

CS623: CAD FOR VLSI DESIGN

Assignment 1

January 25, 2002

Weightage: 10%

Submission Date: 15 Feb, 2002

1 Introduction

The assignment is to develop a Verilog model for a 4-bit processor from scratch, the structure of which is shown in Figure 1. This processor supports an *instruction-set* of size seven and supports *one addressing* mode. It is a *load-store* type of architecture, with no explicit registers for use by assembly language programmers.

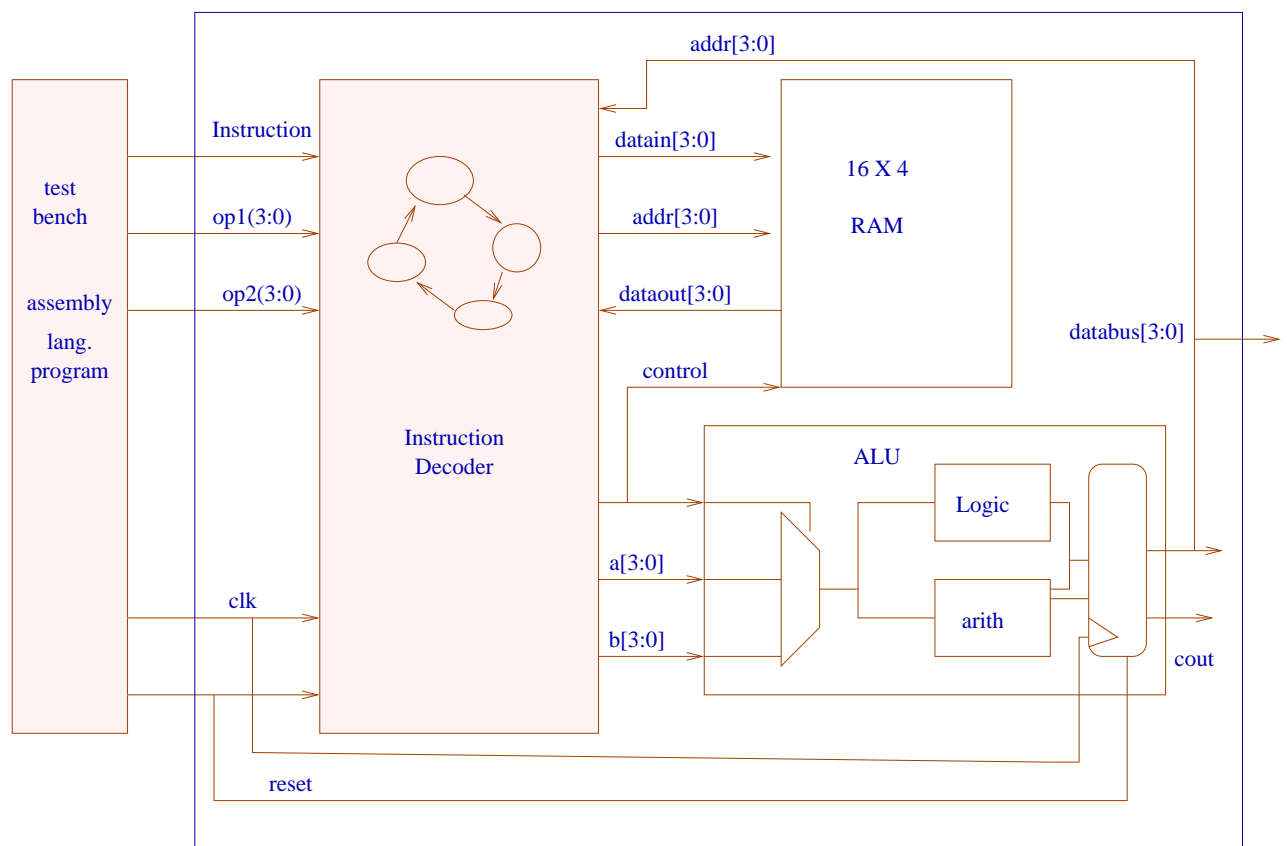
The goal is that, your model should take a *assembly-language* program written using the instruction set as a *test-bench* and execute it. Importantly, your design should give *warnings* for illegal operations, like addressing an non-existing memory etc. The *error-set* for this purpose is left to your imagination.

2 Specifications

The assignment involves nine steps of varying complexity. We list them below.

