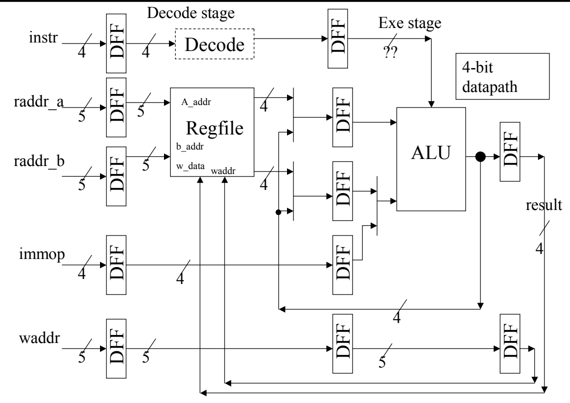
Simple Pipelined System

- All Spice problems in this homework are to be done for technologies
  - tsmc018.model, Vdd = 2.5 V, default temp
  - all input waveforms should have rise/fall times of 200 ps.
- It is encouraged that you work in teams of two (if you work alone, you must provide me with a reason).
- Will be worth twice that of previous assignments.



Goal

- Implement a simple pipelined system using static CMOS
- Goal is performance
- Problems you will face
  - Clock network will have a significant load
  - Must generate a gate level netlist from an RTL description
  - Must choose your own cells for this for this implementation.

Design Constraints

- Create your own Verilog, Spice level gate library for this design
  - Any complex gate design is valid
  - For spice level netlists, use the P_def and N_def transistor subcells used in previous labs
  - Other than inverters, no transistor can exceed a width of 8*wmin. For inverters, cannot use anything transistor larger than 32*wmin.
  - No layout – just spice level netlists
  - Can use any DFF design that you can find
  - Might want to consider integrating logic with latches to increase speed
- The register file will be simulated using a Verilog-A model in the spice level netlist.

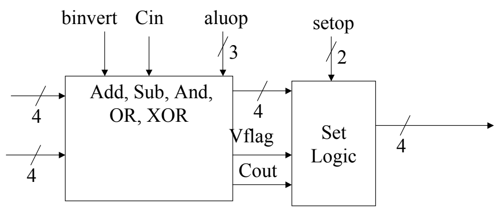
Design Constraints (cont.)

- You are free to design the ALU anyway you want to meet the required functionality
  - You cannot move any logic from the EXE stage back into the Decode stage other than doing pre-decoding on the instruction
- You must provide a Verilog gate level netlist that implements the RTL
  - Must be compatible with the current Verilog testbench
- I will provide a Spectre testbench at a later date
  - All inputs, including the clock, will driven by 1X buffers
  - All extra drive will have to provided by your spice netlist implementation

Supported Instructions

- Three operand format
  - op rdest, ra, rb rdest = ra op rb
- Arithmetic: add, sub
- Logic: xor, and, or
- Other: slt, sltu
- Set-Less-Than (slt rdest, ra, rb)
  - Rdest = 1 if ra < rb (signed comparison)
- Set-Less-Than-Unsigned (sltu rdest, ra,rb)
  - Rdest = 1 if ra < rb (unsigned comparison)

One Possible ALU Structure



aluop: Add, Sub, And, Or, Xor

setop: Slt, Sltu, or pass through if not a Set instruction

Add/Sub/And/Or Cell

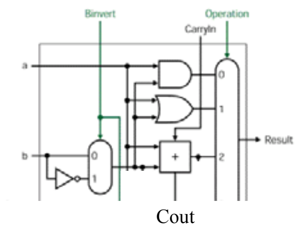


Diagram from Computer Architecture textbook. Need to add XOR.

Operations: AND, OR, ADD, SUB

If SUB, then Carryin LSB = 1, Binvert = 1.

Overflow Flag Logic

Overflow logic depends on whether doing an addition or subtraction:
if (addition) overflow = (A*msb* and B*msb* and (not Smsb)) or
((not A*msb*) and (not B*msb*) and Smsb)

i.e. For addition, if sign bits of operands are the same, but the result sign bit is different, then OVERFLOW has occurred. Smsb is the most significant bit of the result.

