Hardware Verification Assignment 1 – Processor Verification with ACL2

j.schmaltz@tue.nl
www.win.tue.nl/∼jschmalt/

This first assignment will take you through the use of ACL2 and its special libraries for the verification of hardware designs written in the Verilog Hardware Description Language. In this assignment, you will design and verify a simple processor. You need to hand in your solution (a .lisp file with a lot of comments is acceptable) before Friday September 18th 2015 09:00.

Note. To get documentation about anything in ACL2, just type :doc <whatever> at the ACL2 prompt. The main on-line documentation for ACL2 is the combined ACL2 and books manual available at :

http://www.cs.utexas.edu/users/moore/acl2/v7-1/combined-manual/?topic=ACL2____TOP

1 Tools installation

To start, you need to install several tools on your machine.

1.1 ACL2

To install ACL2 visit the page below. You will need to install a common LISP. Among the options proposed on the ACL2 website. You should use CCL.

http://www.cs.utexas.edu/users/moore/acl2/v7-1/HTML/installation/installation.html

It is important that you certify all books, in particular, all books under "books/centaur". Certifying all books will take several hours. So, run this over night!

1.2 Glucose SAT solver

To verify hardware designs, external SAT solvers can be used within ACL2. In the course, we use "Glucose". You will find the source code at the following page:

http://www.labri.fr/perso/lsimon/glucose/.

Download the source code of the latest version and compile it.

1.3 GTKWave

GTKWave is a wave form viewer. The Verilog verification flow in ACL2 supports the generation of wave form descriptions that can be read and viewed by GTKWave. The best way to install GTKWave is probably using your package management system (e.g., apt-get for Linux, port for Mac OS X). More information about GTKWave can be found at http://gtkwave.sourceforge.net.

2 ESIM Tutorial

Before designing and verifying your ALU, you need to learn about how ACL2 can be used to analyse hardware. To do so, open the file located at "/books/centaur/esim/tutorial/alu16.lsp" in, for instance, an emacs buffer. You will see examples about using different techniques like SAT and BDD. You will also learn how to create symbolic test vectors and use them for testing or proving theorems using GL.

3 Designing a simple ALU

3.1 Ripple Carry Adder

The main component of our ALU will be a simple Ripple Carry Adder. The main blocks used to build this adder are half-adder and full-adder modules. The half-adder is shown in Figure 1 and defined by the following equations:

$$\begin{aligned} HA_s(a,b) &= a \oplus b \\ HA_c(a,b) &= a \wedge b \end{aligned}$$

where $\oplus$ denote the exclusive or between Booleans.

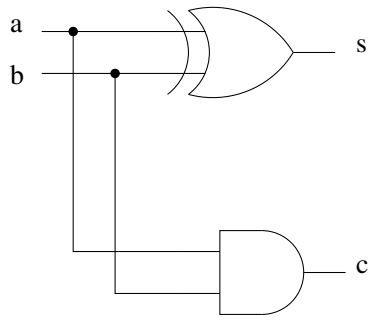

Figure 1: Circuit for an half-adder.

The full-adder is shown in Figure 2 and defined by the following equations:

$$\begin{aligned} FA_s(a,b) \equiv & \quad HA_s(c, HA_s(a,b)) = c \oplus HA_s(a,b) = c \oplus a \oplus b \\ FA_c(a,b) \equiv & \quad HA_c(c, (HA_s(a,b)) \vee HA_c(a,b)) = c \wedge HA_s(a,b) \vee a \wedge b \\ = & \quad c \wedge (a \oplus b) \vee a \wedge b \end{aligned}$$

Exercise 1. Write a description of the half-adder and of the full-adder in Verilog. Start with a module for the half-adder and then instantiate two copies in the full adder.

Addition between two bit vectors of n bits is computed by cascading several full adders. Figure 3 shows an adder for 4-bit unsigned numbers. Because the carry is passed to the next computation, such a circuit is called a *ripple carry adder* or *carry chain adder*.

