SystemVerilog Design of an Embedded Processor

Introduction

This exercise is done individually and the assessment is:

- By formal report describing the final design, its development, implementation and testing.
- By a laboratory demonstration of the final design on an Altera FPGA development system

Specification

The objective of this exercise is to design an n-bit implementation of picoMIPS.

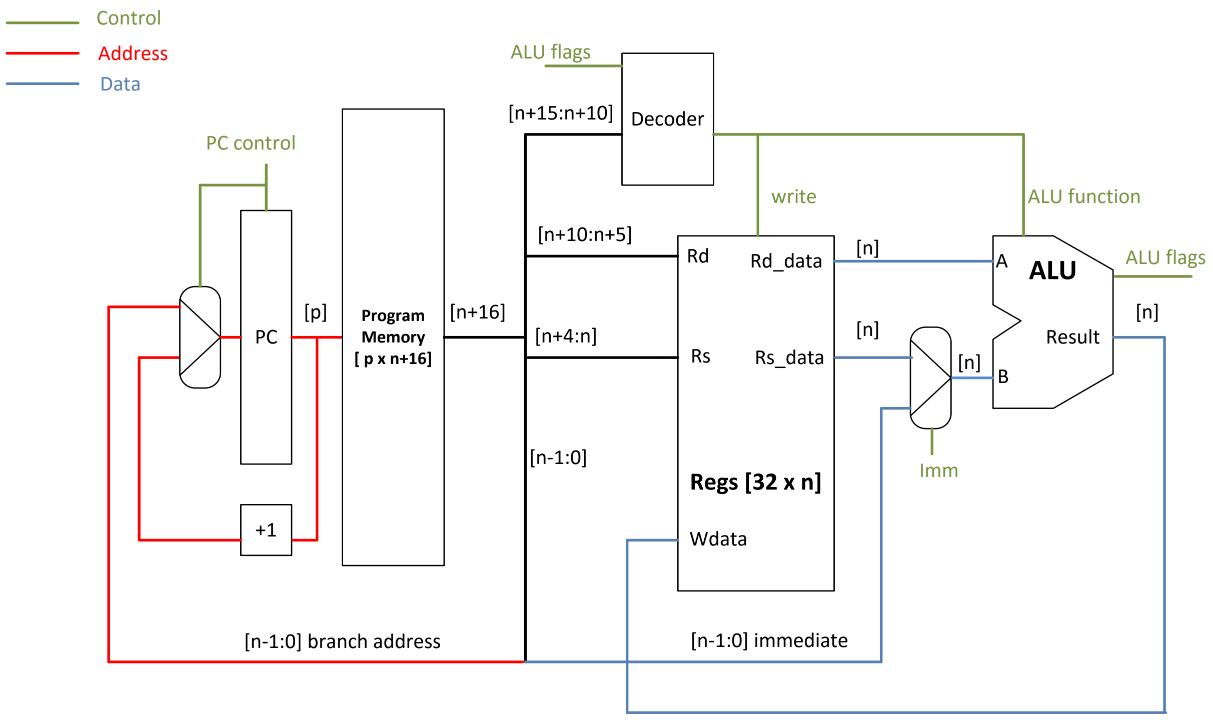

Figure 1. picoMIPS, version without RAM.

The design should be as small as possible in terms of FPGA resources but sufficient to implement the affine transformation algorithm described below. The size cost function of the design is defined as follows:

Cost = number of Logic Elements used + max(0,number of embedded multipliers used – 2) +30 x Kbits of RAM used

Each Logic Element has a flip-flop hence flip-flops are included in the above cost figure. The cost figure should be calculated for Altera Cyclone IV EP4CE115 and should be as low as possible. Altera Cyclone IV has 266 18x18 bit embedded hardware multipliers. If embedded multipliers are used in your implementation, up to two of them are ‘free’, i.e. they do not contribute to the size cost. To demonstrate the cost figure of your design show in your report Altera Quartus synthesis statistics for Cyclone IV EP4CE115.

To facilitate lab demonstrations and the cost figure calculation, structure your design as shown in Figure 2 below. Calculate the cost figure only for the picoMIPS module which you develop.

