Lecture 26 –
Single Cycle CPU Datapath, with Verilog

2004-10-29

Lecturer PSOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

Halloween plans? ⇒ Try the Castro!

Sun 2004-10-31, from 7pm-mid ($3 donation)
go at least once…

halloweeninthe Castro II

A Neighborhood Celebration

halloweeninthe Castro.com
Anatomy: 5 components of any Computer

- Processor: Control ("brain")
- Datapath ("brawn")
- Memory: (where programs, data live when running)
- Devices: Input, Output
- Keyboard, Mouse
- Disk: (where programs, data live when not running)
- Display, Printer

This week and next...
Outline of Today’s Lecture

• Design a processor: step-by-step
• Requirements of the Instruction Set
• Hardware components that match the instruction set requirements
How to Design a Processor: step-by-step

1. Analyze instruction set architecture (ISA) => datapath requirements
   - meaning of each instruction is given by the register transfers
   - datapath must include storage element for ISA registers
   - datapath must support each register transfer

2. Select set of datapath components and establish clocking methodology

3. Assemble datapath meeting requirements

4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.

5. Assemble the control logic
Review: The MIPS Instruction Formats

• All MIPS instructions are 32 bits long. 3 formats:

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>op</strong></td>
<td><strong>rs</strong></td>
<td><strong>rt</strong></td>
<td><strong>rd</strong></td>
<td><strong>shamt</strong></td>
<td><strong>funct</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
<tr>
<td>I-type</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-type</td>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• The different fields are:
  • **op**: operation ("opcode") of the instruction
  • **rs**, **rt**, **rd**: the source and destination register specifiers
  • **shamt**: shift amount
  • **funct**: selects the variant of the operation in the "op" field
  • **address / immediate**: address offset or immediate value
  • **target address**: target address of jump instruction
Step 1a: The MIPS-lite Subset for today

- **ADDU and SUBU**
  - `addu rd, rs, rt`
  - `subu rd, rs, rt`

- **OR Immediate:**
  - `ori rt, rs, imm16`

- **LOAD and STORE Word**
  - `lw rt, rs, imm16`
  - `sw rt, rs, imm16`

- **BRANCH:**
  - `beq rs, rt, imm16`
Register Transfer Language (Behavioral)

- RTL gives the **meaning** of the instructions

\[ \{ \text{op} , \text{rs} , \text{rt} , \text{rd} , \text{shamt} , \text{funct} \} = \text{MEM}[\ \text{PC} \ ] \]

\[ \{ \text{op} , \text{rs} , \text{rt} , \text{Imm16} \} = \text{MEM}[\ \text{PC} \ ] \]

- All start by fetching the instruction

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfers</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU</td>
<td>( \text{R[rd]} = \text{R[rs]} + \text{R[rt]} );</td>
<td>( \text{PC} = \text{PC} + 4 )</td>
</tr>
<tr>
<td>SUBU</td>
<td>( \text{R[rd]} = \text{R[rs]} - \text{R[rt]} );</td>
<td>( \text{PC} = \text{PC} + 4 )</td>
</tr>
<tr>
<td>ORI</td>
<td>( \text{R[rt]} = \text{R[rs]} \mid \text{zero_ext(Imm16)} );</td>
<td>( \text{PC} = \text{PC} + 4 )</td>
</tr>
<tr>
<td>LOAD</td>
<td>( \text{R[rt]} = \text{MEM}[\ \text{R[rs]} + \text{sign_ext(Imm16)}];\text{PC} = \text{PC} + 4 )</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>( \text{MEM}[\ \text{R[rs]} + \text{sign_ext(Imm16)} ] = \text{R[rt]};\text{PC} = \text{PC} + 4 )</td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td>if ( ( \text{R[rs]} == \text{R[rt]} ) ) then</td>
<td>( \text{PC} = \text{PC} + 4 + (\text{sign_ext(Imm16)} | 00) )</td>
</tr>
<tr>
<td></td>
<td>else ( \text{PC} = \text{PC} + 4 )</td>
<td></td>
</tr>
</tbody>
</table>
Step 1: Requirements of the Instruction Set

• Memory (MEM)
  • instructions & data

• Registers (R: 32 x 32)
  • read RS
  • read RT
  • Write RT or RD

• PC

• Extender (sign extend)

• Add and Sub register or extended immediate

• Add 4 or extended immediate to PC
Step 2: Components of the Datapath

- Combinational Elements
- Storage Elements
  - Clocking methodology
16-bit Sign Extender for MIPS Interpreter

// Sign extender from 16- to 32-bits.
module signExtend (in,out);
    input [15:0] in;
    output [31:0] out;
    reg [31:0] out;

    out = { in[15], in[15], in[15], in[15],
            in[15], in[15], in[15], in[15],
            in[15], in[15], in[15], in[15],
            in[15], in[15], in[15], in[15],
            in[15], in[15], in[15], in[15],
            in[15:0] };
endmodule  // signExtend
2-bit Left shift for MIPS Interpreter

// 32-bit Shift left by 2
module leftShift2 (in,out);
    input [31:0] in;
    output [31:0] out;
    reg [31:0] out;

    out = { in[29:0], 1'b0, 1'b0 }; 
endmodule // leftShift2
Combinational Logic Elements (Building Blocks)

- **Adder**
  - A \(32\) \rightarrow \text{Adder} \rightarrow \text{Sum} \(32\)
  - B \(32\) \rightarrow \text{Adder} \rightarrow \text{CarryOut}