1. **The XOR gate:** Write a behavioural description of an XOR gate and test it using proper test benches. You should **not** use the Verilog `xor` construct. Inputs are *a, b* (each one bit). Output is *c* (one bit).
2. **The 4-bit full adder:** Write a RTL or structural description of a 4-bit full adder, either using top-down or bottom-up approach and verify using a *good* testbench. You should use the XOR gate of step 1. The adder takes two 4-bit numbers and `cin` as inputs and gives one 4-bit number and `cout` as outputs.
3. **The 4-bit ALU:** This has three selection bits: `S2, S1` and `S0` and hence support 8 operations. It uses the adder of step 2. The functionality is described in the table below. The ALU takes the three selection bits, two

Figure 1



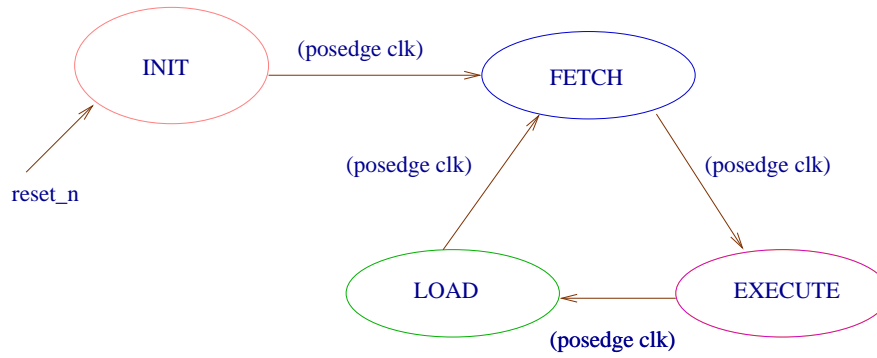


Figure 2: The States of the Instruction Decoder

4-bit numbers and `cin` for the adder as inputs and outputs a 4-bit number `f` and `cout`.

S2	S1	S0	cin	f	Description
0	0	0	0	$f = a$	Transfer a ($b = 0000$)
0	0	0	1	$f = a + 1$	Increment a
0	0	1	0	$f = a + b$	Add b to a
0	0	1	1	$f = a + b + 1$	Adds with carry
0	1	0	0	$f = a + (\text{not } b)$	Subtract with borrow
0	1	0	1	$f = a + (\text{not } b) + 1$	Subtract
0	1	1	0	$f = a - 1$	Decrement a
0	1	1	1	$f = a$	Transfer a ($b = 1111$)
1	0	0	X	$f = a \text{ or } b$	OR
1	0	1	X	$f = a \text{ xor } b$	XOR
1	1	0	X	$f = a \text{ and } b$	AND
1	1	1	X	$f = \text{not } a$	NOT

All arithmetic and transfer operations should use the adder of Step 2 and nothing more. The XOR operation should use the `xor` gate of step 1.

4. **The Registered ALU:** This introduces clocked procedures for register modeling. Register the output signals `f` and `cout` from the ALU of Step 3. The output should be synchronized to a rising edge of the clock. An active *low* asynchronous reset signal controls the initialization of these output signals. The recommended coding style is to add one more procedure to the original ALU module. This procedure will contain the asynchronous reset and the clock statements.
5. **The Instruction Decoder:** The module is a state-machine which takes as input a instruction comprising of a 3-bit **opcode** and two 4-bit **operands**.

The instruction set is described below. With the asynchronous (`reset_n` = 0) reset the decoder unit goes to the `init` state. The state-machine is shown in Figure 2. With the positive edge of every subsequent clock the unit cycles between *Fetch* - fetches data from the location specified by the first operand; *Execute* - execute the instruction using the ALU after setting all control lines, depending on the instruction; and, *Store* - store the data into the location specified by the first operand.

opcode	op1 (destination)	op2 (source)
<code>sto</code>	<code>mem_addr</code>	constant
<code>add</code>	<code>mem_addr</code>	constant
<code>sub</code>	<code>mem_addr</code>	constant
<code>and</code>	<code>mem_addr</code>	constant
<code>or</code>	<code>mem_addr</code>	constant
<code>xor</code>	<code>mem_addr</code>	constant
<code>not</code>	<code>mem_addr</code>	constant

6. **16 X 4 RAM Model.** The input ports are:

- `addr` (4-bit vector)
- `datain` (4-bit vector)
- `csn` (1 bit): Chips select, it is 0 when memory in use.
- `rwn` (1 bit): 1 = read memory; 0 = write memory

The output port:

- `dataout` (4-bit vector): Output data from RAM when in read mode

A subtle point: Whenever `dataout` is not used, when memory is in write mode or chip-select is high, the `dataout` is set to ZZZZ. Meaning, nothing is driven into the output bus.

The testbench for the RAM model is a typical *pass-fail* type test walking through all memory locations. This is a very common technique to test memory structures both for simulating and production testing.

7. **The 4-bit Processor:** Create a `dsp_processor` module and instantiate the *RAM*, *ALU* and *Instruction Decoder* modules and hook everything correctly as shown in Figure 1.

8. **Assembly Language programs:** Create a file containing assemble language program for the processor. It will be something like

```
sto 1 8
add 1 4
xor 1 3
```

Develop a small *compiler* in C/C++/Perl which will translate this program into a test-bench program, in which these instructions are input to the Instruction decoder of Figure 1, with *necessary* timings between instructions.

9. **Synchronous RAM with Bidirectional Data:** A *clock* and *reset* signal will be added into the RAM model we designed in Step 6. The code has to be modified to sync up the data input and output with the clock. The *datain* and *dataout* signals will have to be combined into an *inout* signal. This bidirectional data bus will now be controlled by four tristate drivers written in RTL style.