California Institute of Technology
Department of Computer Science
Computer Architecture

CS184a, Winter 2005 **Assignment 2: Space-Time Multiply** Monday, January 17

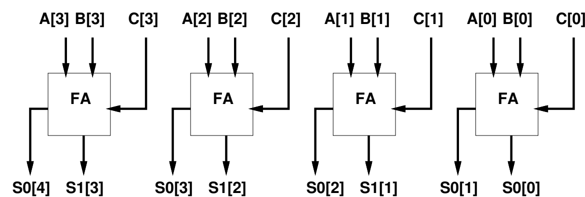
Due: Monday, January 17, 9:00AM

Everyone should do all problems.

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

- Give latency and area in terms of the operand bitwidth, w.
- When asked to draw an implementation or provide microcode, you may show the $w = 4$ case.

1. Show a $w \times w$ spatial multiplier built out of simple, ripple-carry adders.
   - What is the area and latency?
2. Consider using *delayed addition* within the spatial multipler. In delayed addition, the adders use the same full adder bitslice as in a ripple-carry adder. However, the carries are wired up differently.



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

   - What is the latency of a single addition?
   - How does the C input to the delayed adder get used?
   - Use these adders to build a spatial multipler. Show the resulting, spatial multiplier which starts with numbers in standard form, but uses these delayed adders internally.
   - What do you need to do on the output of the multiplier to convert the result back into normal form? What is the overall latency and area of the multiplier including this final return to standard form?

- How do you minimize the latency of this multiplier from input to final output in normal form.
- What is the final area and latency of this multiplier?

3. Combine the delayed adders which make up the previous multiplier with an associative reduce tree.
   - How deep is the reduce tree?
   - Show the resulting multipler
   - What is the final area and latency of this multiplier?

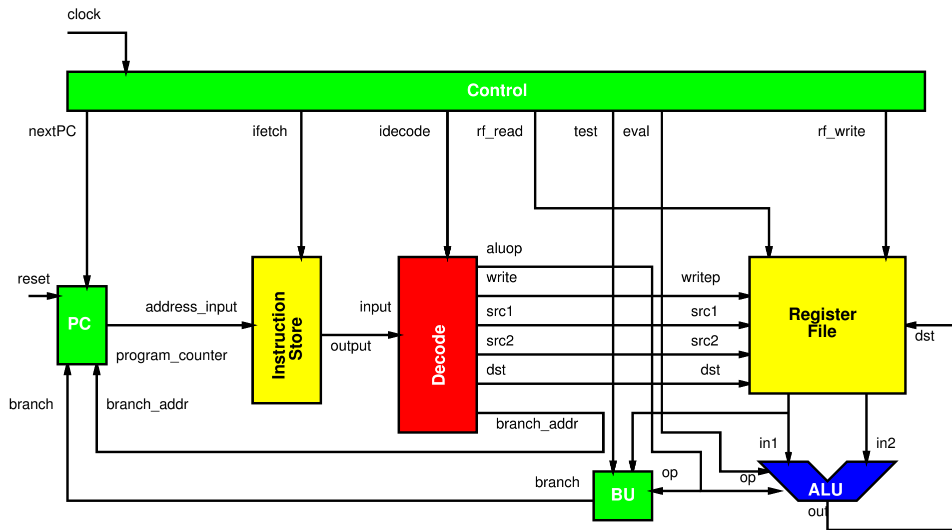
4. Write vertical μcode to implement: $C = (A \times B)$. For the datapath and simulator provided (see end of assignment and `/cs/courses/cs184/winter2005/arith_ucode/`).
   - Don't worry about writing any special code for overflow. Just show the basic computation code for the multiply.
   - Turnin your microcode for multiplication (show both a symbolic form and the assembled bits)
   - 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))

5. Consider modifying the datapath from the previous problem so that it uses delayed addition.
   - Describe how the datapath would need to change. (we are not asking for the implementation; just write text and, if appropriate, provide a diagram.)
   - What impact would this have on the datapath cycle time? (under what situations would this be beneficial?)
   - 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))
   - 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	encoding	operation
ADD	0x00	dst← src1+src2
INV	0x01	dst← ∼(src1)
SUB	0x02	dst← src1-src2
XOR	0x03	dst← src1∧src2
OR	0x04	dst← src1—src2
INCR	0x05	dst← src1+1
AND	0x08	dst← src1&src2
BNZ	0x0A	if (src1!=0) pc←branch_addr
SRA	0x0B	dst← src1>>1; dst[31]=src1[31]
SRL	0x0C	dst← src1>>1; dst[31]=0
SLA	0x0D	dst← src1<<1
SLL	0x0E	dst← src1<<1
DONE	0x0F	stop execution

A simple C simulator is provided for this datapath. `run.c` is the top level for the simple processor simulator. The processor is assembled and its cycle-by-cycle execution occurs in `branching_processor.c`. Each component of the processor has its own C file. The provided `Makefile` will build the `run` simulator.

The instruction format and encodings are defined in `instruction.h`.

`run` takes 3 arguments:

- instruction file (read)
- initial register file (read)
- final register file (written)

The instruction file and register file each consist of a series of lines with one hex number per line. The ith line, should contain the value for the ith entry in the instruction store and register file, respectively.

For example inputs, see `ex1.ibits` (instruction file) and `ex1.rbits` (initial register file). `ex1.asm` is an example showing symbolic comments to go with the raw instruction words.