ELEC6234 – Embedded Processor Synthesis

Peter Alexander
pa4g17
Electronics with Computer Systems
Dr Sasan Mahmoodi

ABSTRACT: *A synthesisable application-specific n-bit processor was developed in SystemVerilog to implement a given algorithm whilst minimising the hardware cost of the design. It was verified and synthesised on an FPGA.*

1. 1. Introduction

The goal of this assignment as given in [1] was to produce synthesisable n-bit application-specific processor in SystemVerilog for computing an Affine transform on pixel data presented on an input port. An 8-bit version of this processor was synthesised on a DE-1 FPGA development board. The hardware cost of the system was minimised.

The processor has been implemented successfully. The algorithm runs successfully both in simulation and on the DE-1 board.
The general-purpose picoMIPS processor submitted for Assignment 1 was used as the basis for this processor. The ALU was redesigned to minimise control hardware, program size, and the size of the instruction set. The registers, program counter, and program memory were shrunk but otherwise unchanged. The Python assembler written for Assignment 1 was edited to support the new instruction format.

1. 2. Overall architecture of the design

Figure 2 shows the system block diagram for the final design, Figure 1 shows the application software, and Figure 3 shows the top-level testbench output.

1. 2. 1. I/O

By limiting how I/O is used in the software, simplifications were 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 used only 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 both perform two independent operations. Table 2 provides examples of how the operations required by the Affine transformation can be performed using this instruction set.
All instructions are in the format OP %rd %rs imm with 2 bit opcode, 3 bit register values, and 8 bit immediate giving an instruction length of 16-bits.

```
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 1: The application software

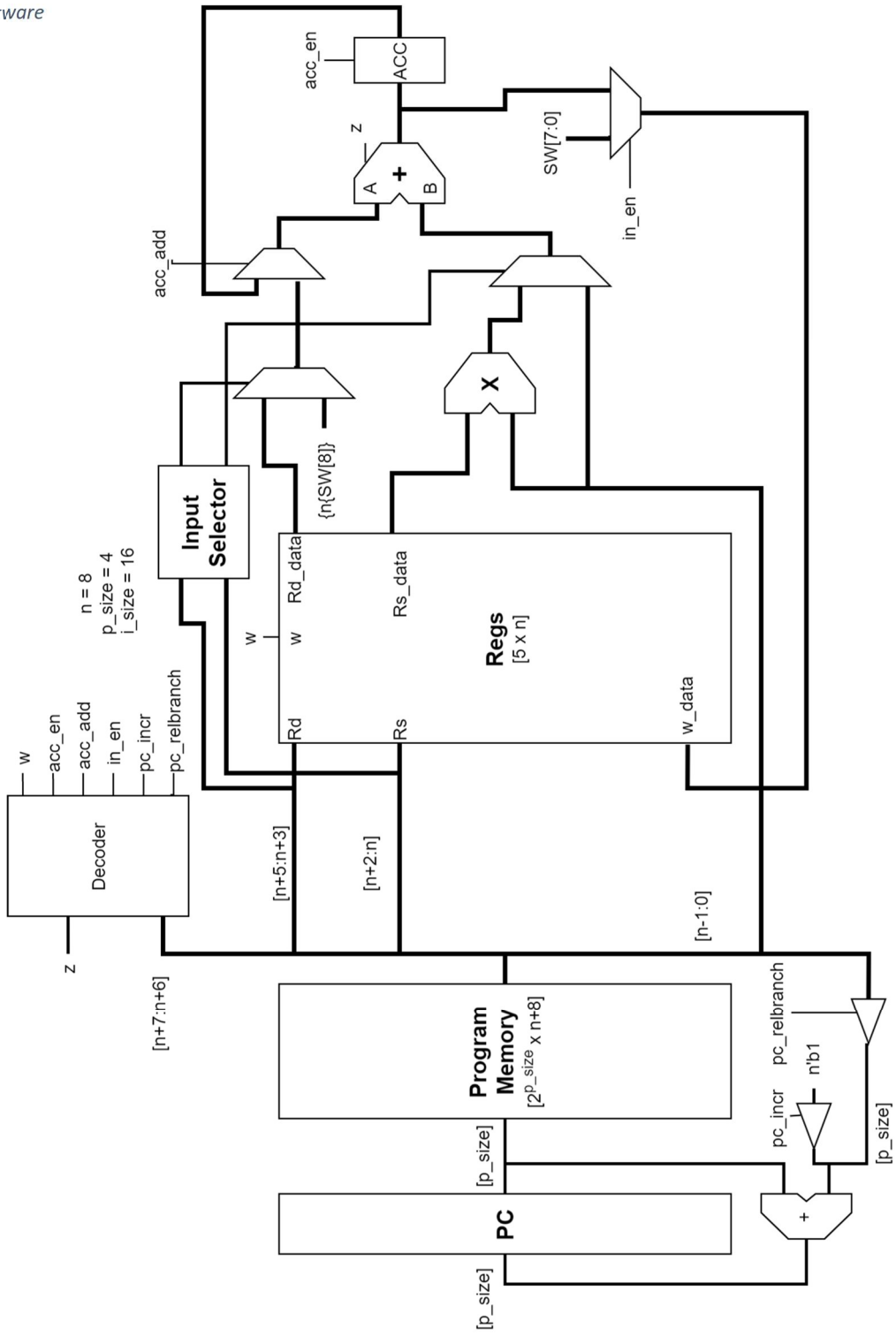

Figure 2: The system block diagram for the final design

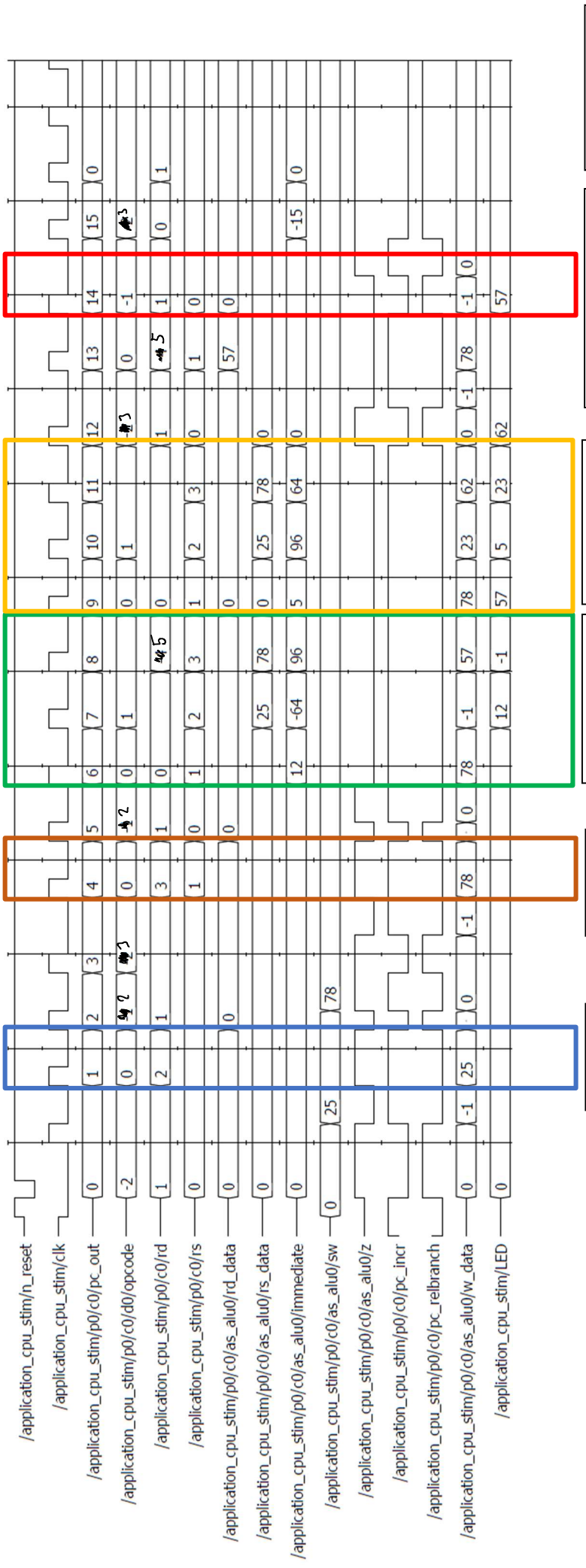

Figure 3: The output of the picoMIPS4_test testbench. x1 is loaded on instruction 1, y1 is loaded on instruction 4. y2 is computed in instructions 6,7,8, and x1 is computed in instructions 9,10,11. This leaves x2 in ACC for display. y2 is displayed on instruction 14.

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 decimal representation errors).

Instruction	Description
ACCI Rd Rs Imm	Rd = SW[7:0]; ACC = Rd + Rs*imm (or Rd + imm if Rs == %1)
MACI Rd Rs Imm	ACC = ACC + Rs*imm; Rd = ACC + Rs*imm (or ACC + imm if Rs == %1)
BEQ Rd Rs Imm	Rd = Rs + Rd
BNE Rd Rs Imm	Rd = lower half of Rs * Rd (signed)

