University of Pennsylvania
Department of Electrical and System Engineering
Computer Organization

ESE680-002, Spring 2007 **Assignment 2: Space-Time Multiply** Wed., Jan. 17

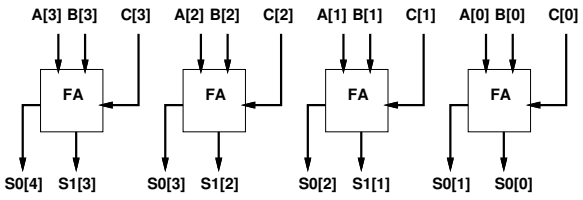
Due: Monday, January 29, 12:00PM

Everyone should do all problems.

We saw in lecture how to build various adders. In this problem, We're asking you to review or develop various techniques for building multipliers.

- Give latency and area in terms of the operand bitwidth, w. (we'll take asymptotic analysis, or you can use symbolic constants in terms of primitive gates such as $T_{fulladderslice}$, T_{and2})
- When asked to draw an implementation or provide microcode, you may show the $w = 4$ case. You may use hierarchical schematics.

1. Show a $w \times w$ spatial multiplier built out of simple, ripple-carry adders.
   - What is the area and latency?
2. Let's consider an alternate technique that uses the same full adder bitslice as in the previous ripple-carry adder design, but which wires up the carries differently. [This technique is known as **delayed addition**.]



   Here, A and B will be your normal two inputs to the adder. $S0$ and $S1$ together store the sum.

   (a) What is the latency of a single w-bit delayed addition?

   (b) How can the C input to the delayed adder be used?

   (c) Use these delayed-addition adders to build a spatial multipler. The two input-operands to the adder are in standard form. Output values are represented as two numbers (*i.e.* S0, S1 form shown above). Show the resulting, spatial multiplier which starts with numbers in standard form, but uses these delayed adders internally.

(d) What do you need to do to the multiplier output to convert the result back into normal form? (**Hint:** Remember, S0 and S1 jointly encode the final result) Can we further reduce the latency of the multiplier? (*i.e.* How do we implement the final format conversion efficiently?)

(e) What is the final area and latency of this multiplier?

3. Continuing to the use full-adder bitslice used above, wire them up as an associative reduce tree to compute the result of the multiplication from all the bit-wise partial-products ($a_i \wedge b_j$).

   (a) How many partial-product bits do you start with?

   (b) What reduction do you get with a single stage of full adders?

   (c) How deep is the full reduce tree?

   (d) Show the resulting multipler.

   (e) What is the final area and latency of this multiplier?

4. Write μcode to implement: $C = (A \times B)$. For the datapath provided (see end of assignment).

   (a) Don't worry about writing any special code for overflow. Just show the basic computation code for the multiply.

   (b) Turnin your microcode for multiplication

   (c) What is the area and latency for this multiplier (assume the datapath is widenned with w; give latency in the same units as before (not in cycles, but in time))?

   - Assume the ALU datapath limits cycle time.
   - Reason about the latency required for each of the operations the ALU provides (see table below).

5. Consider modifying the datapath from the previous problem so that it uses the delayed addition introduced above in place of normal addition/subtraction for all add-related operations in the ALU.

   (a) Describe how the datapath would need to change. (we are not asking for the implementation; just write text and, if appropriate, provide a diagram.)

   (b) What impact would this have on the datapath cycle time? (under what situations would this be beneficial?)

   (c) What is the area and latency for this multiplier?

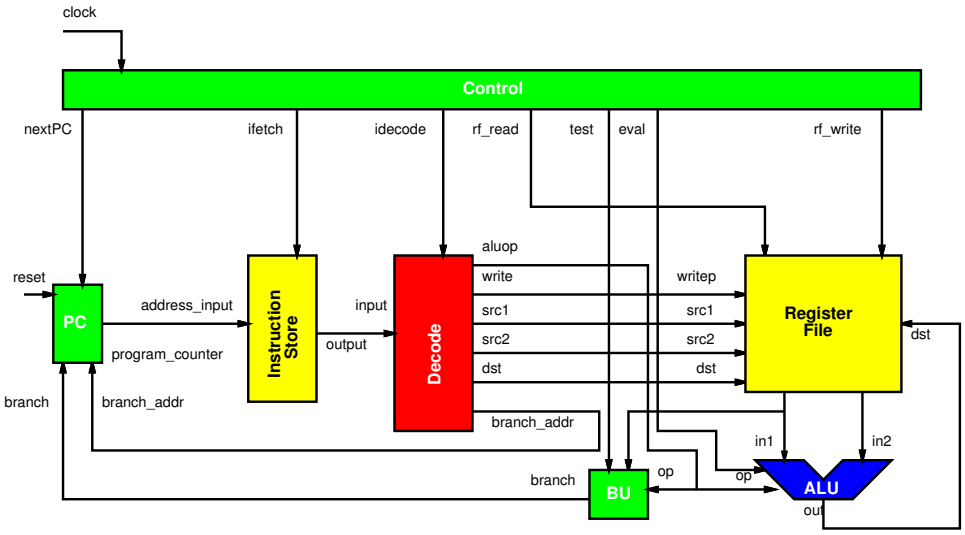
   - assume the datapath is widenned with w
   - assume the ALU datapath limits cycle time.
   - reason about the latency required for each of the operations the ALU provides (see table below).
   - give latency in the same units as before (not in cycles, but in time)); so think both about the number of cycles and the ALU cycle time.

- Your μcode may need to change to exploit this datapath change; you do not need to provide code for this case, but you will need to sketch out the implementation enough to justify your latency answer above.

6. Fillin the following table from your area/latency answers to the problems above:

Design	**Area**	**Latency**
P1: Ripple-Carry Based		
P2: Delayed-Addition Based		
P3: Associative Reduce Delayed-Addition		
P4: μcoded		
P5: μcoded using Delayed Addition		

Simple Branching Processor Datapath



The basic processor organization is as shown above. Non-branching instructions are of the form:

bits	13:10	9	8:6	5:3	2:0
field	op	w	src1	src2	dst

- op – operation to be performed (typically by ALU)
- w – write back ALU output to register file? (1=yes, 0=no)
- src1 – address of first ALU operand in register file
- src2 – address of second ALU operand in register file
- dst – address in register file into which the result should be sotred

For branch operations, the branch_addr is the low 6 bits of the instruction; that is, it is in the same place we would have placed src2 and dst in a normal, ALU operation.

bits	13:10	9	8:6	5:0
field	BNZ	0	src1	branch_addr

Generally, on each cycle the processor performs:

```
op,w,src1,src2,dst = instruction_store[pc]
...,branch_addr = instruction_store[pc]
in1=register_file[src1]
in2=register_file[src2]
if (w==1)
  register_file[dst]←(in1 op in2)
if ((op==BNZ) && (in1!=0))
  pc←branch_addr
else
  pc←pc+1
```

A special "done" operation indicates the computation is done and the program counter should stop incrementing. A reset signal tells the program counter to set its value to zero and begin computation.

The following ops are defined:

aluop	operation
ADD	dst← src1+src2
INV	dst← ∼(src1)
SUB	dst← src1-src2
XOR	dst← src1∧src2
OR	dst← src1—src2
INCR	dst← src1+1
AND	dst← src1&src2
BNZ	if (src1!=0) pc←branch_addr
SRA	dst← src1>>1; dst[31]=src1[31]
SRL	dst← src1>>1; dst[31]=0
SLA	dst← src1<<1
SLL	dst← src1<<1
DONE	stop execution