ELEC6234
Embedded Processor Design

FPGA synthesis of a microprocessor core
Notes on ALU synthesis
picoMIPS block diagram - version 1

pMIPS version 1 – no RAM, no branches
picoMIPS with branches and RAM
Data paths for arithmetic and logic instructions
operands and result are in registers, RAM (data memory) is not involved
RAM memory load and store instructions
the only instructions using the RAM (data memory) are load (LD) and store (ST)

- ALU is used for address calculation
  - typically ALU is programmed to copy operand B to Result: address is in register Rs or immediate
- Data move from RAM to Rd (load) or from Rd to RAM (store)

Load operation: Rd <= RAM[address]
Store operation: RAM[address] <= Rd

ALU function: Result = B

Sync RAM
[2^n x n]

Address
Rdata
Wdata

Regs [2^m x n]
WRd_data

[n]
[n]

Rs Rs_data

[n]

[n-1:0] immediate

Rd Rd_data
Absolute jump:
PC <= branch address
where branch address is: Instruction[p-1:0]

Relative jump:
PC <= PC + Instruction[p-1:0]

Relative offset needs an adder within the PC

Branch Address: bits [p-1:0] of the instruction
Indirect Jump
absolute or relative

Absolute jump:
PC <= branch address
where branch address is: Instruction[p-1:0]

Relative jump:
PC <= PC + Instruction[p-1:0]
Conditional branches
absolute or relative

Absolute conditional branch:
PC \leq (\text{condition? branch address: } PC+1)
where branch address is: Instruction[p-1:0]

Relative conditional branch:
PC \leq (\text{condition? } PC + \text{Instruction}[p-1:0]: PC+1)

ALU is programmed for subtraction: Rd-Rs
Subtraction result is not stored but ALU flags: Z,N,C are used to determine if the branch should be taken

Branch address: bits [p-1:0] of the instruction
The picoMIPS decoder is a combinational logic module.
// sample picoMIPS program 1
// n = 8 bits, Isize = 16+n = 24 bits
// format: 6b opcode, 5b %d, 5b %s, 8b immediate or address

// HEX ///////////////////////////////////////////////////// BINARY ////////////////////////////////////////////////////////// ASSEMBLER /////
000000 // 24'b0000_0000_0000_0000_0000_0000 NOP
282005 // 24'b0010_1000_0010_0000_0000_0101 ADDI %1, %0, 5; load 5 in register 1
284007 // 24'b0010_1000_0100_0000_0000_0111 ADDI %2, %0, 7; load 7 in register 2
082200 // 24'b0000_1000_0010_0010_0000_0000 ADD %1, %2; %1 = %1 + %2
000000 // 24'b0000_0000_0000_0000_0000_0000 NOP
000000 // 24'b0000_0000_0000_0000_0000_0000 NOP
000000 // 24'b0000_0000_0000_0000_0000_0000 NOP
CPU module – ver 1

see file cpu.sv

Top-level Entity Name: cpu
Family: Cyclone IV GX
Total logic elements: 135
Total combinational functions: 111
Dedicated logic registers: 38
Total registers: 38
picoMIPS block diagram - version 2

pMIPS version 2 – no RAM, with branches
This decoder implements full PC control for conditional and unconditional branches.
Decoder – version 2

// ------------ code starts here -----------
// instruction decoder
logic takeBranch; // temp variable to control conditional branching

always_comb
begin
  // set default output signal values for NOP instruction
  PCincr = 1'b1; // PC increments by default
  PCabsbranch = 1'b0; PCrelbranch = 1'b0;
  ALUfunc = opcode[2:0];
  imm=1'b0; w=1'b0;
  takeBranch = 1'b0;
  case(opcode)
    `NOP: $;
    `ADD, `SUB: begin // register-register
      w = 1'b1; // write result to dest register
    end
    `ADDI, `SUBI: begin // register-immediate
      w = 1'b1; // write result to dest register
      imm = 1'b1; // set ctrl signal for imm operand MUX
    end
  // relative conditional branches
    `BEQ: takeBranch = flags[1]; // branch if Z==1
    `BNE: takeBranch = ~flags[1]; // branch if Z==0
    `BGE: takeBranch = ~flags[2]; // branch if N==0
    `BLO: takeBranch = flags[0]; // branch if C==1
    default:
      $error("unimplemented opcode %h",opcode);
  endcase // opcode
...