Table 1: A table showing the implemented instruction set.

Operation	Instruction
Load SW[7:0] into register %2 (Loading x1 and x2)	`ACCI %2 %0 0`
Load an immediate operand into ACC (Loading b1 and b2)	`ACCI %0 %1 5`
Multiply accumulate without write back (ACC += a11*x1)	`MACI %0 %2 b0110…`
Multiply accumulate with write back (%4 = a12*y1 + ACC)	`MACI %4 %3 b1000…`
Wait until SW[8] goes to 0/1	`BNE/BEQ %1 %0 0`

Table 2: A table showing how operation of the Affine transformation are performed.

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

1. 3. 1. ALU

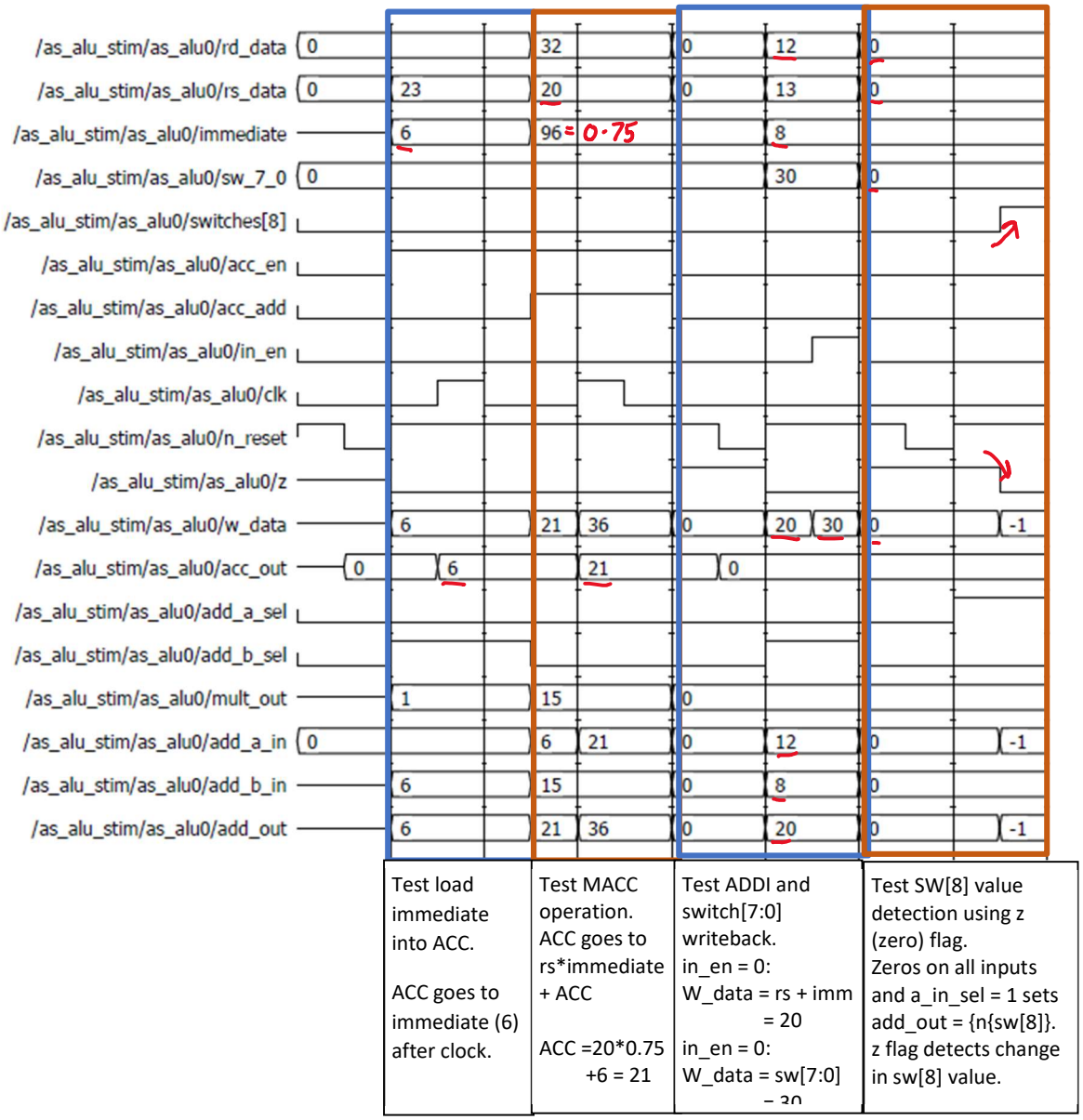

Figure 4: A plot of the as_alu module testbench

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 connected by 2-input MUX's minimises 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. The z flag is generated with a comparator on the output of the adder.

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 shows the system as a whole is functioning correctly. 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 larger values 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

1 . 4. 2. Integrated Design Synthesis

As the as_alu was the only new module in this design, it was decided that the sub-module synthesis step should be skipped. Only one synthesis warning was given for the new hardware: an incorrectly defined port size. This was fixed and the as_alu module rtl diagram was checked for correctness.

1 . 4. 3. Synthesised System Testing

To determine that that FPGA implementation was working correctly the tests performed in the ./rtl/picoMIPS4test_stim.sv testbench were repeated along with others generated with a Python script. The correct outputs were observed, with small variation of ± 1 due to representation errors, and the behaviour matched the simulated system.

1. 6. Conclusion

Synthesising the final programmed CPU on the DE-1 board required the following resources:

ALMs	46
Registers	36
DSP Blocks	1

This gives a cost value of 46.
That is 39 fewer ALMs and 18 fewer registers than the generic picoMIPS processor submitted for Assignment 1. The processor was successfully able to complete the algorithm given in [1] both in simulation and when synthesised on a DE-1 board.

This project provided experience in modifying existing, verified processor cores to perform application-specific tasks, and in producing a minimal instruction set and datapath to complete a specific algorithm.

One possible extension to this project would be to remove registers %0 and %1 from the register file as they are never read from. This may reduce the register count in the event that the synthesis tools don't prune them automatically.

1. 7. References

[1] T. Kazmierski, "ELEC6234 Assigmnent 1 Brief," 2021.