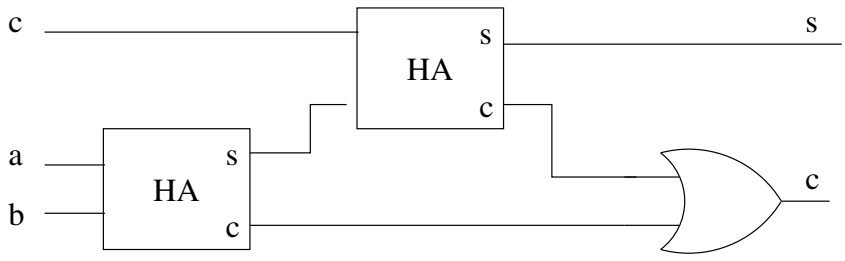

Figure 2: Circuit for a full-adder.

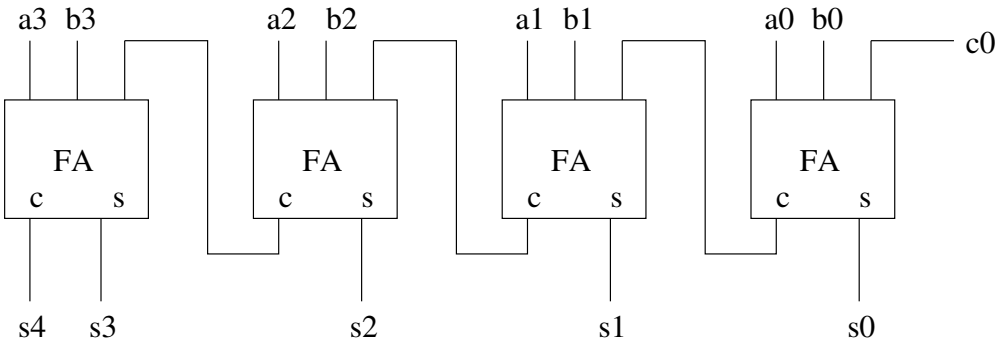

Figure 3: Circuit for a 4-bit ripple carry adder.

Exercise 2. Write the description of an 8-bit ripple carry adder in Verilog. Let call this module `rca8`. Its inputs are two 8-bit numbers and an input carry. Its outputs are a carry bit and an 8-bit result.

You now have the description of the basic circuit of the ALU.

3.2 Arithmetic Logic Unit

An Arithmetic Logic Unit (ALU) is built from an adder and logic gates (AND, OR, etc.) to compute the result of logic and arithmetic functions. Figure 4 shows an example of a 4-bit ALU. The control bits of the ALU are $ALU.op = kijc_{in}$. This 4-bit vector encodes the operations that the ALU can perform. These operations are summarised in Table 1.

Exercise 3. Using your adder of the previous exercise, write the description of an 8-bit extension of the 4-bit ALU shown in Figure 4. The ALU should compute the following flags: C (carry), Z (zero), V (2's complement overflow), and N (negative). The main idea is to replace the cascade of full-adders in Figure 4 with your ripple carry adder. The equations to compute the different flags are as follows:

- $C \equiv s_8$ (that is, the carry out of the ripple carry adder)
- $Z \equiv (s == 0)$
- $V \equiv a_7 \cdot b_7 \cdot \overline{s_7} + \overline{a_7} \cdot \overline{b_7} \cdot s_7$
- $N \equiv s_7$

ALU.pp	operation
1 0 0 0	a
1 0 0 1	$a+1$
1 0 1 0	$a-1$
1 1 0 0	$a+b$
1 1 0 1	$a+b+1$
1 1 1 0	$a+\overline{b}$
1 1 1 1	$a-b$
0 0 0 0	$a \wedge b$
0 0 1 0	$a \oplus b$
0 1 0 0	$a \vee b$

Table 1: Encoding of ALU operations.