If (subtraction) OF = (Amsb and (not Bmsb) and (not Smsb)) or
((not Amsb) and Bmsb and Smsb)

Note: In all cases, Binvert = 1 for subtraction, Binvert = 0 for add

Set Logic

SLT, SLTU : output is non-zero if A < B; ALU always does a SUB operation, SET logic will output a zero or non zero value based on flags.

if SLT (signed comparison) then
Let Nf = sign bit of result, OF = overflag
Result (LSB) = ((not OF) and Nf) or (OF and (not Nf))
Result other bits = 0.

If SLTU (unsigned comparison) then
Result (LSB) = not(CarryFlag)
Result other bits = 0.

Note that a Set operation always produces as a result either '1' ('0001') or '0' ('0000').

Verilog RTL, Testbench

- In distribution, the directory ./modelsim/src/alu_rtl contains the verilog files
  - alu.v - RTL for pipelined system
  - tb_alu.v - testbench for alu.v
- To place modelsim on path do 'swsetup modelsim'
- To compile files do
  - cd ./modelsim/src/alu_rtl
  - gmake –f alu_rtl/Makefile
- To run simulation:
  - qhsim –lib alu_rtl tb_alu –c –do "run 20 us;quit"

Verilog Testbench

- The Verilog testbench file (tb_alu.v) reads a file called 'prog.obj' that contains the object code for the program to be executed.
- A perl script (assemble.pl) is used to assemble a program.
  - *assemble.pl add_prog.asm* will produce an object file called add_prog.obj
  - You should either link or copy your object file to 'prog.obj'.
- The perl script is *./modelsim/src/assemble.pl*
- Testbench provides inputs on rising clock edge.

A Sample Program

```
# comment line
add r1,r0,1   # r1 = 1
add r2,r0,2   # r2 = 2
add r3,r1,r2  # r3 = 3, activate bypass B
add r3,r3,r1  # r3 = 3+1 = 4, activate bypass A
sub r4,r3,r3  # r4 = 0
add r7,r3,-7  # r7 = 4-7 = -3 (0x13)
```

srcB, can be register or 4-bit signed immediate

dest   srcA, must be a register

Registers: r0-r31, r0 is always ‘0’.

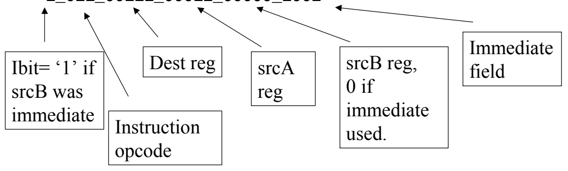
Object Code file produced by ‘*assemble.pl’*

```
// add r1,r0,1   # r1 = 1
1_011_00001_00000_00000_0001
// add r2,r0,2   # r2 = 2
1_011_00010_00000_00000_0010
// add r3,r1,r2  # r3 = 3, activate bypass B
0_011_00011_00001_00010_0000
// add r3,r3,r1  # r3 = 3+1 = 4, activate bypass A
0_011_00011_00011_00001_0000
// sub r4,r3,r3  # r4 = 0
0_100_00100_00011_00011_0000
// add r7,r3,-7  # r7 = 4-7 = -3 (0x13)
1_011_00111_00011_00000_1001
```

Must copy this file to prog.obj to be read by *tb_alu.v*

Object Code format

```
// add r7,r3,-7  # r7 = 4-7 = -3 (0x13)
1_011_00111_00011_00000_1001
```


Ibit= ‘1’ if srcB was immediate
Instruction opcode
Dest reg
srcA reg
srcB reg, 0 if immediate used.
Immediate field

tb_alu.v

```
module tb_alu;
wire [3:0] result, instr;
wire [4:0] waddr,raddr_a,raddr_b;
wire [3:0] immop;
reg clk;

reg [(1+3+5+5+5+4)-1:0] progmem[0:100];
integer pc,i;

assign {instr[3],instr[2:0],waddr,raddr_a,raddr_b,immop} = progmem[pc];
alu alu0 (result,instr,raddr_a,raddr_b,waddr,immop,clk);
initial begin
  clk = 0; pc = 0;
  for (i=0;i<100;i=i+1) progmem[i] = 0;
  $readmemb("prog.obj",progmem);
  /* run clock cycles */
  for (i=0;i<100;i=i+1)
   begin
    #100 clk = 1;
    #100 clk = 0;
    pc = pc +1;
   end
end
endmodule
```

Declare program memory array

Instantiate ALU

Read ‘prog.obj’ file.

