**ELEC6234 – Embedded Processor Synthesis**

Peter Alexander

pa4g17

Electronics with Computer Systems

Dr Sasan Mahmoodi

**ABSTRACT:** *Summarise your work in less than 100 words stating briefly what was achieved.*

1. **1. Introduction**

*State the objectives of the assignment. Summarise briefly your preparation work, your experimental work,, and results achieved. Specifically, state which parts of the assignment were delivered according to the requirements and summarise any extensions to the basic specification you have carried out with references to the sections. ( approx. 0.5 page).*

1. **2. Overall architecture of the design**

Figure 2 shows the system block diagram for the final design, Figure 1 shows the application software. The main focus for hardware reduction was simplifying the ALU. This allowed a greatly simplified instruction set and a shorter program. The structure of the ALU was specialised for the Affine transformation, limiting the number of control signals needed to operate it. The control of the ALU was inspired by NISC design principles. There is no dedicated ALU controller, giving the designer greater freedom when defining the instruction set. Figure 3 shows the output of the picoMIPS4\_test testbench, demonstrating a successful Affine transformation.

1. **2. 1. I/O**

By limiting how I/O is used in the software, simplifications can be made to switch reading hardware over the general purpose picoMIPS implementation. SW[7:0] are never used directly in a calculation, they are used to load x1 and y1 into registers at the beginning of the program. SW[7:0] are multiplexed onto the w\_data line when the in\_en flag goes high. This arrangement allows the ACC and registers to be loaded with the same instruction (see Tables 1 and 2 for details on the ACCI instruction). SW[8] is only used in branching instructions. It is only ever read into the A input on the adder so it is only included in that input selector (more details on this in Section 1.3.2).

The LEDs are wired directly to the accumulator output so the results of each accumulation operation can be observed.

1. **2. 2. Instruction Set**

Table 1 shows the instructions implemented and their operations. To limit the number of instructions, and therefore decoder logic and opcode bits, the ACCI (Accumulate-Intrinsic) and MACI (Multipliy-Accumulate-Intrinsic) instructions perform two independent operations. Table 2 provides examples of how the operations required by the Affine transformation can be performed using this instruction set.

Figure 1: The application software

8800 //1000100000000000 BEQ %1 %0 0 # Wait while SW[8] == 0

1000 //0001000000000000 ACCI %2 %0 0 # Read SW[7:0] into %2

C800 //1100100000000000 BNE %1 %0 0 # Wait while SW[8] != 0

8800 //1000100000000000 BEQ %1 %0 0 # Wait while SW[8] == 0

1900 //0001100100000000 ACCI %3 %1 0 # Read SW[7:0] into %3

C800 //1100100000000000 BNE %1 %0 0 # Wait while SW[8] != 0

010C //0000000100001100 ACCI %0 %1 12 # Load b2 into ACC

42C0 //0100001011000000 MACI %0 %2 b11000000 # Add a21\*x1 to ACC

6360 //0110001101100000 MACI %4 %3 b01100000 # Add a22\*y1 to ACC to give y2. Save to %4

0105 //0000000100000101 ACCI %0 %1 5 # Set ACC to 5

4260 //0100001001100000 MACI %0 %2 b01100000 # Add a11\*x1 to ACC

4340 //0100001101000000 MACI %0 %3 b01000000 # Add a12\*y1 to ACC to give x2.

