Simple CPU-core Example

• Why study CPU cores?
  1. Another large design example.
  2. More experience with RTL descriptions.
  3. A classic controller + Data-path type design example.
  5. Complements prior knowledge from CS61c of MIPS processor.

• This example:
  – Simple “8-bit” processor core with 7 instructions.
  – Just look at CPU-core, no memory or I/O design.
  – Made up just for EECS150 (pin the blame on Wawrynek)
  – Sufficiently simple so all details can be covered in class.
  – But, general enough to be useful for real programming. Could write and run real programs (assembly only) on it.

Lecture Outline

1. ISA description.
2. Implementation constraints and assumptions.
3. Draft micro-architecture.
4. RTL for each instruction.
5. Data-path refinement for each instruction.
7. High-level controller design.
8. Controller implementation.

Instruction Set Architecture (ISA)

The ISA is the abstraction that the hardware supports and provides to the software. It comprises a description of all the software visible registers, all the instructions, and the core interfaces.

• Interfaces:
  – CL K RST
  – DIN
  – DOUT
  – Addr
  – CPU
  – Memory Interface

• Instructions: Two formats
  – r-format
    – op1 ra rb
  – o-format
    – op1 ra rb

• Registers:
  – 4 8-bit general purpose registers (GPR).
  – R0 reads as all 0s.
  – ra, rb, rc are 2-bit GPR specifiers
  – r-format opcode is specified by op1
  – o-format opcode is specified by op1 and op2.

  – Program counter (PC) points to next instruction in memory. Resets to 0.
Instruction Set Architecture (ISA)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Assembly Language</th>
<th>Operation</th>
<th>op1</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add rc,ra,rb</td>
<td>rc ← ra+rb</td>
<td>00</td>
<td>-</td>
</tr>
<tr>
<td>subtract</td>
<td>sub rc,ra,rb</td>
<td>rc ← ra-rb</td>
<td>01</td>
<td>-</td>
</tr>
<tr>
<td>bit-wise nor</td>
<td>add rc,ra,rb</td>
<td>rc ← ra NOR rb</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>load byte</td>
<td>ldb rc,ra,offset</td>
<td>rc ← memory[ra+offset]</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>store byte</td>
<td>stb rc,ra,offset</td>
<td>memory[ra+offset]&lt;–rc</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>branch equal</td>
<td>beq rc,ra,offset</td>
<td>IF rc=ra pc&lt;–pc+1+offset</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Implementation Constraints and Assumptions

- Non-pipelined instruction execution.
  - Keeps things simple.
  - Take cs152 for details on processor pipelining.
- Multiple cycles per instruction.
  - Instructions will execute one at a time over several cycles.
  - Within the cycles used to execute each instruction, the next instruction will be fetched from memory.
  - The final step of each instruction execution will involve a transfer of control to the next instruction.
- Critical path is assumed to be both memory & ALU
  - therefore need complete cycle for ALU operations, and complete cycle for memory read or write operation.

Draft Micro-architecture

At this point, based on our assumptions we know that our datapath will need registers in addition to the ISA registers:

- To hold the 2 bytes of current instruction:
  - INST1
  - INST2
- Memory address register:
  - MAR
  - on memory write, address must be stable in MAR on posedge CLK
  - assume asynchronous read.
  - Will use other µarchitecture registers as memory data-in and data-out registers.
- Single ported general purpose register file.

Instruction RTL Description

add: X1<–GPR[ra];
X2<–GPR[rb], PC<–INST1[5,4];
Y<–X1+X2, INST1<–MEM[], PC<–PC+1, MAR<–PC+1;
GPR[rc]<–Y, <dispatch>;

Assumptions:
Both MAR and PC are left at the end of each instruction pointing to the byte after the current instruction.

<dispatch> expands as follows:

```
switch (op1):
    case 00: goto add;
    case 01: goto sub;
    case 10: goto nor;
    case 11:
        switch (op2) {
            case 00: goto ldb;
            case 01: goto stb;
            case 10: goto beq;
        }
```
<table>
<thead>
<tr>
<th>Instruction RTL Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub: X1←GPR[ra]; X2←GPR[rb]; RC←INST1[5,4]; Y←X1-X2, INST1→MEM[], PC←PC+1, MAR←PC+1; GPR[rc]←Y, &lt;dispatch&gt;;</td>
</tr>
<tr>
<td>nor: X1←GPR[ra]; X2←GPR[rb]; RC←INST1[5,4]; Y←X1 NOR X2, INST1→MEM[], PC←PC+1, MAR←PC+1; GPR[rc]←Y, &lt;dispatch&gt;;</td>
</tr>
<tr>
<td>ldb: X1←GPR[ra], INST2→MEM[]; X2←INST2, RC←INST1[5,4]; MAR←X1+X2; Y←MEM[], PC←PC+1, MAR←PC+1; INST1→MEM[], PC←PC+1, MAR←PC+1; GPR[rc]←Y, &lt;dispatch&gt;;</td>
</tr>
</tbody>
</table>

Data-path for add, sub, nor

Control signals shown in courier font.

Data-path with modifications for ldb

Control signals shown in courier font.
Data-path with modifications for stb

Control Signals

From data-path to controller:
- op1, op2: instruction opcode, used for dispatch
  - Note that “zero” signal is used internal to the data-path and does not need to go to the controller.

From controller to data-path:
- regRW: selects read or write for register file, GPR
- X1Sel: controls X1 mux
- X1Enb: write enable for X1
- X2Sel: controls X2 mux
- X2Enb: write enable for X2
- regSel[1:0]: chooses instruction field for register file address
- ALUcntl[1:0]: selection operation for ALU
- YSel: controls Y mux
- YEnb: write enable for Y
- I1Enb: Instruction Register 1 enable (don’t need one for 2)
- RCEnb: RC register enable
- MARSel: controls MAR mux
- MAREnb: write enable for X1
- memRW: selects read or write for memory
- PCEnb: write enable for PC
- branch: asserted on 4th cycle of beq, lets ALU write PC

Complete Data-path (including beq)

High-level controller design

- Controller design is simply a matter of designing a FSM.
  - Input is op1 and op2, output is the 18 control signals.
  - In this case we have 31 different states (sum of all the RTL cycles over all instruction types).
  - Each state puts out the appropriate control signals.
  - Most of the state transitions are not based on input (unconditional).
  - The last state in each instruction branches to one of the 7 instruction start states based on op1 and op2.
Controller Implementation

- Because of the special structure of the controller state transition diagram, a memory based implementation is efficient.
- Each word in a special memory stores the control signals for one state of the FSM.
- A counter (called micro-PC) keeps track of which state is currently active and is used to address the memory.
- On most cycles the micro-PC is simply incremented to get to the next state.
- On the last state of each instruction control sequence, the micro-PC is replaced by the contents of a jump table, indexed by op1 and op2.
- The replacement of the micro-PC is controlled by one additional control signal stored in the memory.
- This style of controller design is called micro-programming.
  - The contents of the controller memory is called micro-code.

Micro-programming

- Micro-programming provides a particularly simple way to design a controller when the control sequence matches the structure of a “program”. Straight state sequences with few branches.
- It makes changing the controller, to fix bugs or add features, easy. Allows changes late in the design process.
- Computers have been manufactured with user writeable control store (WCS)! Micro-code stored in RAM instead of ROM.
  - DEC VAX 780
  - Why?