**ELEC6234 Embedded Processor Design**

FPGA synthesis of a microprocessor core

Notes on ALU synthesis

---

**Approaches to ALU synthesis**

- Methods to synthesise ADD, SUB
  - 1) use \(a+b\) in 9 bits to extract carry on 9th 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’d0000; // 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

// 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

module decoder2to4(
    input logic [1:0] i,
    output logic [3:0] outp);

always_comb // note absence of sensitivity clause
begin
    outp = 4’d0000; // 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

// 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
alu.sv

-------- code starts here --------

// create an n-bit adder
// and then build the ALU around the adder
logic[n-1:0] ar,b1; // temp signals
always_comb
begin
if(func==`RSUB) b1 = ~b + 1'b1; // 2's complement subtrahend
else b1 = b;
ar = ar+b1; // n-bit adder
end // always_comb

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

-------- code starts here --------

// create the ALU, use signal ar in arithmetic operations
always_comb
begin
// default output values; prevent latches
flags = 3'b0;
result = a; // default
case(func)
`RA : result = a;
`RB : result = b;
`RADD : begin
result = ar; // arithmetic addition
end
`RSUB : begin
result = ~ar; // arithmetic subtraction
end
`RAND : result = a & b;
`ROR : result = a' & b;
`RXOR : result = a ^ b;
endcase
end

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

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:

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

ALU testbench

 ámbición code starts here -----
initial
begin
  a = 8'b11011011; // test all functions
  #20 func = RRA; // result = a
  #10 func = RRA; // result = b
  #10 func = RRA; // result = a & b
  #10 func = RRA; // result = a | b
  #10 func = RRA; // result = a ^ b
  #10 func = RRA; // result = ~ (a | b)
end // initial
endmodule // end of alutest