

**6.004 Computation Structures  
Design Project**

**Instructions**

1. **MAKE SURE** you have the latest beta.uasm and project files.
2. optimize your Beta design as described below
3. using the checkoff file verify that your program operates correctly and submit it to the on-line checkoff system (just as for a Lab). There is just one checkoff file for this project. Completing the project will earn up to 10 points.

**Project Description**

For this project we're asking you to optimize the price/performance of your Beta design. The "price" is determined from the area of your circuit as reported by JSim at the end of each simulation run (it reports the size in square microns in the message area at the bottom of the netlist window). The "performance" is determined by the time needed to complete a set of benchmark programs. Benchmarking is the traditional, if somewhat unreliable, way of characterizing processor performance. The benchmark suite can be found in /mit/6.004/bsim/projcheckoff.uasm; in addition to the functional test run during Lab #6, it includes four benchmarks:

- Benchmark #1 makes two calls to a subroutine that performs an unsigned divide of its arguments, and then stores the quotients and remainders in main memory.
- Benchmark #2 makes two subroutine calls: one to do an in-place reverse an 11-element list, the second to compare the reversed list with an "answer" list to see if all went well.
- Benchmark #3 makes a copy of itself further up in memory and then jumps to the first location of the copy. The process is repeated 2 times.
- Benchmark #4 just performs a lot register arithmetic and writes the result to memory -- should be a slam dunk for pipelined and superscalar machines.

See projcheckoff.uasm for details about what values are written by each benchmark. The suite requires on the order of 1200 cycles to execute on a standard beta. Start by giving it plenty of time to run the benchmark; you'll recognize its completion by the 1-instruction BEQ loop at 0x44. You can then trim run time down to just get to this BEQ, to maximize your score.

Note: Long simulations of big circuits can produce *very* large history files, so it can take a while to prepare the waveform plots at the end of your simulation. Be patient and be modest in the number of signals you choose to plot.

Since your optimized design may operate differently than the Beta described in the Lab handouts, the checkoff file (/mit/6.004/jsim/projcheckoff.jsim) is a little different than the others we've seen so far this semester. It contains no test circuitry at all; you'll need to provide the main memory, waveform generators for CLK and RESET, etc. – whatever your optimized design needs to execute correctly. The only constraint on your netlist is that you must provide a 32-bit

wide, 1024 location main memory called Xmem to store the benchmark program and hold the results. The checkoff file checks certain locations of this memory to see that the benchmark has completed successfully. If your design passes verification, you can submit it to the on-line checkoff system which will report the number of points your design has received. The points are determined by the following formula:

$$\text{Benchmark}^{\circledR} = 1e-10 / (\langle \text{ending simulation time in seconds} \rangle * \langle \text{size of circuit in meter}^2 \rangle)$$

$$\text{Points} = (\text{Benchmark} - 10) / 2 \quad [\text{rounded; min} = 0, \text{max} = 10]$$

The smaller your circuit and the faster it completes the benchmark, the better the Benchmark. A good Beta design completed as described in the Lab handouts has a Benchmark of 15 or more (depending on various design choices) and would receive several points if submitted. So you'll need to do some modest innovation to max out on points – see the Hints section for some suggestions.

Your netlist should have the form:

```
// Design Project w/ beta implementation as described in lab handouts

.include "/mit/6.004/jsim/nominal.jsim"
.include "/mit/6.004/jsim/stdcell.jsim"
.include "/mit/6.004/jsim/projcheckoff.jsim"

// your optimized beta might have different terminals
.subckt beta clk reset irq ia[31:0] id[31:0] ma[31:0] moe mrd[31:0] wr mwd[31:0]
...
.ends

// matches .subckt above w/ IRQ tied to ground
Xbeta clk reset 0 ia[31:0] id[31:0] ma[31:0] moe mrd[31:0] wr mwd[31:0] beta

// your memory must also be called Xmem (so the checkoff code can find it!) but
// can have a different number of ports. It should however be initialized with
// the contents shown below -- the benchmark suite. A file containing these
// values can be found at /mit/6.004/jsim/projcheckoff.bin.
Xmem
+ vdd 0 0 ia[11:2] id[31:0]
+ moe 0 0 ma[11:2] mrd[31:0]
+ 0 clk wr ma[11:2] mwd[31:0]
+ $memory width=32 nlocations=1024
+ file="/mit/6.004/jsim/projcheckoff.bin"

// 20ns cycle time, assert RESET for first cycle. Your cycle time may vary..
Vclk clk 0 pulse(3.3,0,9.99ns,.01ns,.01ns,9.99ns)
Vreset reset 0 pwl(0ns 3.3v, 30ns 3.3v, 30.1ns 0v)

// Run the simulation for 1205 cycles. Your design might require more or less
// cycles depending on how it executes instructions. Run only as long as necessary
// to complete the benchmark since the ending simulation time is used to compute
// your circuit's Benchmark
.tran 24100ns
```

