The Codesign Challenge

Objectives

In the codesign challenge, your task is to accelerate a given software reference implementation as much as possible. You can use any of the previously discussed techniques to accelerate the implementation: optimize the software, build a coprocessor, optimize the hardware/software communication. The constraints of your implementation are

1. that it must be completed by 11/30/2009 at 5:00PM;
2. that it must run correctly on the Spartan 3E starter kit;
3. and that it follows the given testing procedure to demonstrate the performance of your implementation.

All solutions will be strictly ranked and graded accordingly. Thus, there will be a single 'best' solution, a single 'second' solution, and so on. There is no perfect solution; the only 'perfect' solution is the one that ranks higher than all the others. The quality of your solution will be evaluated using the following criteria:

1. The resulting clock cycle count of your implementation, with a ‘clock cycle’ corresponding to one tick of a PLB Timer module clocked at 50MHz.
2. The product of the Slice Usage (Spartan 3E slices) and the Clock Cycle Count.
3. The time when the solution was turned in (before the deadline, but earlier is better).

The clock cycle count is most important, the area-time product is second-most important, and the design time is third-most important. Faster (but correct) designs will always win. For solutions which have a cycle count within 5% of each other, the area-time product (item 2) will be used as a distinctive factor. In case two solutions are within 5% of each with respect to item 1 as well as item 2, the time of posting the solution will be used to resolve the ranking of the two designs.

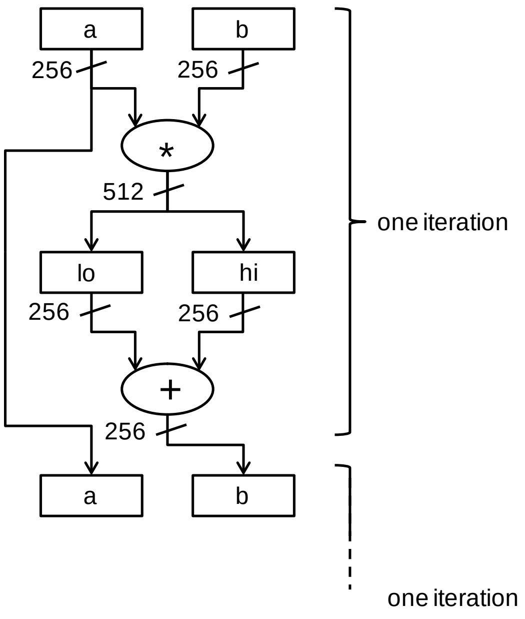
Hence, it is in your interest to find the highest possible performance that can still be accommodated on a Spartan3 board, and to find that solution as quickly as possible.

Assignment: Monster Multiplier Madness

The task is to create a very efficient implementation of the Monster Multiplier Madness (MMM) algorithm. This algorithm is specified as:

```
widemagic(input a, input b, output lo) {
  loop 16384 times {
    (hi,lo) = a * b;
     b = hi + lo;
  }
}
```

In this algorithm, the variables `a`, `b`, `hi` and `lo` are unsigned 256-bit numbers. The algorithm multiplies two 256-bit operands, `a` and `b`, and produces a 512-bit result. `hi` is the upper 256-bit, while `lo` is the lower 256-bit of the result. The algorithm then adds the upper part and the lower part together, and uses that as an input to the next iteration.

