Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering
Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group_17.33
Andrea Casalino, Flavio Tanese

February 16, 2018

Contents

CHAPTER 1

Introduction

1.1 Requirements

It has been chosen to implement the pro version of the DLX microprocessor.
The basic requirements are:

- Basic structure of the pipelined datapath;
- A control unit to implement a subset of the instruction set;
- The synthesis of the final microprocessor;
- The physical design of the DLX.

Added features:

- High performance multiplier;
- Sum and multiplication Floating Point Unit.
- Microprogrammed control unit

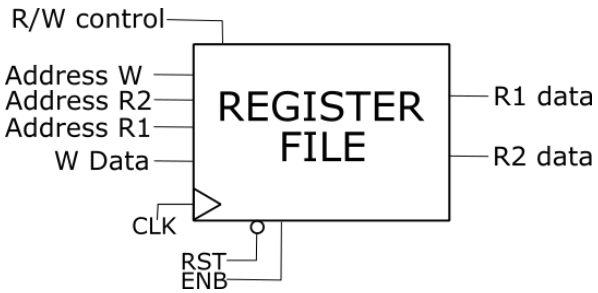
CHAPTER 2

Fetch and Decode

2.1 Fetch

During this stage the processor is retrieving an instruction from the instruction RAM memory. In order to simulate this the memory must be initialized with a process which reads from a file named "test.bin" before actually attempting any read from it.
The actual hardware implementing the fetch stage is quite simple: a program counter is used as pointer to memory and incremented each clock cycle, while a couple of registers take care of sampling the output of the memory (i.e. the instruction retrieved) and of the program counter for the next stages. In addition to the general reset signal affecting the whole processor, the fetch stage is governed by an enable signal: when the stage is disabled, the output instruction is NOP and the corresponding program counter is set to 0. A multiplexer selects the next program counter between the increment of the current one and an external one coming from the execute stage, thus implementing jumps.
Jumps and branches are detected in the following stage that disables instruction fetch for two clock cycles: this means that any branch or jump is followed by two NOPs, the first one stalling the pipeline while waiting for the execute stage to calculate the target address and the second one stalling the pipeline while waiting for program counter to be updated.

2.2 Decode



The decode stage takes care of decoding the instruction fetched in the previous cycle and providing the correct operands to the execute stage. There are thus two main components: the control unit, handling the behaviour of the rest of the circuit depending on the instruction it received, and the register file, sending operands coming from previous instructions to the next stage.
The control unit is implemented as a horizontal microcode unit: to avoid having multiple ports on the microcode ROM, the signals for all stages are provided at the same time to some delaying logic, so that the micro-program counter does jump on each clock cycle and the unit becomes essentially equivalent to a hardwired control unit. Different operations are mapped to different locations in memory: conflicts (due to R-type and F-type instructions having their function mapped on the same location as other instructions) are resolved with an intermediate look-up table offsetting F-type instructions and relocating I-type and J-type instructions to unused memory cells. The register file is a simple unwindowed architecture providing 32 general purpose registers and 32 floating point registers, with two reading ports and one writing port. Each port is governed by a control signal (read ports are always on in our implementation) and includes a 6-bit address (floating point flag plus 5-bit register id) and the 32-bit data port.

