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
Approaches to ALU synthesis

• Methods to synthesise ADD, SUB
  – 1) use a+b in 9-bits to extract carry on 9\textsuperscript{th} bit,
  – 2) use a+b in 8-bits, but add extra logic to extract carry from ALU MSB slice,
  – 3) synthesise ALU on structural level using 1-bit slices.

• For small hardware size use method 2) or 3)
• Subtraction can be implemented using 2’s complement subtraction rule:
  – A − B = A + \neg B + 1
  – This way the same hardware can be used for ADD and SUB
• 2’s complement overflow (if needed!) can be obtained from simple logic as shown in ALU code below.
Writing synthesizable SystemVerilog code for combinational logic synthesis

- When writing code for decoders or multiplexers, use conditional statements
  - if-then-else
  - case

- Common trap:
  Conditional statements with incompletely specified signals cannot be mapped into combinational logic:

  ```verilog
  if(a)
  b = 1'b0; // b is incompletely specified
  ```

- Use the `default` clause
  - Assign default values to all signals driven by combinational logic blocks (examples will follow)

- In arithmetic circuit synthesis
  - Arithmetic operations `+` `-` are mapped into standard adders/subtractors
  - Synthesis tools, e.g. Synplify Pro/Quartus map the multiplication operator `*` into dedicated multipliers if available, or combinational (cellular) multipliers – fast but very costly!
module decoder2to4(
    input logic [1:0] i, output logic [3:0] outp);

always_comb // note absence of sensitivity clause
begin
    outp = 4'b0000; // default value
    unique case (i) // unique will force completeness test
        0: outp[0] = 1'b1;
        1: outp[1] = 1'b1;
        2: outp[2] = 1'b1;
        3: outp[3] = 1'b1;
    endcase
end
endmodule
module mux4to1(
    input logic [3:0] i, input logic [1:0] sel, output logic outp);

always_comb // note absence of sensitivity clause
unique case (sel) // unique will force completeness test
    0: outp = i[0];
    1: outp = i[1];
    2: outp = i[2];
    3: outp = i[3];
endcase
endmodule
Arithmetic circuits - 8-bit adder

// 8-bit adder - note the trick of using 9-bit addition to obtain carry out
module adder8
    (input logic [7:0] A, B, input logic Cin,
     output logic [7:0] Sum, output logic Cout);

    assign {Cout,Sum} = {A[7],A} + {B[7],B} + Cin; // sign-extend by 1 bit
endmodule
Arithmetic circuits - 8-bit adder synthesised for Altera Cyclone III FPGA (1)
Arithmetic circuits - 8-bit adder synthesised for Altera Cyclone III FPGA (2)
Sample ALU module

`include "alu.sv"

module alu #(parameter n = 8) (
  input logic [n-1:0] a, b, // ALU operands
  input logic [2:0] func, // ALU function code
  output logic [3:0] flags, // ALU flags V,N,Z,C
  output logic [n-1:0] result // ALU result
);
...

`define RA 3'b000
`define RB 3'b001
`define RADD 3'b010
`define RSUB 3'b011
`define RAND 3'b100
`define ROR 3'b101
`define RXOR 3'b110
`define RNOR 3'b111
This always_comb block creates an adder and logic for \(-B\) needed in subtraction.

The adder result (\(ar\)) will be used in the always_comb block that follows.
alu.sv

This always_comb block creates the ALU.

Note the logic for ALU flags:
V – overflow
C - carry on MSB

See following slides for explanation of V and C logic
alu.sv

ALU module - cont

... 

// calculate flags Z and N
flags[1] = result == {n1'b0}; // Z
flags[2] = result[n-1]; // N

end //always_comb

endmodule //end of module ALU

The logic for ALU flags:
Z – zero result
N - negative result
Carry logic

When coding addition as A+B, we don’t know Cin at MSB, but we know Sum[7]

Carry is set at MSB slice if:


In subtraction A-B, complement B[7]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Sum</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Detection of 2’s complement overflow

Overflow occurs in addition if the signs of both operands are the same, but the result sign is different, i.e.


In subtraction A-B, complement B[7]

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$Cin$</th>
<th>$Sum$</th>
<th>$Cout$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
ALU testbench

---

`include "alucodes.sv"
module alutest;
parameter n =8; // data bus width
logic [n-1:0] a, b; // ALU input operands
logic [2:0] func; // ALU function code
logic [3:0] flags; // ALU flags V,N,Z,C
logic [n-1:0] result; // ALU result

alu #(.n(n)) alu1 (.*); // create alu object

... //------------- code starts here ---------
initial
begin a= 8'h7a; b= 8'h08;
// test all functions
#10 func = `RA; // result = a
#10 func = `RB; // result = b
#10 func = `RADD; // result = a+b
#10 func = `RSUB; // result = a-b
#10 func = `RAND; // result = a & b
#10 func = `ROR; // result = a | b
#10 func = `RXOR; // result = a ^ b
#10 func = `RNOR; // result = ~ (a | b)
end //initial

endmodule //end of alutest
Modelsim waveforms
Altera Quartus synthesis
Altera Quartus synthesis - cont