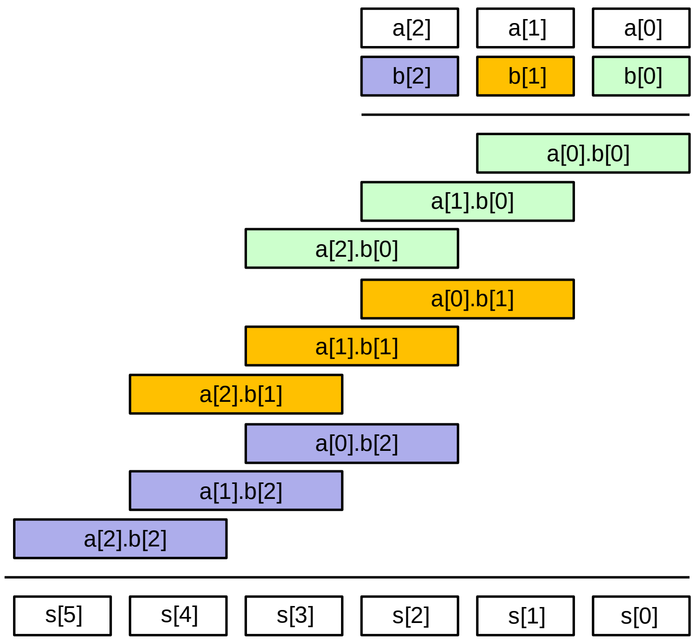


The reference software implementation implements this algorithm on a 32-bit Microblaze processor. In that case, one must implement so-called *multi-precision arithmetic* operations. Here is an example how a 256-bit addition can be implemented on a 32-bit processor.

```
void add32(unsigned a,
           unsigned b,
           unsigned *q,
           unsigned *y) {
  unsigned max = (a > b) ? a : b;
  *q += (a + b);
  if (*q < max)
    *y = 1;
  else
    *y = 0;
}

void wideadd(unsigned a[8],
             unsigned b[8],
             unsigned t[8]) {
  unsigned i, cy;

  for (i=0; i<8; i++)
    t[i] = 0;

  cy = 0;
  for (i=0; i<8; i++) {
    t[i] = cy;
    add32(a[i], b[i], &(t[i]), &cy);
  }
}
```

The procedure `wideadd` makes use of a 32-bit addition procedure, `add32`, that can detect a carry bit. Using this addition, a wide addition is iteratively implemented from the least significant word to the most significant word, while the carry bit is propagated.

The multiplication of 256-bit is quite interesting as well, because there are multiple ways to do it. We illustrate just one of them, the 'pencil and paper' method.



In this example, we assume two multiprecision arguments of just 3 words rather than 8. Using each word, we than calculate all partial products: a[0].b[0], a[1].b[0], a[2].b[0], and so forth. Note that the wordlength of a partial product will be twice as big as that of an operand word. For example, on a 32-bit processor, the operand words are 32 bit and the partial products are 64 bit. After the partial products are known, we accumulate all of them into the final multiplication result. For two multiprecision arguments of 3 words, the multiplication result will be 6 words.

The following is an implementation of the multi-precision multiplication in pseudocode. C and S are 32-bit unsigned integers; the assignment to (C,S) is an assignment of a 64-bit unsigned integer with C the upper part and S the lower part.

```
for i = 0 to W-1 do {
  C = 0
  for j = 0 to W - 1 do {
    (C,S) = s[i+j] + a[j] * b[i] + C
    t[i+j] = S
  }
  t[i+W] = C
}
```

An implementation of this multiplication algorithm in C is as follows. The following helper routines are used: `add32` performs a 32-bit addition with carry generation. `mult64` multiplies two 32-bit words into a 64-bit result. `inc64` increments a 64-bit number represented as two 32-bit words.