As mentioned in the netlist comments, your memory should be initialized with the binary code for the benchmark suite. A file with the appropriate values can be found at /mit/6.004/jsim/projcheckoff.bin – you can use the “file=...” argument to the \$memory component to load that file into your memory.

Your first step should be to incorporate your current Beta design into the netlist template shown above, and then adjust the timing of the CLK signal so that your circuit executes the checkoff code correctly. As usual you can click on the green checkmark after running a simulation to check your results. When you submit your results, the server will tell you your Benchmark and how many points you earned. You can submit as often as you'd like and the server will keep track of your best design to date.

Good luck! Using good design practices to implement the hints below can result in a Benchmark over 30. If you get close to this, congratulations! But don't be discouraged if your design doesn't make it on the first try – it takes some practice to get the knack of making circuits both fast and small. If you achieve a great Benchmark, let us know and come join the 6.004 staff ☺.

### Hints for size

1. The benchmark **DOESN'T USE the multiply instructions**; so the first thing you should do is to axe your multiplier, if you have one, and take credit for the size reduction. The benchmark *does* require a 1024-word memory, however; don't bother trying to reduce its size.
2. The biggest single size reduction comes from eliminating one or more ports on the multi-port main memory. Ideally, you'd like to get it down to a single port. To do this you'll have to multiplex the memory between fetching instructions and doing data accesses. You might, for example, fetch instructions during the first half of each cycle and do data accesses (if necessary) during the second half. You can use a MUX to select between the instruction and data addresses, and use tristate drivers to send write data to the memory's data pins (the memory has these built-in for sending data back to the processor – that's what the MOE signal has been controlling). See Appendix 2 of Lab #6 for details on how the memory operates.
3. Use logic gates instead of a ROM to generate the necessary control signals. Even if you do this for only some of the control signals, you'll see a noticeable reduction in circuit size. Logic can also be faster than a ROM for control signals in your critical path.
4. Design an incrementer circuit for adding 4 to the PC instead of using a regular 2-input adder. This is easy to do – just think about an adder where the second input is the constant 4 (i.e., all but one of the input bits is zero!) and then eliminate/simplify the adder logic appropriately.
5. See if you can eliminate the separate adder used to calculate the new PC for branch instructions – can you figure out a way to use the ALU instead? Hint: you'll need a more complicated BSEL MUX circuit that can send either  $sxt(literal)$  or  $4*sxt(literal)$  to the second input to the ALU. If you also pipeline your circuit remember that using the ALU for branch-offset arithmetic might increase the number of branch delay slots.

### Hints for speed

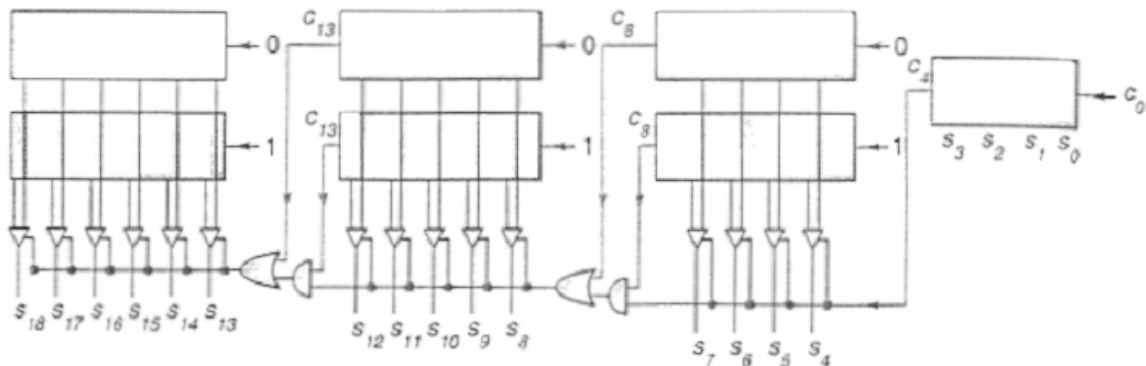
1. *minimize load-dependent delays.* As you connect the output of a gate to additional inputs, its capacitance increases and so changes in the signal value take longer (see the  $t_R$