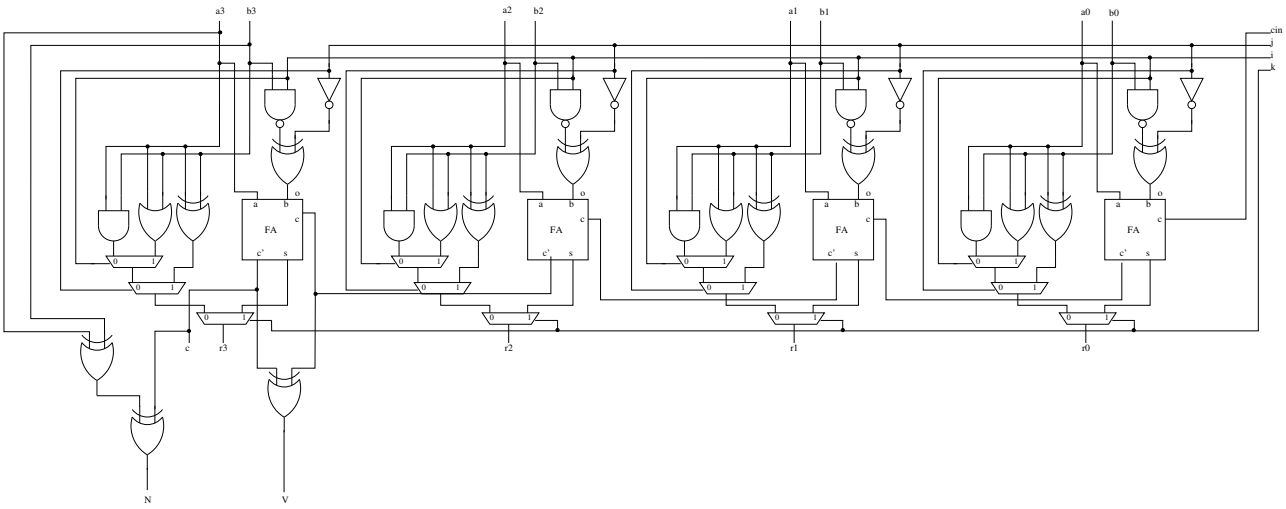

Figure 4: 4-bit Arithmetic Logic Unit

Note that you might need to describe some extra modules, e.g., a 2 input multiplexer `mux2`.

4 Proving correctness of the ALU (60 minutes)

4.1 Translation and Symbolic Test Vector

During the ESIM tutorial, you should have learned how to "translate" Verilog modules into ACL2 terms. You should also have learned how to define a symbolic test vector and run it using `stv-run` and `stv-debug`.

Exercise 4. Translate your ALU design to ACL2 and write a symbolic test vector for it.

Now that you have an STV for your design, you could run it. But, you won't do that and you will start right away with writing a simple theorem stating, for instance, correctness of the "a" operation of the ALU. The idea is to write something like the following:

```
(def-gl-thm myop-is-correct
  ;; correctness of myop
  :hyp
  (and myhyps)
  :concl
  (b* ((outs (stv-run (my-stv)
                      (my-stv-autoins)))
       ((assocs myin1) outs)
       ((assocs myin2) outs)
       )
      (and result is correct
           flags are correct))
  :g-bindings
  (gl::auto-bindings some bindings))
```

Exercise 5. Prove such a def-gl theorem for one of the operations of your ALU.

4.2 Macro's

Before proving more operations correct, it is – as you have seen it in the ESIM tutorial – helpful to write some macro's to make statements from concise.

Exercise 6. Write a macro such that the theorem of the previous exercise is proven by simply writing:

```
(alu8-thm myop-is-correct
          :opcode *myop*
          :spec (myspec))
```

4.3 Proofs

You have all the necessary machinery to prove correctness of all the operations mentioned in Table 1.

Exercise 7. Prove correctness of the ALU operations in Table 1.

5 Extension to a simple execution unit

You have all the basic block to build a complete processor! What you're missing is an execution unit that will fetch, decode, and execute instructions. You also need to add general purpose registers, instruction register, program counter, status register, and some memory.