CHAPTER 3

Arithmetic and Logic Unit

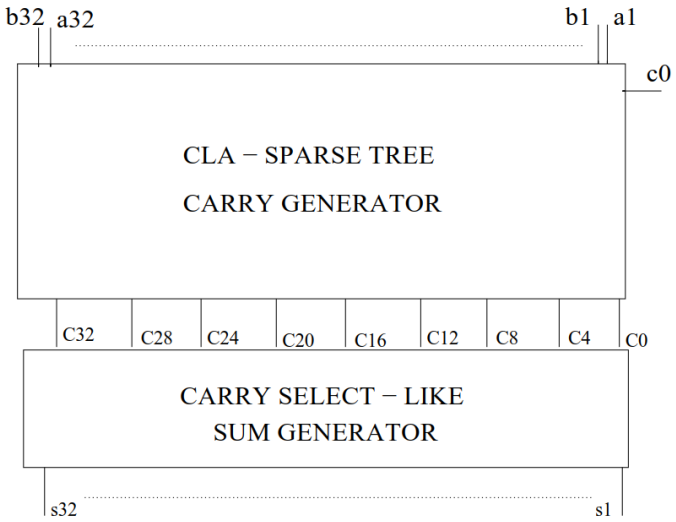
The execute stage is taken totally in charge by the arithmetic and logic unit, with very minor additional hardware in the form of multiplexers for operand selection and output registers to implement the pipeline. Depending on the value of logic_op and output_sel signals the ALU will perform different operations.

output_sel	logic_op	output	output_sel	logic_op	output	output_sel	logic_op	output
000	–	ADD	100	010	XOR	110	111	SLL
001	–	FP_ADD	100	110	XNOR	110	110	SRL
010	–	MUL	101	000	A>B	110	101	SLA
011	–	FP_MUL	101	110	A≥B	110	100	SRA
100	000	AND	101	001	A=B	110	001	ROL
100	100	NAND	101	101	A≠B	110	000	ROR
100	001	OR	101	010	A<B	111	–	B
100	101	NOR	101	100	A≤B			

As no instruction operates on the overflow bit, it was decided to ultimately drop it by leaving the output open: in case additional instructions were to be introduced, an overflow flag register could be used instead for more sophisticated branch options.

3.1 P4 Adder

The adder is the implementation of a very fast carry lookahead adder using a sparse tree for the generation of every 4th carry. For each 4-bit block, two small ripple carry adders compute the sum of the operands assuming the input carry is '0' or '1' respectively and by the time the two sums are ready, the tree outputs are used to select the correct one. The final output is simply the concatenation of all 4-bit selected results.



While this component is very fast, it is also pretty expensive in terms of area and complexity: it would be better used in a more complex architecture supporting out-of order operation execution and multiple pipelines of different length. Usually is not a good idea to have an extremely fast component that has to wait for much slower components to produce results. Still, as the architecture was ready for use, we decided to just plug in the component into the ALU.

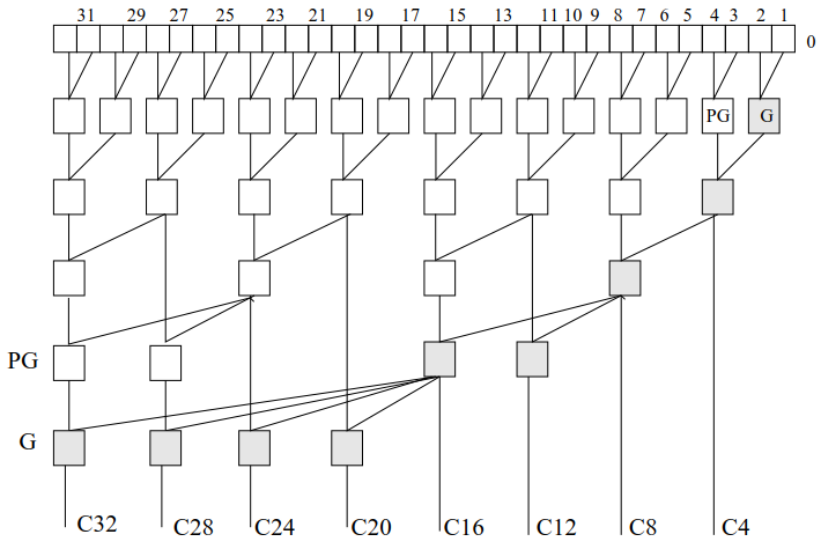
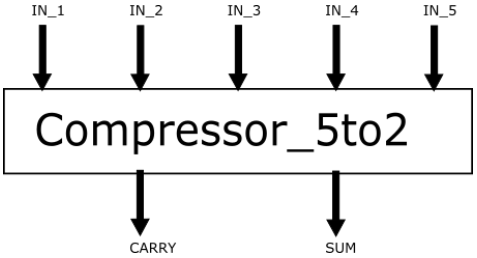


Figure 3.1: Sparse tree configuration

3.2 Multiplier

The multiplier designed is based on both the *Wallace tree* and *Booth* algorithms.