and  $t_F$  columns in the standard cell table on page 2 of Lab #3). Heavily loaded signals should be buffered to reduce the total delay. There are several strengths of INVERTERS and BUFFERS available for driving different sized loads.

2. *use inverting logic.* NAND, NOR, etc. gates are noticeably faster than their non-inverting counterparts (AND, OR, ...). Rearranging logic to use inverting gates can make a big difference performance along the critical path of your circuit. Positive logic gates are included in the standard cell library because they are often smaller than their inverting logic equivalents and thus may be useful for implementing logic that is not on the critical path.
3. *minimize delay along critical paths.* Look at the device and cycle for which the minimum observed setup was reported – see Appendix 1 of Lab #6 to learn how to do this. By tracing back through the circuitry that lead to this *critical path* you should get some good ideas about where your circuit could use a speed boost. It's often possible to rearrange your logic to reduce the number of gate delays along the critical path, sometimes at the cost of additional gates elsewhere but that's a tradeoff which is often worthwhile. Using the “complex” functions provided by the AOI21 and OAI21 gates could be quite helpful in this regard. In a ripple-carry adder, you want to minimize the CIN to COUT delay when trying to optimize overall circuit performance.
4. *use a different adder architecture.* Consider using an architecture that avoids rippling the carry through all 32 bits of the adder: carry-select and carry-lookahead are two techniques that are described in more detail below.
5. An unpipelined Beta makes two accesses to the relatively slow main memory (4ns access time for the SRAM-sized memory we've been using). Even a two-stage pipeline (instruction fetch and everything else) can come close to cutting the cycle time in half. A pipelined design is fun but a lot of work, so try the other hints first – they should be sufficient to max out on points.

### Appendix 1: Carry-select adders

Idea: do two additions, one assuming the carry-in is 0, the other assuming the carry-in is 1. Use a MUX to select the appropriate answers when the correct carry-in is known:

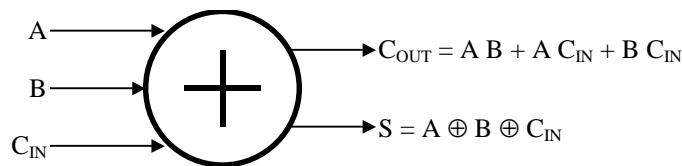


Blocks on the left can include more bits since there is more ripple time while waiting for the select signal to arrive.

With one stage (i.e., using carry-select on the top sixteen bits of your adder driven by the carry-out from the bottom 16-bits) the circuit is 50% larger but almost twice as fast as ripple-carry adder. With multiple, variably-sized blocks,  $t_{PD}$  approaches  $\sqrt{N}$  where  $N$  is the number of bits in the operand. Carry-select can be combined with carry-lookahead (see below) for even greater performance improvements.

## Appendix 2: Carry-lookahead adders

The basic building block used in the ripple-carry adder is the **full adder**, the 3-input, 2-output combinational logic circuit show below:

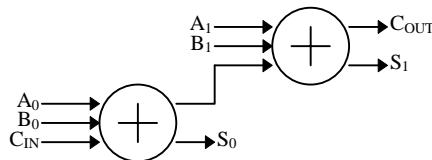


Since the  $C_{OUT}$  signals of each full adder are in the critical path, let's see if we can't improve the adder's performance by generating the  $C_{OUT}$  signals more quickly. First let's rewrite the equation for  $C_{OUT}$ :

$$C_{OUT} = A B + A C_{IN} + B C_{IN} = A B + (A+B) C_{IN} = G + P C_{IN}$$

where  $G = A B$  is true if a carry is *generated* by the full adder and  $P = A + B$  is true if the carry is *propagated* by the full adder from  $C_{IN}$  to  $C_{OUT}$ . Actually the propagate signal is sometimes defined as  $P = A \oplus B$  which won't change the computation of  $C_{OUT}$  but will allow us to express  $S$  as a simple function of  $P$  and  $C_{IN}$ :  $S = P \oplus C_{IN}$ . Note that  $P$  and  $G$  depend only on  $A$  and  $B$  and *not* on  $C_{IN}$ .

