EECS150 - Digital Design
Lecture 25 - High-level Design and Optimization 3, CPU Core Example

Nov 14, 2002
John Wawrzynek
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 Wawrzynek)
  – 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:**

- **Registers:**
  - 4 8-bit general purpose registers (GPR).
  - R0 reads as all 0s.
  - Program counter (PC) points to next instruction in memory. Resets to 0.

- **Instructions:** Two formats
  - r-format
    - op1  rc  ra  rb
  - o-format
    - op1  rc  ra  op2  offset
  - ra, rb, rc are 2-bit GPR specifiers
  - r-format opcode is specified by op1
  - o-format opcode is specified by op1 and op2.
### Instruction Set Architecture (ISA)

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

- ALU input and output registers:
  - X1
  - X2
  - Y
  - zero
  - ALU control
  - Zero output is asserted if result of subtraction is zero.
  - Assume controller supplies input to define function of ALU.
Instruction RTL Description

add: \[ X_1 \leftarrow \text{GPR}[ra]; \]
[130x67] 
\[ X_2 \leftarrow \text{GPR}[rb], \quad RC \leftarrow \text{INST1}[5,4]; \]
[154x139] 
\[ Y \leftarrow X_1 + X_2, \quad \text{INST1} \leftarrow \text{MEM}[], \quad \text{PC} \leftarrow \text{PC} + 1, \quad \text{MAR} \leftarrow \text{PC} + 1; \]
[178x139] 
\[ \text{GPR}[rc] \leftarrow Y, \quad \text{<dispatch>}; \]

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:

\[
\text{switch} \ (\text{op1}): \{ \\
\quad \text{case } 00: \quad \text{goto add;} \\
\quad \text{case } 01: \quad \text{goto sub;} \\
\quad \text{case } 10: \quad \text{goto nor;} \\
\quad \text{case } 11: \quad \text{\quad } \text{switch} \ (\text{op2}) \{ \\
\quad \quad \text{case } 00: \quad \text{goto ldb;} \\
\quad \quad \text{case } 01: \quad \text{goto stb;} \\
\quad \quad \text{case } 10: \quad \text{goto beq;} \} \}
\]
Instruction RTL Description

sub:  \[ X_1 \leftarrow \text{GPR}[ra]; \]
    \[ X_2 \leftarrow \text{GPR}[rb], \text{RC} \leftarrow \text{INST1}[5,4]; \]
    \[ Y \leftarrow X_1 - X_2, \text{INST1} \leftarrow \text{MEM}[], \text{PC} \leftarrow \text{PC}+1, \text{MAR} \leftarrow \text{PC}+1; \]
    \[ \text{GPR}[rc] \leftarrow Y, \text{<dispatch>}; \]

nor:  \[ X_1 \leftarrow \text{GPR}[ra]; \]
      \[ X_2 \leftarrow \text{GPR}[rb], \text{RC} \leftarrow \text{INST1}[5,4]; \]
      \[ Y \leftarrow X_1 \text{ NOR } X_2, \text{INST1} \leftarrow \text{MEM}[], \text{PC} \leftarrow \text{PC}+1, \text{MAR} \leftarrow \text{PC}+1; \]
      \[ \text{GPR}[rc] \leftarrow Y, \text{<dispatch>}; \]

ldb:  \[ X_1 \leftarrow \text{GPR}[ra], \text{INST2} \leftarrow \text{MEM}[]; \]
      \[ X_2 \leftarrow \text{INST2}, \text{RC} \leftarrow \text{INST1}[5,4]; \]
      \[ \text{MAR} \leftarrow X_1 + X_2; \]
      \[ Y \leftarrow \text{MEM}[], \text{PC} \leftarrow \text{PC}+1, \text{MAR} \leftarrow \text{PC}+1; \]
      \[ \text{INST1} \leftarrow \text{MEM}[], \text{PC} \leftarrow \text{PC}+1, \text{MAR} \leftarrow \text{PC}+1; \]
      \[ \text{GPR}[rc] \leftarrow Y, \text{<dispatch>}; \]
Instruction RTL Description

\[ \text{stb: } X1 \leftarrow \text{GPR}[ra], \text{ INST2} \leftarrow \text{MEM[]} \;
X2 \leftarrow \text{INST2} \;
\text{MAR} \leftarrow X1 + X2, \ X2 \leftarrow \text{GPR}[rc] \;
\text{MEM[]} \leftarrow X2, \ \text{PC} \leftarrow \text{PC} + 1, \ \text{MAR} \leftarrow \text{PC} + 1 \;
\text{INST1} \leftarrow \text{MEM[]} \,, \text{PC} \leftarrow \text{PC} + 1, \ \text{MAR} \leftarrow \text{PC} + 1 \;
<\text{dispatch}>; \]

\[ \text{beq: } X1 \leftarrow \text{GPR}[ra], \text{ INST2} \leftarrow \text{MEM[]} \;
X2 \leftarrow \text{GPR}[rb] \;
\text{ZERO} \leftarrow X1 - X2, \ X1 \leftarrow \text{PC}, \ X2 \leftarrow \text{INST2} \;
\text{if} \ \text{ZERO} \ \text{PC} \leftarrow X1 + X2 \;
\text{PC} \leftarrow \text{PC} + 1, \ \text{MAR} \leftarrow \text{PC} + 1 \;
\text{INST1} \leftarrow \text{MEM[]} \,, \text{PC} \leftarrow \text{PC} + 1, \ \text{MAR} \leftarrow \text{PC} + 1 \;
<\text{dispatch}>; \]
Data-path for add, sub, nor

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

- ALU
- X1
- X2
- RC
- op1
- op2
- regSel[1:0]
- regRW
- Y
- MAREnb
- memRW
- MARSel
- MAR
- PC
- PCEnb
- I1Enb
- RCEnb
- regSel[1:0]
- X2Sel
- X2Enb
- X1Enb
- ALUcntl[1:0]
- YEnb
- +1
- INST1
- INST2
- YEnb
Data-path with modifications for stb
Complete Data-path (including beq)

- **INST1**
  - Inputs: X1Sel, X1Enb, ALUcntl[1:0], YSel
  - Outputs: X1, X2, X2Sel, Y, YEnb

- **INST2**
  - Inputs: I1Enb, RCEnb
  - Outputs: YEnb, MAR, MAREnb, MARSel

- **GPR**
  - Inputs: regSel[1:0], from PC, memRW
  - Outputs: X1Sel, X2Sel, X1Enb, X2Enb

- **ALU**
  - Inputs: ALUcntl[1:0], YSel, YEnb
  - Outputs: X1, X2

- **RC**
  - Inputs: branch, from zero

- **MAR**
  - Inputs: MARSel, memRW

- **PC**
  - Inputs: +1

- **MEM**
  - Inputs: branch, from zero, memRW

The image shows a detailed schematic of a data-path including a branch instruction (beq). The diagram illustrates how different components interact, such as the instruction register (INST1 and INST2), the general-purpose register (GPR), the arithmetic logic unit (ALU), the memory address register (MAR), and the program counter (PC). The schematic highlights the flow of data and control signals, showing how operands are selected and how results are directed to the appropriate destinations.
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
- ALUcnt1[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
High-level controller design

- Controller design is simply a matter of designing a FSM.
  - Input is $\text{op}_1$ and $\text{op}_2$, 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 $\text{op}_1$ and $\text{op}_2$. 

![Diagram of controller design](image-url)
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.
Controller Implementation

memory decoder

micro-PC

+1

jump table

op1

op2
todata-path

0
00
000
001

5

27
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?