Before going further, Figure 5 shows a first extension of an ALU. This is a block diagram of the ALU together with:

- `SREG`: Status Register. This is a 4-bit register storing the value of the four flags of the ALU.
- `IR`: Instruction Register. This is a 16-bit register storing the current instruction. It has a control bit (`IRie`). This bit is such that IR is written only when this bit is high.

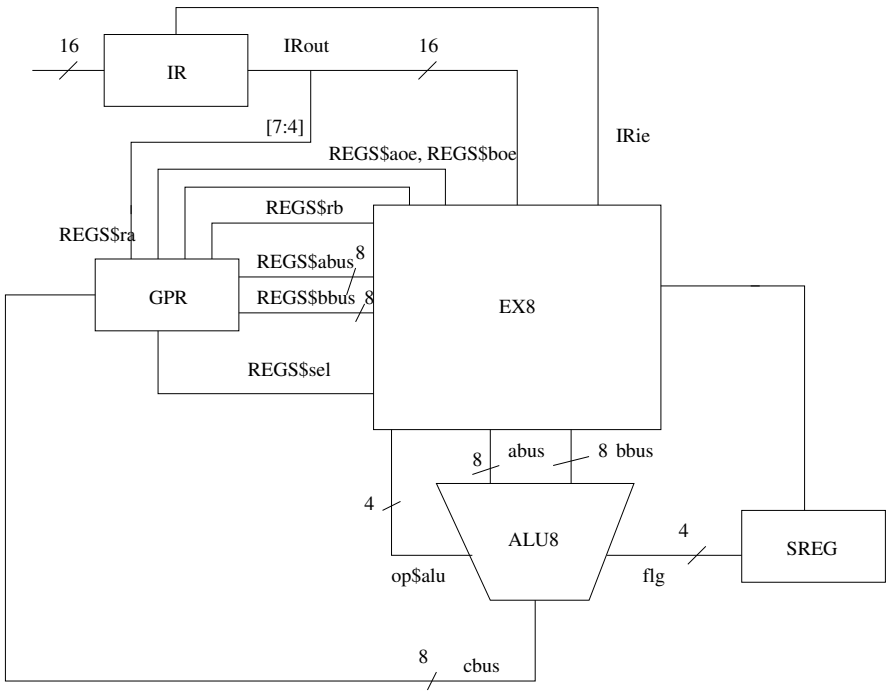

Figure 5: First extension of the ALU.

- `GPR`: General Purpose Register. This is a block containing the general purpose registers of the processor. We will start with a GPR module with only 4 registers.
- `EX8`: Execution Unit. This is the state-machine controlling the execution of the instructions.

The basic idea is that the execution unit reads the registers, the instruction, and the flag. Depending on those values, the unit will orchestrate the different bits so that the ALU outputs the correct result. Before going into more details, we first need to determine which instructions we will have and how they will be encoded.

5.1 Instruction Set Architecture

We will encode instructions over 16 bits. We fill start with two formats. Instruction working on two registers and instruction working on one register and an 8-bit value. The format of a 2 register instruction is:

$$op\ op\ op\ op\ op\ op\ op\ op\ dddd\ rrrr$$

The first 8 bits will encode the instruction. The remaining 8 bits will encode the two registers used by the instruction, namely, R_d and R_r. The format of a 1 register instruction is:

$$op\ op\ op\ op\ kkkk\ dddd\ kkkk$$

The first 4 bits will encode the instruction. Bits *dddd* encode the register used by the instruction. The remaining bits encode the 8-bit value. This way, R_d is always encoded by the same bits in all instructions.

The Instruction Set Architecture consists in specifying all the operations of a processor. As an example, we will consider the AVR8 ISA. The link below will point you to the document. You should *not* read the document. We will only look at some of the instructions to get some inspiration.

http://www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf

The AVR8 is also an 8-bit architecture. We will use the opcode of some of the AVR instructions as a specification for our processor. This means that the AVR ISA will specify for each opcode what are the expected outputs.

