COMP22111: Processor Architecture

Exercise 2: Stump ALU

The first module you will design will be the ALU, which is purely combinatorial logic. The ALU has to provide functions to ADD and SUBtract (with and without a carry input), plus bitwise AND and OR operations. You should also consider what the role of the AU is when performing LDST and BCC instructions. In the Stump design the ALU always generates status flags, which may be captured and used to make decisions later – this is determined externally by the control logic. A condition code register is provided where the status flags will be latched if the instruction tells the processor to do so (the instruction is appended with an 'S'). The processing functions are fairly straightforward, but you should allow time for debugging the flag behaviour because - in some cases - this is rather more involved.

The interface to the Stump ALU can be derived from the Verilog header provided (Stump_ALU.v) and is illustrated in Figure 1. Here, the inputs to the ALU are:

- operand_A and operand_B – two 16-bit signed input operands that are operated on by the ALU,
- func – 3-bit value that specifies the required operation performed using operand_A and operand_B,
- c_in – the carry in to the ALU, which comes from the status register, and
- csh – the carry out from the shifter, which is used only for logical functions where the flag update bit 'S' is set.

The outputs from the Stump ALU are:

- result – the 16-bit result of the required operation, and
- flags_out – the state of the status flags after the operation has been performed.

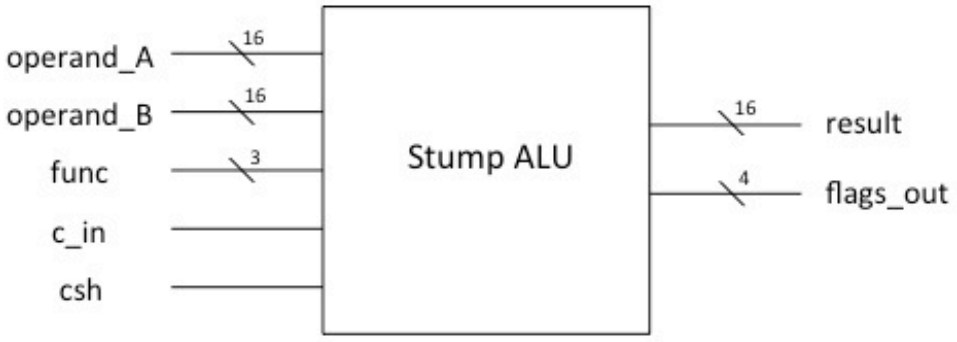

Figure 1: Stump ALU interface

There are eight possible 'instructions' as specified by the three most significant bits of a Stump op-code [15:13]. Figure 2, which lists the different instructions, the required ALU function, and the operands being operated on for each instruction. The ALU function is determined externally by the control logic, which determines the operation by decoding the current instruction held in the IR.

Code	Mnemonic	ALU function	operand_A	operand_B
000	ADD	add	srcA	srcB or immediate
001	ADC	adc	srcA	srcB or immediate
010	SUB	sub	srcA	srcB or immediate
011	SBC	sbc	srcA	srcB or immediate
100	AND	and	srcA	srcB or immediate
101	OR	or	srcA	srcB or immediate
110	LD/ST	add	srcA	srcB or immediate
111	Bcc	add	R7 (PC)	immediate

Figure 2: ALU functions

Within the ALU the value of flags_out is always set for all operations. The system flag register (outside the ALU) is updated only when an instruction requires the flag register to be updated – this is handled externally by the control logic.

It is worth reiterating what the flags indicate:

N - the negative flag is set if the ALU result is negative when interpreted as a 2's complement number.

Z - the zero flag is set if the result is zero.

V - the overflow indicates that the result from an addition/subtraction is 'wrong' if interpreted as a two's complement number.

C - the carry indicates that the result from an addition/subtraction is 'wrong' if interpreted as an unsigned number.

The first two flags are easy to derive and apply to both arithmetic and logical functions. The last two flags are somewhat harder but are only relevant for arithmetic operations. Figure 3 summarises the conditions under which the various condition bits are set, where S is a 16-bit result of an arithmetic or logic operation, i.e. the ALU result, and C_{14} and C_{15} are the carry out from bits 14 (into bit 15) and 15 (the carry out) respectively of the arithmetic operation.

In the case of a subtract operation (SUB or SBC) the carry out from the ALU operation must be inverted. In the case of a logical instruction the carry flag is determined by the status of csh. csh will be 0 unless it is a Type 1 instruction containing a shift operation, in which case the carry flag may be updated with a value other than 0.

The ADC instruction adds the value of operand_A to operand_B along with c_in = C (the carry flag). Both SUB and SBC should be implemented as 2's complement addition operations and not as subtraction operations so you gain more understanding of 2's complement arithmetic – marks will be deducted for implementations using a subtraction operation!