A new basic block called compressor_5to2[1] is added. The concept is similar to the one of the *Carry Save Adder*: it takes five different inputs and compacts them into two outputs (Sum, Carry).



Once the basic function of the block is clear, it is possible to describe the multiplier structure (see figure 3.2). For the sake of simplicity let's say from now on that we are going to perform $P = A \cdot B$.

The first stage is just the application of radix-4 Booth algorithm [2]. The number A is left shifted (multiplied by a power of 2) in order to properly feed the different multiplexer, while the number B is used by the Booth encoder to generate the select signals for the MUXes. The aim of the Booth's algorithm is to reduce the number of partial products that will be added together in the next stages. Actually these numbers can be both positive and negative (expressed in 2's compliment), which means that is needed to keep also some informations about the signs of the partial products. This issue is managed using XOR gates and storing sign infos in 4 sign vectors that will be added with the partial products.

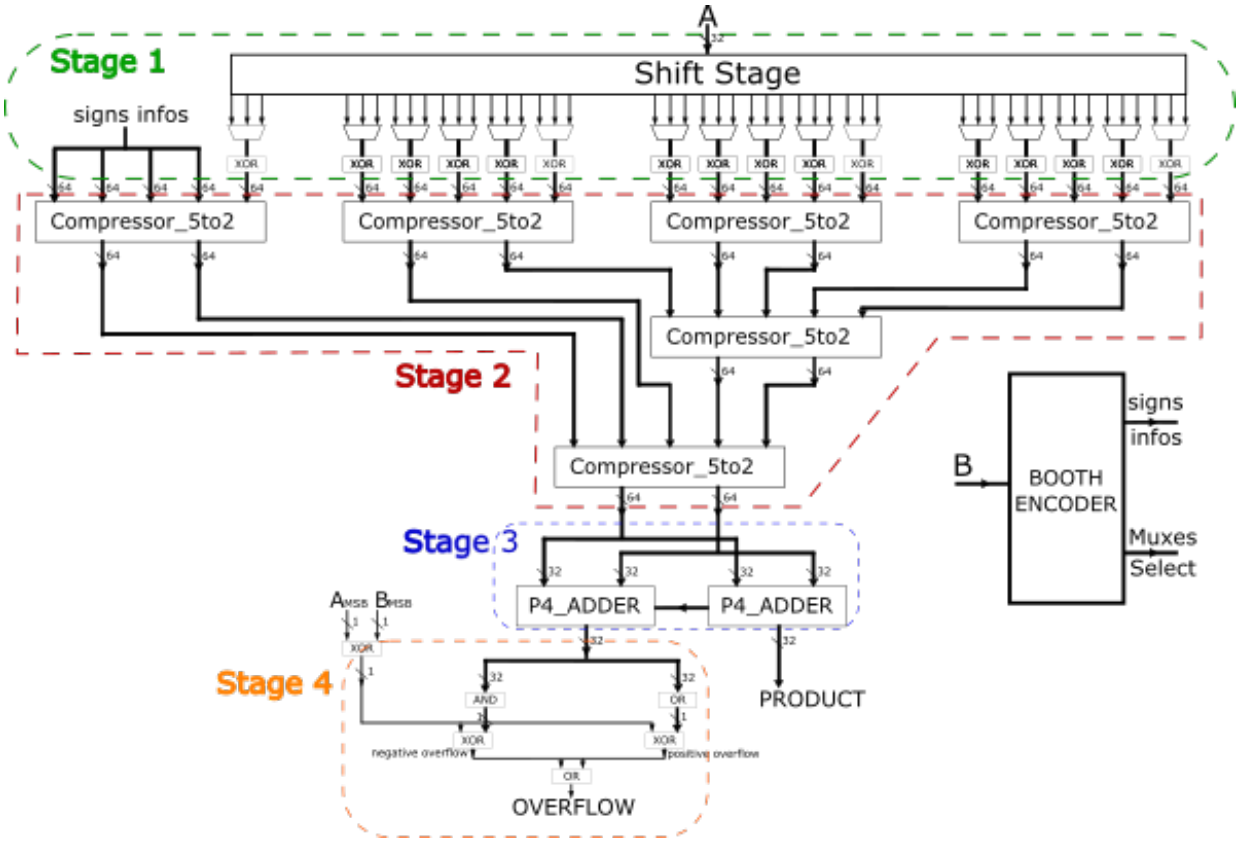