Temp signal takeBranch is set according to ALU flags
This if statement implements the PC control logic for relative branches

```plaintext
if(takeBranch) // relative branch condition is true;
begin
    PCincr = 1'b0;
    PCrelbranch = 1'b1;
end

end // always_comb

endmodule //module decoder
```
### prog.hex – version 2 with branches

// sample picoMIPS program 2
// n = 8 bits, Isize = 16+n = 24 bits
// format: 6b opcode, 5b %d, 5b %s, 8b immediate or address
//
// HEX /////////// BINARY ////////////// ASSEMBLER //////////

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Binary Code</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>24'b0000_0000_0000_0000_0000_0000</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>282003</td>
<td>24'b0010_1000_0010_0000_0000_0011</td>
<td>ADDI %1, %0, 3; load 3 in register 1 (loop count)</td>
<td></td>
</tr>
<tr>
<td>284000</td>
<td>24'b0010_1000_0100_0000_0000_0000</td>
<td>ADDI %2, %0, 0; clear reg 2</td>
<td></td>
</tr>
<tr>
<td>284201</td>
<td>24'b0010_1000_0100_0010_0000_0001</td>
<td>ADDI %2, %2, 1; increment reg 2</td>
<td></td>
</tr>
<tr>
<td>2C2101</td>
<td>24'b0010_1100_0010_0001_0000_0001</td>
<td>SUBI %1, %1, 1; decrement reg 1</td>
<td></td>
</tr>
<tr>
<td>6C2101</td>
<td>24'b0110_1100_0010_0001_1111_1110</td>
<td>BNE %1, %0, -2; branch if %1!=0</td>
<td></td>
</tr>
<tr>
<td>084200</td>
<td>24'b0000_1000_0100_0010_0000_0000</td>
<td>ADD %2, %2, %0; %2 = %2, display reg 2</td>
<td></td>
</tr>
<tr>
<td>084200</td>
<td>24'b0000_1000_0100_0010_0000_0000</td>
<td>ADD %2, %2, %0; %2 = %2, display reg 2</td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>24'b0000_0000_0000_0000_0000_0000</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>24'b0000_0000_0000_0000_0000_0000</td>
<td>NOP</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1) syntax is simple: each line contains instruction code in hex followed by a comment

2) this is a simple test that uses NOP, LDI (Load Immediate to Register) and ADD instructions

3) LDI is a ‘synthetic’ instruction; LDI %d, imm is implemented as: ADDI %d, %0, imm
Other instructions

- **J** – Jump
  - Syntax: J <address>
  - Operation: PC = <address>
- **JAL** - Jump and link, simple subroutine call
  - Syntax: JAL <subroutine address>
  - Operation: PC = <subroutine address>, %31 = PC+1 (return address)
- **JR** – Jump to address in a register
  - Syntax: JR %r
  - Operation: PC = %r, (JR %31 may be used as return from subroutine)
- **MULT** – multiply signed
  - Syntax: MULT %s, %d
  - Operation: %d = %s *%d (dedicated registers might be needed in hardware)
pMIPS version 3 – with RAM and branches

- **Control**, **Address**, **Data**

**picoMIPS with RAM**

- **PC control**
- **Program Memory [p x n+16]**
- **[n+16]**
- **[n+4:n]**
- **[n+10:n+5]**
- **[n+15:n+10]**
- **Decoder**
- **Regs write**
- **ALU function**
- **ALU flags**
- **RAM write**
- **Sync RAM [2^n x n]**
- **Address**
- **Rdata**
- **Wdata**

**[n-1:0] branch address**

**[n-1:0] immediate**

**ddata**

**[n]**

**Rs**

**Rs_data**

**Wd_data**

**[n]**

**Rd**

**Rd_data**

**[n]**

**Result**

**Sync RAM [2^n x n]**

**Address**

**Rdata**

**Wdata**
RAM instructions

- **LW** – load word from RAM
  - Syntax LW %d, %s, imm
  - Operation: %s = RAM[%d + imm]

- **SW** – store word in RAM
  - Syntax SW %d, %s, imm
  - Operation: RAM[%d + imm] = %s

- Load must be performed in two clock cycles if the RAM is synchronous
  - This can be implemented as two instructions in the program memory:
    - LW1 – selects registers, programs ALU and clocks RAM
    - LW2 – writes RAM output into registers
Summary

– Data path
  • ALU, General Purpose Registers,

– Control path
  • Program Counter, Program Memory,

– Simple decoder with essential instructions
  • ADD, SUB, ADDI, SUBI, BEQ, BNE, BGE, BLO

– CPU encapsulating module to allow integration of modules developed so far and execution of simple programs

– Major features still to be implemented:
  • RAM, load and store instructions,
  • subroutine calls and returns
  • multiplier
  • shifter