SUB is implemented as an add of operand_A and operand_B inverted +1, to give a 2's complement inversion of operand_B. The SBC operation is implemented as an add of operand_A with operand_B (or immediate) inverted along with

c_in = $\bar{C}$ (carry flag inverted). Why? If the carry flag is currently set then the result needs to be reduced by 1 to form a borrow.

Instr	Status of flags_out			
	N	Z	V	C
ADD	S_{15}	S=0	$C_{14} != C_{15}$	C_{15}
ADC	S_{15}	S=0	$C_{14} != C_{15}$	C_{15}
SUB	S_{15}	S=0	$C_{14} != C_{15}$	$\bar{C}_{15}$
SBC	S_{15}	S=0	$C_{14} != C_{15}$	$\bar{C}_{15}$
AND	S_{15}	S=0	0	csh
OR	S_{15}	S=0	0	csh
LD/ST	x	x	x	x
BR	x	x	x	x

Figure 3: Status Flags for the Stump

Some examples serve to illustrate the status of the flags for unsigned and signed representations, as illustrated in the tables in Figures 4 and 5. In both tables the operations are the same. The parenthesised decimal numbers indicate the way the values could be interpreted. Some of the answers are 'wrong' - different in each set as indicated by the 'V' and 'C' flags, emboldened appropriately. Note that the zero flag is only really meaningful if the appropriate 'overflow' has not occurred.

Function	Operand A	Operand B	Result	Flags
ADD	0000 (0)	0001 (1)	0001 (1)	**....**
ADD	FFFF (65535)	0001 (1)	0000 (0)	**.Z.C**
ADD	FFFE (65534)	0001 (1)	FFFF (65535)	**N...**
ADD	FFFF (65535)	0002 (2)	0001 (1)	**...C**
ADD	7FFF (32767)	0001 (1)	8000 (32768)	**N.V.**
ADD	7FFF (32767)	0002 (2)	8001 (32769)	**N.V.**
ADD	8000 (32768)	8000 (32768)	0000 (0)	**.ZVC**
ADD	FFFF (65535)	FFFF (65535)	FFFE (65534)	**N..C**
ADD	A000 (40960)	A000 (40960)	4000 (16384)	**..VC**
SUB	1234 (4660)	1234 (4660)	0000 (0)	**.Z..**

Figure 4: Unsigned interpretation

Function	Operand A	Operand B	Result	Flags
ADD	0000 (0)	0001 (1)	0001 (1)	**....**
ADD	FFFF (-1)	0001 (1)	0000 (0)	**.Z.C**
ADD	FFFE (-2)	0001 (1)	FFFF (-1)	**N...**
ADD	FFFF (-1)	0002 (2)	0001 (1)	**...**C
ADD	7FFF (32767)	0001 (1)	8000 (-32768)	**N.V.**
ADD	7FFF (32767)	0002 (2)	8001 (-32767)	**N.V.**
ADD	8000 (-32768)	8000 (-32768)	0000 (0)	**.**Z**V**C
ADD	FFFF (-1)	FFFF (-1)	FFFE (-2)	**N..**C
ADD	A000 (-24576)	A000 (-24576)	4000 (16384)	**..V**C
SUB	1234 (4660)	1234 (4660)	0000 (0)	**.Z..**

Figure 5: Signed interpretation

The carry flag is derived in the same way as other carries in an adder. However, there is a special case when performing logical operations (see above). As a Verilog expression the simplest method of extracting the carry flag is to extend the addition to 17 bits inside the ALU, with the least significant 16 bits forming the result and bit 16 forming the carry out.

There are a few ways that the status of the overflow flag can be determined. One way is to look at the carry in to the msb of the result (from bit 14 into bit 15) and the carry out (of bit 15). If the two values differ then an overflow condition will have occurred. Why? You figure it out! However, this approach requires you to calculate the carries, which may be more difficult to implement and ultimately more expensive in hardware. An alternative method is to look at the signs of the two input operands and consider whether an overflow could be observed or not. For example, if we add two positive numbers the result may overflow, but if we add a positive number and a negative number then we can never observe an overflow condition; as both values are in range, then the result must be in range. If we can identify those cases where an overflow may occur and test whether the sign of the result is what we would expect, then we can easily determine whether an overflow has occurred or not. This approach may be less expensive in hardware, and possibly easier to implement.

In this and the following exercises you will be developing a complete working Stump processor from skeletal component/modules that we have provided for you. To start, run Cadence ("start_cadence COMP22111"). Make sure the work library is set to Stump (File->Set Work Library-> Stump). The 'light bulb' should be on the Stump directory in the right-hand pane.