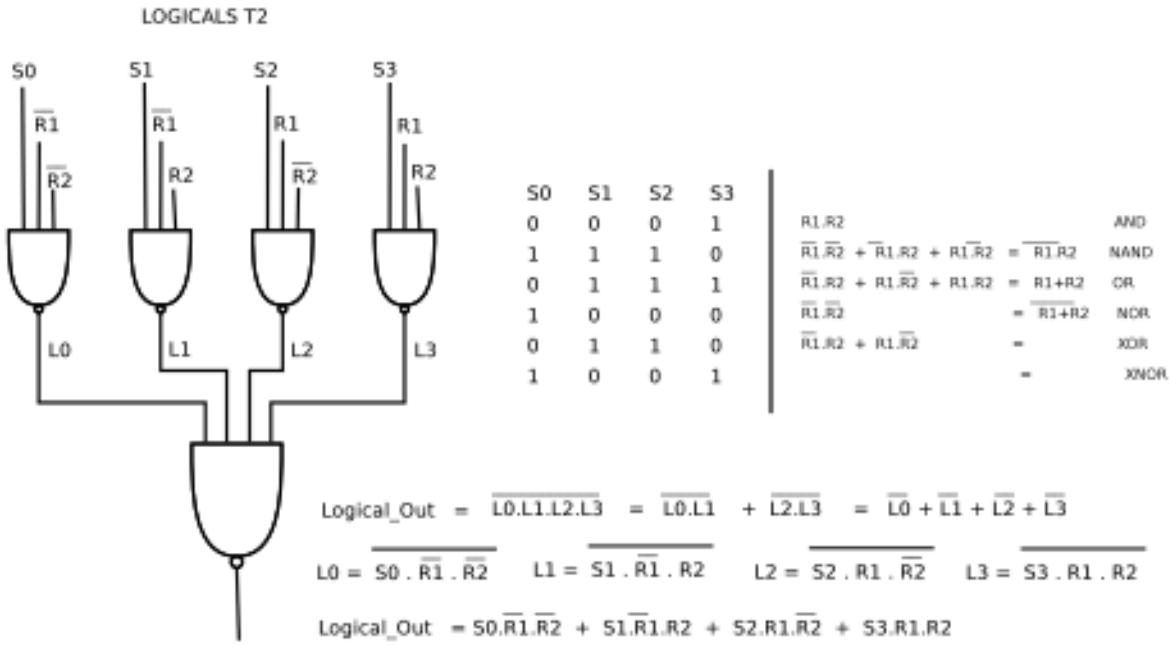
Figure 3.2: Multiplier structure

[1] Appendix B
[2] Appendix A

In the second stage all the Booth outputs are summed together using the new block introduced before. In this way it's possible to rapidly obtain partial sums.
The third stage is the sum of the two final partial sums in order to obtain the final Product. This is done using two P4 Adders cascaded. This could be the final result, but since a 32 bit system is used, a proper truncation on the result must be performed. A correct outcome is guaranteed keeping just the lower 32 bits and applying some logic to the upper 32 in order to notify an overflow occurrence.
The overflow logic (fourth stage) is quite simple. The MSB of both A and B are compared in order to identify the expected sign of the result P. At this point it is possible to distinguish between positive and negative overflow.
If a positive result is expected and there is at least a logic '1' in the upper 32 bits, a positive overflow occurred.
Otherwise if a negative result is expected and there is at least a logic '0' in the upper 32 bits, a negative overflow occurred.