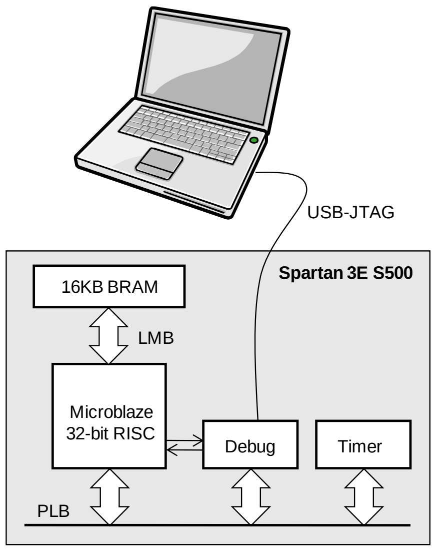
```
void widemult(unsigned a[W],
              unsigned b[W],
              unsigned t[2*W]) {
  unsigned i, j, C, S;
  unsigned hi, lo;
  unsigned q, cy1, cy2;

  for (i=0; i<2*W; i++)
    t[i] = 0;

  for (i=0; i<W; i++) {
    C = 0;
    for (j=0; j<W; j++) {
      mult64(a[j], b[i], &lo, &hi);
      add32(lo, C, &q, &cy1);
      add32(t[i+j], q, &S, &cy2);
      C = hi;
      if (cy1) inc64(&S, &C);
      if (cy2) inc64(&S, &C);
      t[i+j] = S;
    }
    t[i+W] = C;
  }
}
```

Using the wideadd and widemult routines, we can now implement the MMM algorithm as follows:

```
void widemagic(unsigned a[W],
               unsigned b[W],
               unsigned t[W]) {
  unsigned tmp[2*W], btmp[W];
  unsigned i;
  for (i=0; i<W; i++)
    btmp[i] = b[i];
  for (i=0; i<(16384); i++) {
    widemult(a, btmp, tmp);
    wideadd(tmp, &(tmp[W]), btmp);
  }
  for (i=0; i<W; i++)
    t[i] = tmp[i];
}
```

The reference implementation of this algorithm, using only Microblaze software, takes 280 million clock cycles to complete. Your task is to create an optimized implementation of the reference design that decreases the execution time as much as possible.

The reference architecture for the design includes the following elements:



- A 32-bit Microblaze processor with 16 Kilobyte instruction- and data-memory,
- An on-chip timer
- A debug module

This baseline design occupies 30% of the slices of the Spartan FPGA, 40% of the on-chip RAM modules, and 15% of the on-chip multiplier modules.

The testbench for this design takes the following steps:

1. It initializes two 256-bit multi-precision operands a and b.
2. It performs the MMM algorithm using the reference software, and records the execution time with the timer.
3. It performs the MMM algorithm using your optimized implementation, and records the execution time with the timer.
4. It compares the result of the reference MMM implementation with your MMM implementation.
5. It prints the cycle count of both executions.

The speedup of your design is defined as:

```
speedup = (cycles_reference_MMM) / (cycles_your_MMM)
```

Obviously, the objective is to find the highest possible speedup.

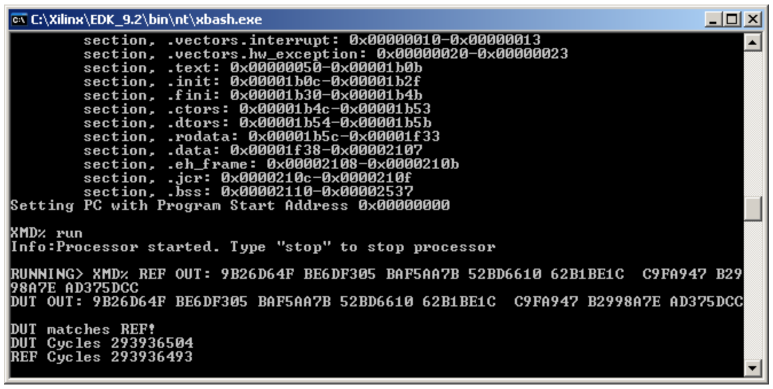
You can use any possible trick to improve the execution speed. Some examples are as follows.

- Adding coprocessors to the PLB or FSL bus, to improve the multiplication speed. Note that a 256*256 bit multiplier requires twice the number of logic resources of your FPGA.
- Optimize the software, using source code transformations or compiler flags.
- Re-allocating memory.

However, you need to stick with the standard testbench, and you cannot tweak the testvectors. The resulting design must also fit onto your Spartan 3E starter kit. If you are in doubt if a tweak is allowed or not, send an email or post a question on the bulletin board.