8800 //1000100000000000 BEQ %1 %0 0 # Wait for SW[8] to become 1

2100 //0010000100000000 ACCI %4 %1 0 # Write y2 to ACC for display

C800 //1100100000000000 BNE %1 %0 0 # Wait for SW[8] to go to 0

80F1 //1000000011110001 BEQ %0 %0 -15 # Unconditional jump to program beginning

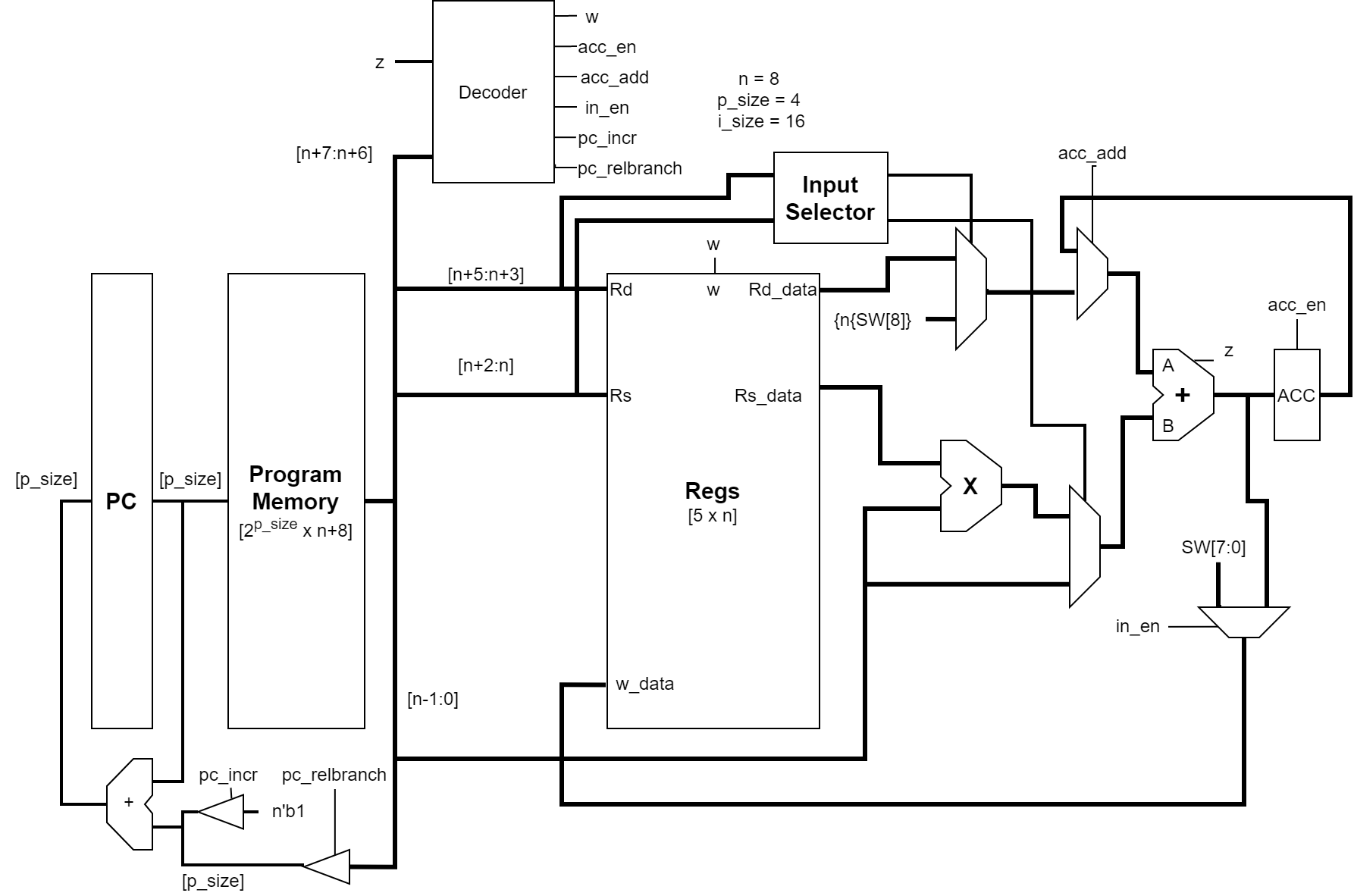


Figure 2: The system block diagram for the final design

x1 = 25, x2 = 78

a11 = 0.75, a12 = 0.5

a21 = -0.5, a22 = 0.75

b1 = 5, b2 = 12

x2 = x1\*a11 + y1\*a12 + b1 = 62  
y2 = x1\*a21 + y1\*a22 + b2 = 58  
 (Actual output is 57 due to representation errors).