3.3 Logic Unit

The logic unit used is the same of the OpenSPARC T2. It is implemented with two NAND gates levels: the rst one has four 3-inputs NANDs, each input is 32-bits wide; the second level has a 4-inputs NAND, whose inputs are the outputs of the rst level gates. The first level NANDs receive as input all possible combinations of affirmed and negated operands and an additional selection signal: particular patterns of the selection signals over the different gates produce all possible logic operations as outputs.



3.4 Shifter and Comparator

This two blocks have a behavioral implementation in order to reduce code size.

3.5 Floating Point Unit

The floating point unit implements IEEE-754 floating point single precision addition/subtraction and multiplication with truncation.

3.5.1 Adder/Subtractor

Floating point addition is quite more complicated than the standard integer or fixed point equivalent: operands need to be normalized to the exponent of the greater one, then added/subtracted and then re-normalized if necessary. The code implementing the operation is behavioural, but based on an RTL implementation:

- A comparator compares the exponent bits of the input operands (from here on A and B) and decides wether to swap them or not, while a small subtractor determines the difference between exponents.
- B, now with exponent smaller or equal to A, gets its mantissa (with explicitly represented hidden bit) shifted right accordingly to the previously calculated difference in order to have the same exponent.
- A and B get their mantissas summed/subtracted in 2's complement (one more bit is added). This stage also checks particular cases like addition/subtraction of infinities or NaNs.
- If a sum was performed and the result is not positive, the exponent of the result is the exponent of A incremented by 1 and the mantissa is shifted right by 1; else, the mantissa is correct and the exponent is the exponent of A.
  If a subtraction was performed and the result is negative, the result gets complemented to get a positive mantissa and the sign bit is flipped. Eventual leading zeroes are removed with a left shift and the exponent is decreased accordingly, taking care to check for underflows before the operation.
- Depending on the operation and the relation between the input operands, the sign bit might be flipped again (as an example, A-B with B having a greater exponent is interpreted as -(B-A) as the operands have to be swapped).

3.5.2 Multiplier

The multiplier is much easier to implement than the adder/subtractor: exponents of the two operands get their bias removed, are added and then biased again to get the exponent of the result. Mantissas are multiplied normally without any shifting or alignment and if the result needs one more bit, it gets shifted right by 1, while the exponent gets increased by 1.

CHAPTER 4

Memory and Writeback

4.1 Memory

The memory stage is quite straightforward: the main component is of course a data memory (sharing the enable signal of the entire stage) that at every cycle either reads from or writes to the address provided by the execute stage. The input data to the memory are provided from the second read port of the register file and conveniently delayed, thus allowing the data from the first read port and the immediate field to be used by the ALU in order to compute the destination address.
Since it is very possible (and indeed extremely common) that the value we need at the output of the stage is just the result of the execute stage, a register delays the result and a multiplexer selects either it or the DRAM output as the stage output.

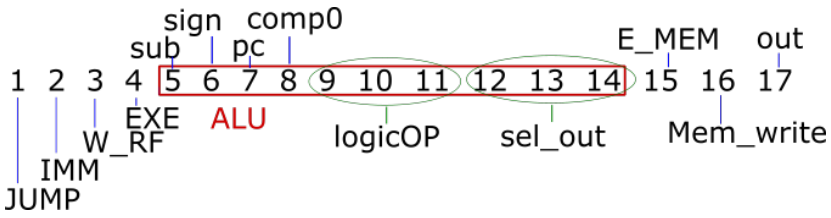
4.2 Writeback

The entirety of the writeback stage is implemented internally to the register file. It is a very simple stage, just taking a write control signal and an address from the control unit and writing the result in the correct register if the write control is active.