How to start

On Blackboard, download the baseline reference implementation. This design will run directly on your Spartan kit. Start by studying the reference implementation software. This reference implementation calls `yourmagic()`. Running the design on your board will show you the following result.



```
        section, .vectors.interrupt: 0x00000010-0x00000013
        section, .vectors.hw_exception: 0x00000020-0x00000023
        section, .text: 0x00000050-0x00001b0b
        section, .init: 0x00001b0c-0x00001b2f
        section, .fini: 0x00001b30-0x00001b4b
        section, .ctors: 0x00001b4c-0x00001b53
        section, .dtors: 0x00001b54-0x00001b5b
        section, .rodata: 0x00001b5c-0x00001f33
        section, .data: 0x00001f38-0x00002107
        section, .eh_frame: 0x00002108-0x0000210b
        section, .jcr: 0x0000210c-0x0000210f
        section, .bss: 0x00002110-0x00002537
Setting PC with Program Start Address 0x00000000

XMD% run
Info:Processor started. Type "stop" to stop processor

RUNNING> XMD% REF OUT: 9B26D64F BE6DF305 BAF5AA7B 52BD6610 62B1BE1C  C9FA947 B29
98A7E AD375DCC
DUT OUT: 9B26D64F BE6DF305 BAF5AA7B 52BD6610 62B1BE1C  C9FA947 B2998A7E AD375DCC

DUT matches REF!
DUT Cycles 293936504
REF Cycles 293936493
```

Hints and Tips

- If you build a coprocessor, it is highly recommended to **construct a cosimulation model** of your design using GEZEL. While you can develop coprocessor hardware directly in VHDL, it will require you to take care of many details at once. Going through cosimulation first enables you to test your idea before taking it to the board.
- Also, when developing hardware, **first test your ideas on 'small' designs.** When the low level components work fine, next verify how well it scales up.
- Also, carefully **consider tradeoffs**. There are obviously many more implementation alternatives than you have time to try out. Thus, you will have to think before you implement, and to experiment to find the largest acceleration as quickly as possible.
- Always **focus on the bottleneck** in the overall system. Remember the earlier examples we discussed. Hardware parallelism is useless if you cannot provide data sufficiently fast.
- Also, **make use of your homework assignments/solutions** to see examples on how a memory-mapped interface or an FSL interface can be created.

What to turn in

By the deadline, post the following information on Blackboard.

- A short report (no more than 4 pages) that summarizes the main characteristics of your design. Your report must at least contain the following table.

Area of the baseline design (slices)	
Performance of the baseline design (cycles)	
Area of the optimized design (slices)	
Performance of the optimized design (cycles)	

In addition, you are encouraged to discuss the trade-offs you made, to provide a block-diagram of the resulting system, to describe the architectural features of the hardware coprocessor you made. Also include a screenshot of the design as it executes in XMD.

- If you developed a cosimulation model in GEZEL, please provide the cosimulation model (C driver and FDL file).

- The optimized implementation in XPS. Before posting the design on Blackboard, make sure you run **Project->Clean All Generated Files**. Then, zip the project directory and post it on Blackboard.

Grading

Your design will be graded as follows:

- Your ranked performance counts for 75% of the grade
- Your report counts for 25% of the grade.

Your cosimulation model and XPS project will be run to verify the correctness of the statements you make in the report.

The performance ranking criteria described above will be used. Having a working solution is *not sufficient* to obtain a good grade. Having a speed improvement of, for example, 3 times, is *not sufficient* to obtain a good grade. The best grade will go to the design with the highest performance. All other designs will be strictly ranked according in relation to the best one. This strict ranking rule is introduced based on the observation that, under free market conditions, better designs have a better chance to make it into a product.

However, don't let this rule spoil the fun.

This is your chance to explore new ideas and to try out what you have learned in this class! We will discuss the design in detail in the class of 2 December.

Good luck!