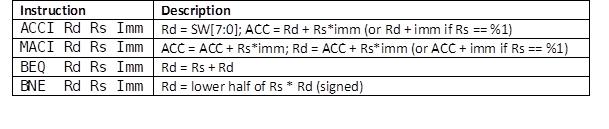
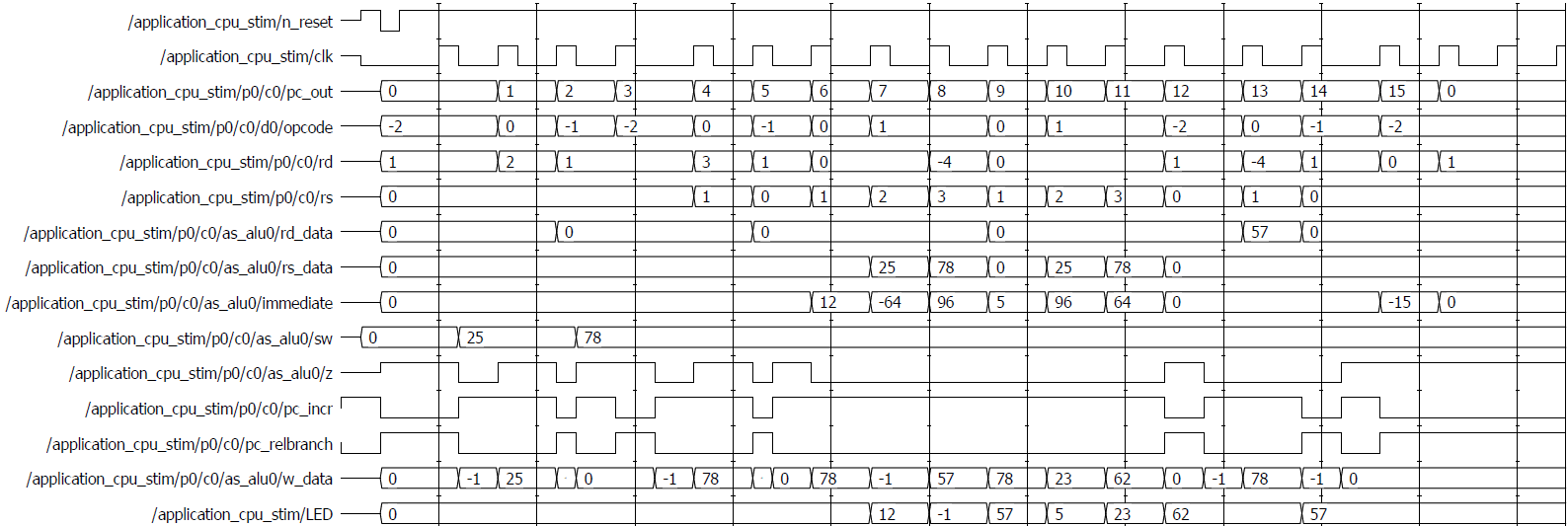


Table 1: A table showing the implemented instruction set.

Table : A table showing how operation of the Affine transformation are performed.

Figure : The output of the picoMIPS4\_test testbench.