Note. The main deviation between the AVR and our simple processor will be the number of cycles per instructions. Our processor will not execute as fast as the AVR. We also only have 4 flags. The AVR has 8. The flags we have are C,Z,V, and N.

Already with our simply ALU, we can implement quite a number of instructions. We will start by only considering the following instructions:

ADC: Add with carry

See page 16 out of 160 of the AVR ISA. This instruction executes an addition with the carry flag. It updates all four flags. We do not have a program counter yet. But if we would have one, it would be incremented by 1.

ADD: Add without carry

See page 17 out of 160. This instruction simply adds two registers.

AND: Logical AND

See page 19 out of 160 of the AVR ISA. This instruction computes the logical conjunction between two registers. Note that it clears the 2's complement overflow flag and leave the carry bit unchanged.

ANDI: Logical AND with immediate

See page 20 out of 160 of the AVR ISA. This instruction computes the logical conjunction between a register and an 8-bit value. Here also, the 2's complement flag is cleared.

LDI: Load Immediate

See page 94 out of 160 of the AVR ISA. This instruction loads an 8-bit value into a register. Note that it does not modify the flags.

5.2 Adding decode and execute stages

Figure 5 adds enough modules to be able to decode and execute the instruction stored in the IR register. It still needs memory modules to also implement instruction fetch. To keep things simple, we will consider the instruction an input.

On the course webpage, you will find a template implementation of Figure 5 in a Verilog module named `core8`. Note that this file does not contain the alu8 module. You should use your own ALU. The remaining modules of `core8` are fully implemented, except for the execution unit (ex8). The specification of each module is as follows:

- `IR`: Inputs are the top level input `instr` of the core8 module and an input enable signal (`IRie`). Its output is the 16-bit value of the instruction. This module is an instantiation of module `spr16`.
- `GPR`: Signal `REGS$ra` specifies the register (from 0 to 3) driving `REGS$abus`. Same thing for `REGS$rb` and `REGS$bbus`. When `REGS$aoe` is high (resp. `REGS$boe`), the output of `REGS$abus` (resp. `REGS$bbus`) is not driven (high impedance, value 'Z'). Finally, the register pointed by `REGS$ra` is updated with the value on `cbus` only when `REGS$sel` is high. `GPR` is an instantiation of module `gpr8`.
- `SREG` is a 4-bit register storing the value of the flags. It is a register equipped with a clear vector to allow clearing any flag. SREG is an instantiation of module `sreg4`.
- `ALU8`: the 8-bit ALU you designed and proved correct.
- `EX8`: the execution unit. Currently, the unit goes through 2 states:
  - fetch (coded as `2'b00`)
  - decode & execute (coded as `2'b01)`

  And that's it! All control signals are assigned a dummy value.

You should perform the two exercises below in lock-step, that is, add one instruction, prove it correct, then add another instruction, prove it correct, etc.

Exercise 8. Complete the design of module `ex8` to decode and execute the 5 instructions mentioned in the previous section. In the template, search for `FIXME`.

Exercise 9. Prove that your execution units sets all the control signals correctly. Note that you should prove module `ex8` correct, not its instantiation in module `core8`. The main difficulty is in writing the specification, that is, an ACL2 function specifying the output value for all output signals.

HINTS. You might need a few hints.

hint01 In ACL2, to specify the merge of bit vectors you can use function `logapp`. Using `:doc` you should be able to get information about this function. Basically, `(logapp size i j)` creates a new integer by moving integer j size bits left of integer i. Integer i is truncated before the operation. So, to concatenate one bit to a bit vector you can write something like `(logapp 1 i j)`. The size argument to this function is the size of the first integer, namely, i.

hint02 To prove theorems about module `ex8` you might need to add assumptions specifying that some signals are stable for a given period. For instance, the instruction register input to this module will not let you prove anything if it gets an arbitrary value every cycle!

hint03 Do not forget to specify some initial values. To do so, you need to add the variables you want to initialise in the `:overrides` part of a `defstv` event.