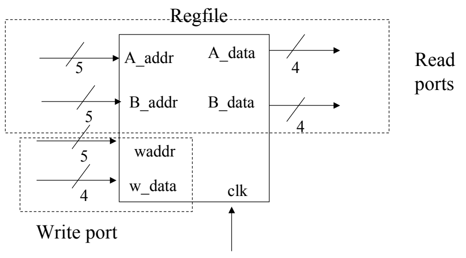
Run 100 clock cycles

alu.v

- Contains modules for the register file and alu (the ‘alu’ module contains everything except the register file)
- Register file module prints to screen anytime a write to register other than ‘r0’ occurs.
- Module alu.v contains the RTL for the functionality of the system
  - You could just synthesize this directly to a gate level netlist using a simple standard cell library, but the resulting logic would not be that fast
  - The block diagram shown earlier corresponds to the RTL description

Register File (32 x 4)

Register file is two read ports, one write port.
32 locations, 4 bits/location



Register File in *alu.v*

```
module regfile(a_data,b_data,raddr_a,raddr_b,waddr, w_data, wbit,clk);
output [3:0] a_data,b_data;
input [4:0] raddr_a,raddr_b,waddr;
input [3:0] w_data;
input wbit,clk;
integer j;
/* declare register file */
reg [3:0] rf[31:0];
initial  rf[0] = 0;
always @(wbit or clk or waddr or w_data)
  begin
   if (clk & wbit) begin
       if (waddr != 0)    rf[waddr] = w_data;
    end
  end
assign # 1 a_data = rf[raddr_a] ;
assign # 1 b_data = rf[raddr_b] ;
endmodule
```

Do a write when clk=1 and wbit=1. In alu, the wbit is always '1', but could change later.

Two read ports

Module alu in *alu.v*

```
module alu (result,instr,raddr_a,raddr_b,waddr,immop,clk);
 `define XOR_OP 0
 `define OR_OP  1
 `define AND_OP 2
 `define ADD_OP 3
 `define SUB_OP 4
 `define SLT_OP 5
 `define SLTU_OP 6

output [3:0] result;
input [3:0] instr;
input [4:0] raddr_a,raddr_b,waddr;
input [3:0] immop;
input clk;
```

Opcodes: cannot change!

Interface to system, cannot change this!!!

alu.v (cont).

Unit delay

```
 /* decode stage registers*/
 always @(posedge clk) id_instr =  #1 instr;
 always @(posedge clk) id_raddr_a =  #1 raddr_a;
 always @(posedge clk) id_raddr_b =  #1 raddr_b;
 always @(posedge clk) id_waddr = #1 waddr;
 always @(posedge clk) id_immop = #1 immop;
```

Statements like this simply represent a set of DFFs where the input is the right hand side, the output is the left hand side.

Bypass Paths in Decode Stage

Muxes in front of DFFs

```
always @(posedge clk)
  if (id_raddr_a == exe_waddr)   exe_a = #1 exe_result;
       else  exe_a = #1 rf_a;

always @(posedge clk)
  if (id_raddr_b == exe_waddr)  exe_b = #1 exe_result;
       else   exe_b = #1 rf_b;
```

If destination of previous instruction is equal to source of current instruction, use value from execute stage since it has not been written to register file yet!!!

```
add r2,r0,2    # r2 = 2
add r3,r1,r2   # r3 = 3, activate bypass B
```

alu.v (cont).

