
  - ADD
  - SUB
  - AND
  - DATA
- one of these is mapped into one of these
- e.g., Fetch
  - Calc Operand Addr
  - Fetch Operand(s)
  - Calculate
  - Save Answer(s)
Designing a Microinstruction Set

1) Start with list of control signals

2) Group signals together that make sense (vs. random): called “fields”

3) Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)

4) To minimize the width, encode operations that will never be used at the same time

5) 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

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>MemRead</td>
<td>None</td>
<td>Memory at address is read, MDR &lt;= Mem[addr]</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</td>
<td>Sign Extended</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>00</td>
<td>2nd ALU input = 4</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>2nd ALU input = Reg[rt]</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2nd ALU input = extended, shift left 2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>2nd ALU input = extended</td>
</tr>
</tbody>
</table>

Multiple Bit Control

Again: Alternative multicycle datapath (book)

- Minimizes Hardware: 1 memory, 1 adder

3&4) 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 ALUOp</td>
</tr>
<tr>
<td>SRC1</td>
<td>5</td>
<td>3 ALUSelA</td>
</tr>
<tr>
<td>SRC2</td>
<td>5</td>
<td>3 ALUSelB, ExtOp</td>
</tr>
<tr>
<td>ALU Destination</td>
<td>5</td>
<td>3 RegWrite, MemtoReg, RegDst</td>
</tr>
<tr>
<td>Memory</td>
<td>3</td>
<td>2 MemRead, MemWrite, IorD</td>
</tr>
<tr>
<td>Memory Register</td>
<td>1</td>
<td>1 IRWrite</td>
</tr>
<tr>
<td>PCWrite Control</td>
<td>3</td>
<td>2 PCWrite, PCWriteCond, PCSource</td>
</tr>
<tr>
<td>Sequencing</td>
<td>3</td>
<td>2 AddrCtl</td>
</tr>
<tr>
<td>Total width</td>
<td>24</td>
<td>15 bits</td>
</tr>
</tbody>
</table>
5) 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>ExtHft</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>Mem</td>
<td>Reg[mem] = 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></td>
<td>ALUoutCond</td>
<td>IF ALU Zero then PC = ALUout</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>

destination: what do these fieldnames mean?

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>RegWrite</th>
<th>MemToReg</th>
<th>RegDest</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>---</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>rd ALU</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>rt ALU</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>rt MEM</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

SRC2:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>ALUSelB</th>
<th>ExtOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>---</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>001</td>
<td>4</td>
<td>00</td>
<td>X</td>
</tr>
<tr>
<td>010</td>
<td>rt</td>
<td>01</td>
<td>X</td>
</tr>
<tr>
<td>011</td>
<td>ExtHft</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>Extend</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>ExtendHft</td>
<td>11</td>
<td>0</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 Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch: Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
<td></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 Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch: Add</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>
</tr>
</tbody>
</table>

#### Rtype: Func
- rs | rt | rd ALU | Seq |
- Fetch

#### Lw: Add
- rs | Extend | Read ALU | Seq |
- rt MEM | Seq |
- Fetch

#### Sw: Add
- rs | Extend | Write ALU | Seq |
- rt ALU | Seq |
- Fetch

#### Ori: Or
- rs | Extend0 | Write ALU | Seq |
- rt ALU | Seq |
- Fetch

#### Beq: Subt.
- rs | rt | ALUoutCond. | Seq |
- Fetch

### 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
- A-Bus
- B Bus
- Next PC
- IR
- ZXSX
- Reg File
- Mem
- A Bus
- B Bus
- Next PC
- IR
- ZXSX
- Reg File
- Mem
- N
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

Summary
° 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
  • Control design reduces to Microprogramming
° Design of a Microprogramming language
  1. Start with list of control signals
  2. Group signals together that make sense (vs. random): called “fields”
  3. Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
  4. To minimize the width, encode operations that will never be used at the same time
  5. Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals
Thought: 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)