ECE4530 Fall 2011:
Codesign Challenge
Bit-Matrix Multiplication on a DE2-115 FPGA Board

Assignment posted on 10 November 8AM
Solutions due on 29 November 8AM

The Codesign Challenge is the final assignment in ECE 4530. This project is an exercise in performance optimization: you will start from a given reference application on a Nios-II processor. You have to improve the performance of the reference application as much as possible, using the hardware/software codesign techniques covered in this course. Typically, you would design a hardware coprocessor. In addition, you would also optimize the driver software, and/or modify the system architecture. There is an important constraint: the final result must execute on the DE2-115 board under a predefined performance-evaluation testbench. The testbench is written in C, so you have to keep a Nios-II processor present in the final implementation.

Application: Bit-matrix multiplication

The application is bit-matrix multiplication (BMM): a matrix multiplication of Galois-Field elements of the field $GF(2)$. Given an N-by-N matrix A with elements `a[i,j]`, and an N-by-N matrix B with elements `b[i,j]`. The matrix multiplication of A and B is a matrix C with elements `c[i,j]` defined as follows.

```
for i = 1 to N do
  for j = 1 to N do
    c[i,j] = 0;
    for k = 0 to N do
      c[i,j] = c[i,j] ^ (a[i,k] & b[k,j])
```

Note that the element-addition and element-multiplication are $GF(2)$ operations: bitwise-xor is used for addition, and bitwise-and is used for multiplication. As an example, here is a two-by-two BMM.

```
  [ 1 1 1 ]   [ 0 1 1 ]     [ 1 1 0 ]
  [ 0 1 0 ] x [ 1 0 0 ]  =  [ 1 0 0 ]
  [ 1 0 0 ]   [ 0 0 1 ]     [ 0 1 1 ]
```

BMM is used to implement Galois-field multiplications (used in cryptography and channel coding) as well as for bit manipulations. BMM has been implemented as an instruction on a Cray supercomputer.

In the Codesign Challenge, you will implement a BMM for matrices of dimensions 16X16, and we will denote the operation for this specific dimension as BMM-16. A straightforward analysis of the algorithm described above indicates that BMM-16 produces a matrix with 256 bits, and that each bit is computed with 16 and operations and 15 xor operations.

Representation in memory

On a 32-bit RISC processor, the bit-matrix will be stored in a compact format, utilizing all bits of a 32-bit word. The bits of the matrix will be stored in a row-wise format, filling the 32-bits of a word from the msb to the lsb. Since there are only 16 bits per row for BMM-16, each 32-bit word can store two rows. If we express the matrix bit from row `i`, column `j` as `b(i,j)`, we can map the 256 bits of the BMM-16 matrix into an array `d[]` of 8 words as follows.

```
d[0] = [b( 1,1) b( 1,2) ... b( 1,16) b( 2,1) ... b( 2,16)]
d[1] = [b( 3,1) b( 3,2) ... b( 3,16) b( 4,1) ... b( 4,16)]
...
d[7] = [b(15,1) b(15,2) ... b(15,16) b(16,1) ... b(16,16)]
```

BMM-16 in C

The following two C procedures illustrate how a bit `b(i,j)` can be read or written. Note that these procedures assume the matrix starts at row 0, column 0 (instead of the more conventional row 1, column 1).

```
// read bit (i,j) from bm[8]
unsigned readbit(unsigned i,
                 unsigned j,
                 unsigned bm[8]) {
  // i, j: 0 .. 15
  return ((bm[((i&15) >> 1)] >> (31 - (16*(i&1) + j))) & 1);
}

// write bit (i,j) from bm[8]
void writebit (unsigned bit,
               unsigned i,
               unsigned j,
              unsigned bm[8]) {
  // i, j: 0 .. 15
  if (bit&1)
    bm[((i&15) >> 1)] |=  (1 << (31 - (16*(i&1) + j)));
  else
    bm[((i&15) >> 1)] &= ~(1 << (31 - (16*(i&1) + j)));
}
```