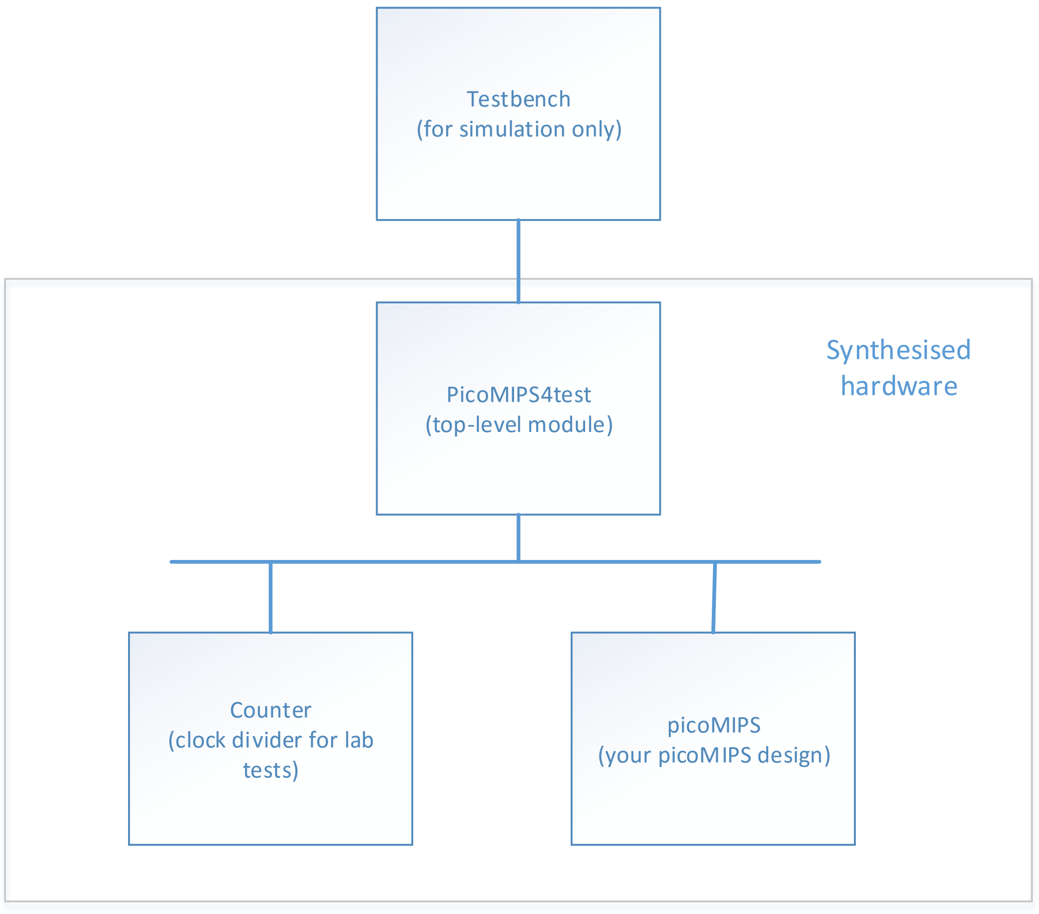

Figure 2. Synthesised design structure.

Use the following code for the top-level module picoMIPS4test.sv and the clock divider counter.sv. The purpose of the clock divider is to eliminate bouncing effects of the mechanical switches which are used to input data as outlined by the pseudocode below.

File picoMIPS4test.sv:

```
// synthesise to run on Altera DE0 for testing and demo
module picoMIPS4test(
  input logic fastclk,  // 50MHz Altera DE0 clock
  input logic [9:0] SW, // Switches SW0..SW9
  output logic [7:0] LED); // LEDs

  logic clk; // slow clock, about 10Hz

  counter c (.fastclk(fastclk),.clk(clk)); // slow clk from counter

  // to obtain the cost figure, synthesise your design without the counter
  // and the picoMIPS4test module using Cyclone IV E as target
  // and make a note of the synthesis statistics
  picoMIPS myDesign (.clk(clk), .SW(SW),.LED(LED));

endmodule
```

File counter.sv:

```
// counter for slow clock
module counter #(parameter n = 24) //clock divides by 2^n, adjust n if
necessary
  (input logic fastclk, output logic clk);

logic [n-1:0] count;

always_ff @(posedge fastclk)
    count <= count + 1;

assign clk = count[n-1]; // slow clock

endmodule
```