%2 = 25

%3 = 78

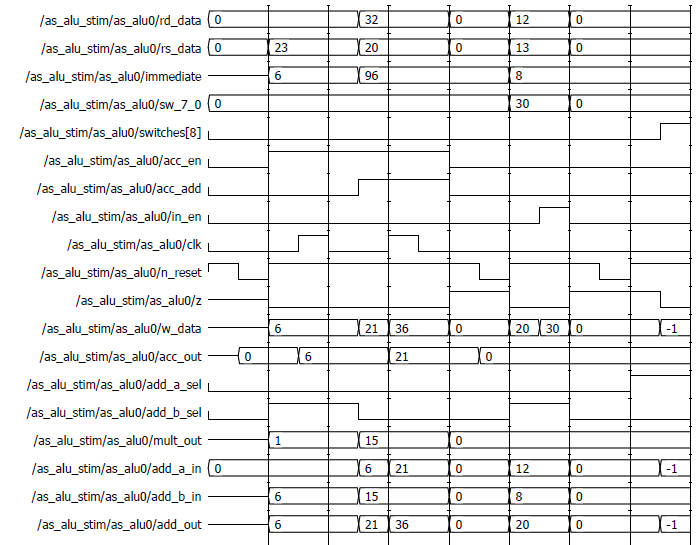
%5 = ACC = 57



ACC = 62 = x2

ACC = %5 = 57 = y2

1. **3. Details of hardware blocks (use appropriate subsection titles for your hardware modules)**
2. **3. 1. ALU**



Test load immediate into ACC.

ACC goes to immediate (6) after clock.

Test MACC operation.

ACC goes to rs\*immediate + ACC

ACC =20\*0.75

+6 = 21

Test ADDI and switch[7:0] writeback.

in\_en = 0:

W\_data = rs + imm

= 20

in\_en = 0:

W\_data = sw[7:0]

= 30

Test SW[8] value detection using z (zero) flag.

Zeros on all inputs and a\_in\_sel = 1 sets

add\_out = {n{sw[8]}.

z flag detects change in sw[8] value.



Figure 4: A plot of the as\_alu module testbench

The only module in this system which differs significantly from the standard picoMIPS implementation is the ALU. The multiplier, adder, ACC and the four multiplexers linking them together are implemented in a contained module called as\_alu in a file of the same name. It was designed around the multiply-accumulate operation. Discrete blocks of hardware with 2 input multiplexers connecting them minimise the control logic needed to implement fundamental operations. The input selector block is sensitive to Rd/Rs = %1. For the adder A input, this is used to read in SW[8]. For the adder B input, this is used to bypass the multiplier to read the immediate straight into the adder (as if you were multiplying by 1). %0 is tied to 0 within the register file. Figure 4 shows the output of the as\_alu testbench, demonstrating the fundamental operations used in the application software.

1. **3. 2. Registers, Program Memory, Program Counter**

As mentioned in Section 1.1, no changes were made to the Registers, Program Memory and Program Counter modules developed for Assignment 1. Information regarding the verification of these modules will not be included here as Figure 3 demonstrates their correct functioning. As the interfaces and function for these blocks were not changed, they did not have to be reverified and the structure for the CPU could remain relatively unchanged.

As only 5 registers were used (including the overwritten values for %0 and %1) Rd and Rs only take up 3 bits each in the instruction. Only 4 instructions were implemented, so the opcode is only 2 bits. This reduced the instruction size (i\_size) to 16 bits. The program is only 16 instructions long, reducing the program size (p\_size) to 4. This indirectly resulted in large hardware savings on synthesis over a higher p\_size and i\_size. It is thought that these values enable the use of 4-input logic devices in the program memory ROM, simplifying the addressing logic.

1. **5. FPGA implementation**

*Explain how you tested your design after programming the FPGA. In case you had to edit your original code and resynthesize – explain what you did. ( approx. 1-2 pages)*

1. **6. Conclusion**

*State which objectives listed in your Introducton have been achieved. Calculate the cost figure of your design for synthesis on a Cyclone V. Give your general conclusion, comment on what you learnt. Comment on ways to improve the design or extend it further. ( approx.0.25 – 0.5 of a page)*

1. **7. References**

*Quote the sources of your information. Especially make reference to any sources you used in the development of your code.*