Using these two procedures, the BMM-16 can now be easily expressed as follows.

```
void bmm_sw(unsigned cm[8],
            unsigned am[8],
            unsigned bm[8]) {
  unsigned i, j, k;
  for (i=0; i<16; i++) {
    for (j=0; j<16; j++) {
      writebit(0, i, j, cm);
      for (k=0; k<16; k++) {
        writebit(readbit(i, j, cm) ^
                 (readbit(i, k, am) &
                  readbit(k, j, bm)), i, j, cm);
      }
    }
  }
}
```

The assignment for the Codesign Challenge is to accelerate this function as much as possible, using hardware-software codesign techniques. It's clear from the C code that the software performance of BMM-16 won't be very good. There are an extensive amount of bit-manipulation operations to read or write single bits of a word. On the other hand, the hardware cost of BMM is not very high, even when the entire operation is executed in parallel. Hence, a possible hardware/software-codesign solution would be to transfer the two 8-element arrays `am` and `bm` form software to a hardware coprocessor, compute the resulting array `cm`, and transfer it back to software.

Reference Design

The reference design for the codesign challenge includes the following elements.

- A GEZEL testbench, with a software driver (`bmmdriver.c`) and a hardware platform containing a single processor (`bmmcoproc.fdl`). The performance of the design is reported in terms of the simulated processor clock cycles (StrongARM).
- A software driver suitable for a Nios-II processor, and a Quartus system (`challenge.qar`) derived from the DE2-115-BASIC-COMPUTER design. This design includes a Nios-II/e, various peripherals, and on off-chip SDRAM interface. The performance of the design is reported in terms of clock cycles of the 50MHz system clock. In the reference implementation, all sections of the code are mapped into SDRAM (off-chip).

Based on your experience with earlier homework, you should be able to run the GEZEL design, as well as build the Quartus reference system. In a nutshell, the following steps will implement the reference design on DE2-115.

- Restore the archived project file, `challenge.qar`
- Compile the resulting design in Quartus, and obtain a bitstream (sof file).
- Open a Nios-II command shell. Go to the project directory. Configure the sof file in the DE2-115 board with `nios2-configure-sof DE2_115_Basic_Computer.sof`.
- Go to the `software` subdirectory. Run `nios2-bsp-editor` and open the `settings.bsp` file in the `hal_bsp` subdirectory. Click generate to generate the BSP configuration file. Exit the BSP editor.
- Generate the BSP source code using `nios2-bsp-generate-files --settings=hal_bsp/settings.bsp --bsp-dir=hal_bsp`.
- Compile the BSP with `cd hal_bsp; make; cd ..`
- Generate an application makefile with `nios2-app-generate-makefile --bsp-dir=hal_bsp --src-files=bmmdriver.c --elf-name=bmmdriver.elf`
- Open a second Nios-II command shell and run `nios2-terminal`
- In the first Nios-II command shell, download the ELF file with `nios2-download bmmdriver.elf --go`
- In the second Nios-II command shell, you should now see the reference design execute. The reference code takes around 64 million clock cycles to execute BMM-16.

Make sure you understand all of the above commands; the have been explained in the previous homework.

Testbench

The testbench consists of two phases: functional verification and performance evaluation. In each case, two functions are compared:

- `void bmm_sw(unsigned cm[8], unsigned am[8], unsigned bm[8])`: The reference implementation in C, as described above.
- `void bmm_hw(unsigned cm[8], unsigned am[8], unsigned bm[8])`: The driver for your hardware-accelerated version of BMM-16.

The functional verification phase will generate 5 random matrices A and B, and compare the result of BMM-16 as computed by `bmm_sw` and `bmm_hw`. The performance evaluation phase will record the cycle count for 5 successive executions of `bmm_sw` and `bmm_hw`, and report the average cycle count and the speedup (cycle count of `bmm_sw` divided by the cycle count for `bmm_hw`).

Your task is to deliver a design which delivers the same functionality as `bmm_sw`, but which is accelerated as much as possible. You will need to document your design per guidelines given below.

Ranking Criteria

All designs will be strictly ranked from best to worst. This section defines what 'best' means. We will use the following metrics to evaluate the rank of your design.