You are required to produce a Verilog module for the Stump ALU that should provide all the necessary functions for your Stump processor – a file, Stump_ALU.v, has been provided for you. It is important that you do not change the interface to the module (as illustrated in Figure 6) as it is used in the test template we supply. Your code in the module should be synthesizable, as you will implement this on the FPGA in a later exercise.

```
module Stump_ALU (input          [15:0]    operand_A,
                  input          [15:0]    operand_B,
                  input          [2:0]     func,
                  input                    c_in,
                  input                    csh,
                  output reg     [15:0]    result,
                  output reg     [3:0]     flags_out);
endmodule
```

Figure 6: Stump ALU interface definition

Top Tip – Ease of implementation

- The '`case`' statement may be very useful.
- Values may be '`assign`'ed values, but you will need to change the declaration of the outputs from the module from `reg` to `wire` (this is the only element of the interface you can change).
- Structure you code appropriately. You could use a single `always` block, or you could use hierarchy to break up your design.

We have provided a test module, Stump_ALU_test.v, for you that tests all the ALU operations with values that can generate legal combinations of flags. It may be worth looking at the test stimulus file as it uses some Verilog features that may be new to you. The test module operates by setting up the test values in arrays and then iterating through these for each ALU operation. (The address and branch calculations are omitted because they are repetitions of ADD; modify the file to add them if you like.) The iteration code comprises nested 'for' loops, which should look familiar. These are useful for describing behaviour, although they will not be synthesized into hardware. In some cases variables from the module being tested have been used. 'func' is the 3-bit function code specifying the ALU function. It can be used in this case because only six functions are tested.

Question: What would happen if you tried to do the following?

```
for (func = 0; func < 8; func = func + 1)
```

if 'func' used all eight cases? (Note that 'func' is a 3-bit value)

In other cases separate variables have been defined. These can be declared as 'integers' for this *behavioural* code although these are not generally synthesizable.

The results may be displayed and checked on screen but it may be easier to test them 'off-line'. The results are output to a dedicated file using $display(), which is created in your 'run' directory unless you specify otherwise. It is possible to output - slightly more simply - to 'stdout' using the $display() system task. In this Cadence setup this is directed to a file ~/Cadence/COMP22111/ncsim.log; this file is also used for other simulation status information.

There is also a Verilog **task** used to output your results to a file "ALU_test_out.txt". If your implemented ALU is working correctly then you should observe the same output compared to the expected results, which can be found in the file

/opt/info/courses/COMP22111/simfiles/ALU_test_out.txt

You may like to alter or enhance the stimulus file to do the checking automatically; you could manage here 'by eye' but automation is essential when the simulation trace may be many megabytes long and you need to check it repeatedly.

Your ALU module must be synthesizable. Even though we are not intending to download this module on its own it can be run through the Xilinx tools to check for possible problems. Select the Stump_ALU module and try the SYN-CHECK button; if this is not synthesizable this will be reported at the end of the log.

[A trial design needed about 5% of the FPGA logic 'Slices' and had an estimated delay of 7.3 ns; yours may be better or worse than this - it doesn't matter much as long as it's not vastly bigger.]

It is important that you document your source files by updating header information and commenting your code. This will benefit you later when you look back at your code and will also help others understand your code. We will look at your commenting during the offline marking.

Submission: ex2a – Stump ALU

Once you have completed your design and are confident that it works correctly by comparing your test output with the "golden" test data, then you can submit your design using the submit script "22111submit" before the deadline for the exercise. Submit as ex2a.

Print out a labprint for the exercise via the submit script and answer the questions before you demonstrate the exercise in your nest scheduled lab.

If you complete the exercise before the deadline and have plenty of time to spare then you can have a go at the optional exercise: the Stump shifter.

You have now completed Exercise 2

Top Tip - Sign extension and concatenation

"Sign extension" (in two's complement notation) is simply the propagation of the most significant (sign) bit into any appended more significant bits. For example, sign extending from 16 to 32 bits:

- 1278 would become 00001278
- ABCD would become FFFFABCD

There are a couple of features of Verilog which may be useful when, for example, sign extending a number.

```
wire  [7:0]  a;
wire  [15:0] b, c;
wire  [3:0]  d;

// Values can be concatenated by bracketing them together:
assign b = {8'h00, a};    // Zero-extend a to 16 bits
                          // (Not, strictly, necessary but good
                          // practice)

// Values can be repeated by prefixing the sequence with a
// number
assign c = {2{a}};        // Duplicate byte ‘a’ in both halves
                          // of ‘c’

// Subfields can be extracted by specifying bit ranges
assign d = a[7:4];        // Take upper nibble of ‘a’
```

These should be strong enough hints …