ELEC6234 lecture slides will describe the picoMIPS architecture in detail and provide SystemVerilog coding suggestions for the picoMIPS blocks. An additional functionality to input 8-bit data and to output 8-bit results will be required as described below. You may design your own instruction set and modify the instruction format in any way you wish. You may also modify the architecture if it helps to reduce the cost figure. However, when modifying the architecture, bear in mind that a processor is still required to implement the affine transformation algorithm outlined below, not a dedicated hardware design. A processor is characterised by the presence of two separate and discernible parts: the control path and the data processing path. The control path should contain a program memory which stores the machine program of the algorithm.

Affine transformation algorithm

An affine transformation is a geometrical transformation that preserves co-linearity, i.e. all points lying on a line will also lie on a line after the transformation and distance ratios are preserved. For two-dimensional images, the general affine transformation can be expressed as:

$$\begin{vmatrix} x_2 \\ y_2 \end{vmatrix} = A \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} + B$$

Where [x1,y1] are the coordinates of a pixel before the transformation and [x2,y2] – after the transformation. The 2x2 matrix **A** and the two-element vector **B** provide the transformation coefficients. For example, a pure translation of pixels occurs if:

$$A = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}, B = \begin{vmatrix} b_1 \\ b_2 \end{vmatrix}$$

Similarly, the following coefficients implement pure scaling:

$$A = \begin{vmatrix} a_{11} & 0 \\ 0 & a_{22} \end{vmatrix}, B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

In practice different affine transformations are combined to produce a complex transformation.

Implementation of affine transformation in picoMIPS

You are required to develop both a smallest possible picoMIPS architecture and a machine-level program for the general affine transformation. The 6 constants that define the transformation must be included as immediate literals in your program. You can choose one out of two sets of transformation constants as outlined below in Section “Data Sets”. Pixel coordinates must be read from the switches SW0-SW7 on the FPGA development system and the resulting pixel coordinates after the transformation displayed on the LEDS LED0-LED7. Switch SW8 provides handshaking functionality as described in the pseudocode below. Switch SW9 should act as an active low reset.

Pseudocode

1. Wait for coordinate *x1* by polling switch SW8. Wait while SW8=0. When SW8 becomes 1 (SW8=1) read the coordinate *x1* from SW0-SW7.
2. Wait for switch SW8 to become 0
3. Wait for coordinate *y1* as specified in step 1.
4. Wait for SW8 to become 0
5. Execute the affine transformation algorithm and display coordinate *x2* on LED0-LED7.
6. Wait until SW8 becomes 1
7. Display coordinate *y2* on LED0-LED7.
8. Wait until SW8 becomes 0
9. Go to step 1.

Input/Output

The input/output functionality can be implemented in several different ways. For example, you can design your own IN and OUT instructions for reading/writing data using external ports. To use fewer hardware resources, you can consider connecting the ALU output, or a register output directly to the LEDs. You could consider dedicating a specific register number, e.g. register 1 to the input port. In this way, an ADD instruction can be used to read data, e.g. ADD %5, %0, %1 would store the input data in register %5. Be creative and use your imagination!

Data sets

In your implementation, choose one of the two following data sets.
1.
$$A = \begin{bmatrix} 0.75 & 0.5 \\ -0.5 & 0.75 \end{bmatrix} \qquad B = \begin{bmatrix} 20 \\ -20 \end{bmatrix}$$
2.

$$A = \begin{bmatrix} 0.5 & -0.875 \\ -0.875 & 0.75 \end{bmatrix} \qquad B = \begin{bmatrix} 5 \\ 12 \end{bmatrix}$$

They both represent rotation, scaling and translation combined into a single affine transformation.

Suggested data formats

Affine transformation and fixed-point representation

For the two-dimensional affine transformation implemented in your picoMIPS:

$$\begin{vmatrix} x_2 \\ y_2 \end{vmatrix} = A \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} + B$$

use the following data formats. Pixel coordinates [x1,y1] and [x2,y2] and coefficients of vector B are 8-bit 2's complement signed integers, i.e. their values are in the range -128..+127.