- Functional Correctness: This requirement is mandatory for all designs. Your design has to work, in order to be considered for ranking. If it does not work, you will be automatically ranked last.
- Metric 1: The cycle count of `bmm_hw`, as executed on the DE2-115 board, measured with a clock frequency of 50MHz. A lower cycle count is better.
- Metric 2: The area of the resulting design, expressed in terms of 'Total Logic Elements' of a Cyclone IV FPGA. A lower LE cell count is better.
- Metric 3: The turn-in time of your design and report, as measured by the turn-in time on Scholar. Turning in the solution earlier is better. Note that, if you turn in the design multiple times, only the latest turn-in time will be used.

Two designs will be compared as follows, to determine their ranking order. If a design is functionally not correct (i.e. fails the testbench), it will automatically moved to the last rank. If multiple designs are not operational, they will all share the same lowest rank. All functionally correct designs will get a better and unique rank.

- First, their cycle count (Metric 1) will be compared. If there is a difference of more than 5% between them (with the fastest design considered 100%), the fastest design will get a better rank. If, one the other hand, the difference is smaller than 5%, Metric 2 will be used as tie-breaker.
- Metric 2 will be used in a similar way to compare two designs. If the difference in area between two designs is larger than 5%, then the smallest design gets the better rank. Otherwise, if they are separated less than 5%, Metric 3 will be used as tie-breaker.
- Metric 3 will be used as a final metric in case the ranking decision cannot be made using Metric 1 and Metric 2 alone. In this case, the design turned-in earlier wins.

The grade for your project is determined as follows.

- 70 points of the grade are determined by the ranking of your project as described above. The best design gets 70 (out of 70) points, the worst design gets 30 (out of 70) points, and all other designs are linearly distributed between 30 and 70 points.
- 30 points of the grade are determined by the quality of the written documentation you provide with the solution. The requirements are listed below.

How to start

In order to design an efficient implementation for BMM-16, you will have to start by looking at the overall problem from a system-level perspective. What is limiting the performance for the reference implementation, and how would the ideal optimization look like? Think about the problem in terms of high-level operations, not in terms of detailed statements of the reference C code.

Another important decision is to define the proper hardware acceleration strategy. We discussed at least two different approaches for it: one using custom-instructions on the Nios-II, and a second one using memory-mapped registers.

The following recent publication provides useful background in optimization strategies. It can be easily found online.

- Hilewitz, Y., Lauradoux, C., Lee, R.B., "Bit Matrix Multiplication in Commodity Processors", Proceedings of 19th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 08), July 2008.

Finally, it's also a good idea to work incrementally. Don't do everything at once: developing a hardware accelerator, generating or writing VHDL code, mapping it on the FPGA, and testing it with the driver software. Instead, work in small steps. Use simulation before doing synthesis: use GEZEL or Modelsim simulation to verify your design. Try to develop a smaller BMM-4 of a BMM-16 is too large at first. Debug your driver software separately from the coprocessor: demonstrate first that you can transfer data to/from the hardware before you insert your entire design in the loop.

What to turn in

You have to deliver the following items for the project result:

- A qar file of your final design. It must contain a Nios processor that is capable to execute the evaluation testbench. Make sure it is a complete system, and, pay particular attention to the following items.
  - Include the driver software for the resulting system as well. By default, the software is not included as part of a qar file - you have to include it by hand.
  - Include the bitstream (sof file) for your design. This can be selected as an option during qar file generation.
- Any GEZEL code that you've developed. It is not mandatory to use GEZEL coding, and you may directly code your design in VHDL or Verilog. However, if you have not done HDL coding before, it's strongly recommended to develop your coprocessor in GEZEL, and follow one of the methods explained in the previous Homework.
- A PDF document that describes your resulting design. Your report must include, at the very least, the resulting cycle count for your `bmmdriver_hw()`, and a screen shot of a NiosII terminal to prove your claim. Additionally, please explain your design strategy, the architecture of your hardware/software solution, and overall observations on the design.

Good Luck!