- **MUX**
  - A \(32\) \rightarrow \text{MUX} \rightarrow \text{Y} \(32\)
  - B \(32\) \rightarrow \text{MUX} \rightarrow \text{Y}

- **ALU**
  - A \(32\) \rightarrow \text{ALU} \rightarrow \text{Result} \(32\)
  - B \(32\) \rightarrow \text{ALU} \rightarrow \text{Result}
Verilog 32-bit Adder for MIPS Interpreter

```verilog
//Behavioral model of 32-bit adder.
module add32 (S,A,B);
    input [31:0] A,B;
    output [31:0] S;
    reg [31:0] S;

    always @(A or B)
        S = A + B;
endmodule // add32
```
Verilog 32-bit Register for MIPS Interpreter

// Behavioral model of 32-bit wide 2-to-1 multiplexor.
module mux32 (in0,in1,select,out);
  input [31:0] in0,in1;
  input select;
  output [31:0] out;
  reg [31:0] out;

  always @ (in0 or in1 or select)
    if (select) out=in1;
    else          out=in0;

endmodule  // mux32
ALU Needs for MIPS-lite + Rest of MIPS

• Addition, subtraction, logical OR, ==:
  ADDU \( R[rd] = R[rs] + R[rt]; \ldots \)
  SUBU \( R[rd] = R[rs] - R[rt]; \ldots \)
  ORI \( R[rt] = R[rs] |\)
     \( \text{zero_ext}(\text{Imm16}) \ldots \)
  BEQ \( \text{if} \ ( R[rs] == R[rt] ) \ldots \)

• Test to see if output == 0 for any ALU operation gives == test. How?

• P&H also adds AND,
  Set Less Than (1 if \( A < B \), 0 otherwise)

Behavioral ALU follows chap 5
Verilog ALU for MIPS Interpreter (1/3)

// Behavioral model of ALU:
// 8 functions and "zero" flag,
// A is top input, B is bottom

module ALU (A,B,control,zero,result);
  input  [31:0] A, B;
  input  [2:0]  control;
  output zero;  // used for beq,bne
  output [31:0] result;

  reg zero;
  reg [31:0] result, C;
  always @ (A or B or control)...

Verilog ALU for MIPS Interpreter (2/3)

```verilog
reg [31:0] result, C;
always @ (A or B or control)
begin
  case (control)
    3'b000: // AND
      result=A&B;
    3'b001: // OR
      result=A|B;
    3'b010: // add
      result=A+B;
    3'b110: // subtract
      result=A-B;  // Documents bugs below
    3'b111: // set on less than
      // old version (fails if A is
      // negative and B is positive)
      // result = (A<B)? 1 : 0; wrong

  endcase
end
```

// Why did it fail?
Verilog ALU for MIPS Interpreter (3/3)

// result = (A<B)? 1 : 0; wrong
// current version
// if A and B have the same sign,
// then A<B works (slt == 1 if A-B<0)
// if A and B have different signs,
// then A<B if A is negative
// (slt == 1 if A<0)

begin
  C = A - B;
end
endcase // case(control)

zero = (result==0) ? 1'b1 : 1'b0;
end // always @ (A or B or control)
endmodule // ALU
### Storage Element: Idealized Memory

- **Memory (idealized)**
  - One input bus: Data In
  - One output bus: Data Out

- **Memory word is selected by:**
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus

- **Clock input (CLK)**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after “access time.”
Verilog Memory for MIPS Interpreter (1/3)

//Behavioral model of Random Access Memory:
// 32-bit wide, 256 words deep,
// asynchronous read-port if RD=1,
// synchronous write-port if WR=1,
// initialize from hex file ("data.dat")
// on positive edge of reset signal,
// dump to binary file ("dump.dat")
// on positive edge of dump signal.

module mem
  (CLK,RST,DMP,WR,RD,address,writeD,readD);
  input CLK, RST, DMP, WR, RD;
  input [31:0] address, writeD;
  output [31:0] readD;
  reg [31:0] readD;
  parameter memSize=256; // ~ Constant dec.
  reg [31:0] memArray [0:memSize-1];
  integer chann,i;
  // Temp variables: for loops...
Verilog Memory for MIPS Interpreter (2/3)

```vhdl
integer chann,i;
always @ (posedge RST)
    $readmemh("data.dat", memArray);
always @ (posedge CLK)
    if (WR) memArray[address[9:2]] = writeD;

// write if WR & positive clock edge (synchronous)
always @ (address or RD)
    if (RD)
        begin
            readD = memArray[address[9:2]];
            $display("Getting address %h containing %h", address[9:2], readD);
        end

// read if RD, independent of clock (asynchronous)
```
Verilog Memory for MIPS Interpreter (3/3)

```verilog
end;
always @ (posedge DMP)
begin
  chann = $fopen("dump.dat");
  if (chann==0)
    begin
      $display("$fopen of dump.dat failed.");
      $finish;
    end
    // Temp variables chan, i
  for (i=0; i<memSize; i=i+1)
    begin
      $fdisplay(chann, "%b", memArray[i]);
    end
  end
  // always @ (posedge DMP)
endmodule // mem
```
Peer Instruction

A. We should use the main ALU to compute PC=PC+4

B. We’re going to be able to read 2 registers and write a 3rd in 1 cycle

C. Datapath is hard, Control is easy
5 steps to design a processor

1. Analyze instruction set => datapath requirements
2. Select set of datapath components & establish clock methodology
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
5. Assemble the control logic

Control is the hard part

Next time!