The coefficients of matrix A are 2's complement signed fixed-point fractions in the range -1 .. +1 – 2^(-8), i.e. they are 2's complement fractional numbers with the radix point positioned after the most significant bits. Therefore the weights of the individual bits are:

Bit position	Weight
7	-2^0
6	2^(-1)
5	2^(-2)
4	2^(-3)
3	2^(-4)
2	2^(-5)
1	2^(-6)
0 (LSB)	2(-7)

When coefficients of the matrix A are multiplied by pixel coordinates, a double-length 16-bit product is obtained which is a 2's complement number with the radix point positioned after the 9-th bit. Note however that the result [x2,y2] must be an 8-bit 2's complement whole number.

Binary multiplication examples

Binary multiplication of 2's complement 8-bit numbers yields a 16-bit result. As one of the numbers is represented in the range $-1..+1-2^{-7}$ and the other in the range -128..127, it is important to determine correctly which 8-bits of the 16-bit result represent the integer part which should be used for further calculations. The following examples illustrate which result bits represent the integer part.

Example 1. Multiply 0.75 x 6.

In 2's complement 8-bit binary representation these two operands are:

weights:	-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
0.75 =	0.	1	1	0	0	0	0	0

weights:	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
6 =	0	0	0	0	0	1	1	0.

The 16-bit result is:

-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
0	0	0	0	0	0	1	0	0.	1	0	0	0	0	0	0

which represents the value of 4.5. The shaded area shows which bits need to be extracted when the representation is truncated to 8 bits. Note that the fraction part is discarded entirely, so the 8-bit result is now 4. Also note that when the leading bit is discarded, the weight of the new leading bit must now change from 2^7 to -2^7. Why? The importance of the correct interpretation of the leading bit's weight is evident in the following example, where the result is negative.

Example 2. Multiply -0.25 x 20.

The two operands in signed binary are:

weights:	-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
-0.25 =	1.	1	1	0	0	0	0	0

weights:	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
20 =	0	0	0	1	0	1	0	0.

The 16-bit result is:

-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
1	1	1	1	1	1	0	1	1.	0	0	0	0	0	0	0

which is -5 in decimal representation. Again, the shaded area shows which 8 bits to extract when the result is truncated from 16 to 8 bits for further calculations. The truncated 8-bit result has the weight of -2^7 on the most significant bit so that it still correctly represents -5:

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	1	0	1	1.

If you would like to experiment more with binary multiplication of signed numbers, run some SystemVerilog simulations of your multiplier in Modelsim, which is what you should do anyway to test your multiplier module before synthesis. If you would like to practice signed binary multiplication by hand, I recommend you use Booth's algorithm (and rather fewer than 8-bits!):

en.wikipedia.org/wiki/Booth's_multiplication_algorithm

The original paper where Andrew Booth published his algorithm back in 1951 can be found here: http://qjmam.oxfordjournals.org/content/4/2/236.short

Design strategy

Develop SystemVerilog code and a separate testbench for each module in your design. Simulate each module in Modelsim. Synthesise each module in Altera Quartus and carefully analyse the synthesis warnings, statistics and RTL diagrams. When you are satisfied that all your picoMIPS modules are correct, write a testbench for the whole design and simulate. Synthesise the whole design and again, carefully analyse the warnings, statistics and RTL diagrams. You will be able to take an FPGA Development System on loan for a few days in Week 6, Week 7 or Week 9. A detailed schedule of loans will be published on the ELEC6233 and ELEC6234 notes websites Test your design either at home or in the laboratory. In Week 9 or 10 (after the Easter Break) you will be asked to demonstrate your design in the Electronics Laboratory.

Formal report

Submit an electronic copy of your report through C-BASS, the electronic handin system, and a printed copy to the ECS front office by the deadline specified on the ELEC6233 and ELEC6234 notes websites. The report should not exceed 3000 words in length. It must contain a full discussion of your design, including the final circuit diagrams, your instruction set and your program. A Word template will be provided with suggested structure of the report. Source files must be packaged in a zip file and submitted electronically as a separate file at the same time. In this exercise, 20% of the marks are allocated to the report, its style and organisation, with the remaining 80% for the technical content. As always, bonus marks are awarded for implementation of novel concepts.

tjk, rev. 7 Feb'17