We can generalize the notion of  $P$  and  $G$  to blocks of several bits. For example, consider a two-bit adder:



Define the block generate signal  $G_{0,1}$  as  $G_{0,1} = G_1 + G_0 P_1$ , i.e., the 2-bit block will generate a carry if a carry is generated in bit 1 ( $G_1$ ) or if a carry is generated in bit 0 and propagated by bit 1 ( $G_0 P_1$ ). Similarly we can define the block propagate signal  $P_{0,1}$  as  $P_{0,1} = P_0 P_1$ , i.e.,  $C_{IN}$  will be propagated to  $C_{OUT}$  only if both bits are propagating their carry-ins.

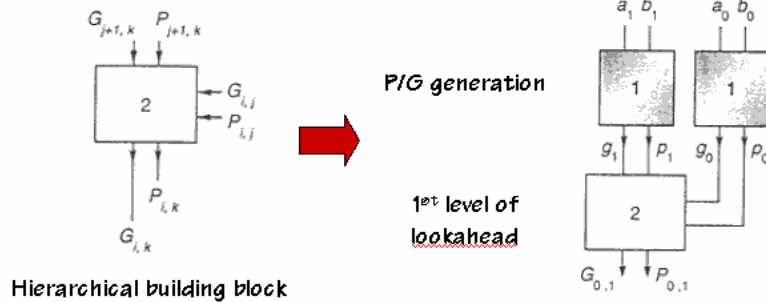
Building on this idea, we can choose the maximum fan-in we want for our logic gates and then build a hierarchical carry chain using these equations:

$$C_{J+1} = G_{IJ} + P_{IJ}C_I$$

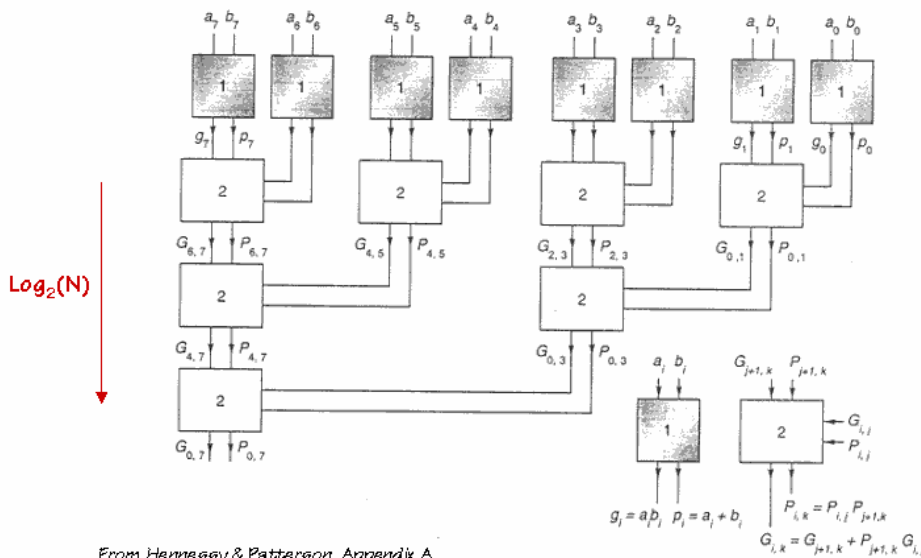
$$G_{IK} = G_{J+1,K} + P_{J+1,K} G_{IJ}$$