Combinational logic blocks.

```
/* alu, 1st stage */
always @(exe_instr  or exe_b or exe_immop )
  begin
   if (exe_instr[3]) exe_op_b = exe_immop;
      else exe_op_b = exe_b;
 end
always @(exe_instr or exe_a or exe_op_b )
  begin
       case (exe_instr[2:0])
       `XOR_OP:   exe_result_a = exe_a ^ exe_op_b;
       `OR_OP:   exe_result_a = exe_a | exe_op_b;
       `AND_OP:   exe_result_a = exe_a & exe_op_b;
       `ADD_OP:   exe_result_a = exe_a + exe_op_b;
       `SUB_OP:   exe_result_a = exe_a + ~exe_op_b +1;
       `SLT_OP:   exe_result_a = exe_a + ~exe_op_b +1 ;
       `SLTU_OP:   exe_result_a = exe_a +  ~exe_op_b + 1;
       default:   exe_result_a = exe_a;
        endcase
  end
```

2/1 mux for immediate operand.

Subtraction via 2's complement

ALU comb logic.

Transforming RTL To Gate Level Netlist

- One approach : could write a Synopsys .lib file for your library and just use synthesis to do transformation
  - Use dummy delays since no time to do characterization
  - Use simple gates like NANDs, NORs
  - Result would work but would be slow
- Another approach: look at required logic at the gate level, and design complex gates to reduce the register-to-register delay time since that will define the maximum clock frequency
  - Write Verilog gate level netlist manually (not that hard for a 4-bit ALU).
  - Execute stage will be the limiting stage so optimize this stage
  - You CANNOT move logic from the execute to the decode stage.

Writing Gate-level Verilog Modules

- The file ./sample_files/libcells.v has some sample module definitions for common library functions like nands, nors, xors, etc.
- When writing your gate level modules, do the following:
  - Use either UPPER case or LOWER case for all terminal/module names
  - Do NOT use mixed case
  - I would suggest using lowercase for everything (more readable).
  - Verilog is case sensitive
  - Use unit delays for gate level modules

A Sample Gate Level Module: Full Adder

```
module FA (SUM,CO, A, B, CI);

output SUM;
output CO;
input A;
input B;
input CI;

xor #2 I0 (SUM, A, B, CI);
and  I1 (N1,A, B);
and  I2 (N2,A, CI);
and  I3 (N3,CI, B);
or  #2 I4 (CO,N1,N2,N3);

endmodule
```

Port declarations

Verilog primitive gate

Instance name

Pinlist – for primitives, first pin is output followed by inputs.

Output gate delay – 2 time units.

A DFF

```
module DFF (Q, CLK, D);
output  Q;
reg Q;
input  D, CLK;


initial begin
 Q=0;
end
always @(posedge CLK)
  Q= #1 D;
endmodule
```

Triggered on positive clock edge

Hierarchical Gate-level Verilog

```
module XNOR8G (Y,A,B);

output [7:0] Y;
input [7:0] A,B;


XNOR2X1 I0 (Y[0],A[0],B[0]);
XNOR2X1 I1 (Y[1],A[1],B[1]);
XNOR2X1 I2 (Y[2],A[2],B[2]);
XNOR2X1 I3 (Y[3],A[3],B[3]);
XNOR2X1 I4 (Y[4],A[4],B[4]);
XNOR2X1 I5 (Y[5],A[5],B[5]);
XNOR2X1 I6 (Y[6],A[6],B[6]);
XNOR2X1 I7 (Y[7],A[7],B[7]);

endmodule
```

A module that instantiates 8 XNOR gates.

Previously declared module.

Due Dates

- Completed Spice Simulation are due on October 3rd (two weeks from today).
- Verilog Gate level simulation due on Thursday Sept 26th (this does not have to be the final gate level netlist, just a progress check).
  - Expect two files: libcells.v that has gate level modules and alu.v that is gate level netlist and register file module (do not change the register file module)

Rankings

- I will rank designs by achievable clock frequency
  - Spectre testbench will be provided by Thursday, Sept 26th.
- Upper 1/3 of class will get 25 points added to any test grade.
- Middle 1/3 will get 12 pts added to any test grade.
- Bottom 1/3 will get no extra points.