$$P_{IK} = P_{IJ} P_{J+1,K} \quad \text{where } I < J \text{ and } J+1 < K$$

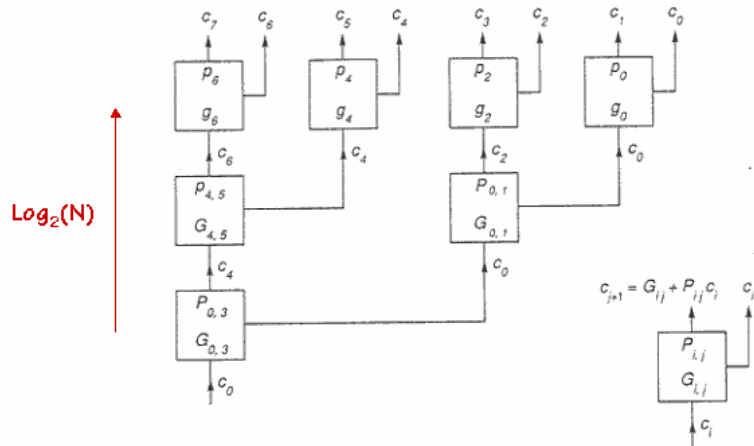
"generate a carry from bits I thru K if it is generated in the high-order (J+1,K) part of the block or if it is generated in the low-order (I,J) part of the block and then propagated thru the high part"



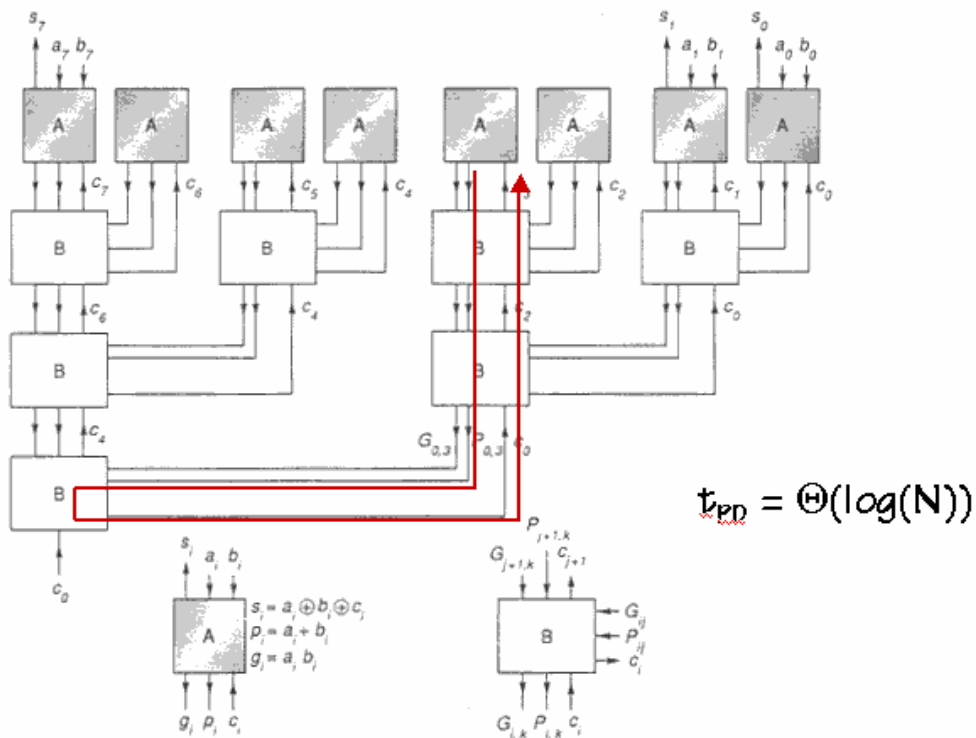
We can use this building block to construct P/G signals for any width operands. The following figure shows how this works for 8-bit operands using 2-input logic gates:



We can then use the P, G and carry-in signals at each level to generate carry-in signals for previous levels of the hierarchy, as shown in the following diagram for an 8-bit adder:



We can combine P/G generation and carry generation into a single building block, leading the following diagram for a complete 8-bit carry-lookahead adder:



In order to use only inverting logic in the building blocks, it's common to have two versions: one that takes inverted inputs and produces non-inverted outputs, and another that takes non-inverted inputs and produces inverted outputs. These blocks can be used on alternative levels of the carry lookahead tree.