CHAPTER 5

Control Unit

To implement the control unit a microcode approach has been taken. The binary value of the instruction is just used as a pointer to a vector which contains the bit instruction for the whole DLX. A new instruction is read each cicle, hence a new instruction vector is generated each clock cycle. Then it's delayed by registers in order to properly feed the correct stage of the DLX in the right moment.
An instruction vector is 17 bits long.



- JUMP: is set only in case of a jump instruction
- IMMEDIATE: is set only in case of immediate instruction
- W RF: is set when is required to write in the Register File (usually any instruction except JUMP)
- EXE: is an enable for the ALU (always set)
- sub: set only for the subtraction operation
- sign: set when the op is signed
- pc: set when a new pc has been calculated
- comp0: set when a comparison with zero is needed
- logic op: select which kind operation must be done by the ALU
- sel out: select the ALU output since all op are done in parallel
- E mem: set the memory needs to be used
- mem write: enables the write operation to the DRAM
- Out: if it's unset the output is taken from the ALU, otherwise it's taken from memory

A few additional controls that depend on instruction type and do not affect the pipeline significantly (is_float flags for the register file) are set by simple hardwired logic, as they are shared by a broad variety of operations and their implementation in microcode would add much greater area overhead. The only special cases that do not have all flags set just depending on the operation type are in fact floating point move operations, that have R-type opcode but operate on F−type (or mixed) registers.

5.1 Instruction set

	Instr	Structure	Meaning	Microcode
	NOP	NOP	No operation	0000 0000 000 000 000
R_ TYPE	ADD	ADD RD, RS1,RS2	R[RD] = R[RS1] + R[RS2]	0011 0100 000 000 100
	ADDU	ADD RD, RS1,RS2	R[RD] = R[RS1] + R[RS2]	0011 0000 000 000 100
	SUB	SUB RD, RS1,RS2	R[RD] = R[RS1] - R[RS2]	0011 1100 000 000 100
	SUBU	SUB RD, RS1,RS2	R[RD] = R[RS1] - R[RS2]	0011 1000 000 000 100
	MULT	SUB RD, RS1,RS2	R[RD] = R[RS1] · R[RS2]	0011 0000 000 010 000
	AND	AND RD, RS1,RS2	R[RD] = R[RS1] and R[RS2]	0011 0000 000 100 100
	OR	OR RD, RS1,RS2	R[RD] = R[RS1] or R[RS2]	0011 0000 001 100 100
	XOR	XOR RD, RS1,RS2	R[RD] = R[RS1] xor R[RS2]	0011 0000 010 100 100
	SLL	SLL RD, RS1,RS2	R[RD] = R[RS1] sll R[RS2]	0011 0000 111 110 100
	SRL	SRL RD, RS1,RS2	R[RD] = R[RS1] srl R[RS2]	0011 0000 110 110 100
	SRA	SRA RD, RS1,RS2	R[RD] = R[RS1] sra R[RS2]	0011 0000 100 110 100
	SGE	SGE RD, RS1,RS2	R[RD] = 1 if R[RS1] ≥ R[RS2] else 0	0011 0000 110 101 100
	SGT	SGE RD, RS1,RS2	R[RD] = 1 if R[RS1] > R[RS2] else 0	0011 0000 000 101 100
	SLE	SLE RD, RS1,RS2	R[RD] = 1 if R[RS1] ≤ R[RS2] else 0	0011 0000 100 101 100
	SLT	SLE RD, RS1,RS2	R[RD] = 1 if R[RS1] < R[RS2] else 0	0011 0000 010 101 100
	SNE	SNE RD, RS1,RS2	R[RD] = 1 if R[RS1] != R[RS2] else 0	0011 0000 101 101 100
	SEQ	SNE RD, RS1,RS2	R[RD] = 1 if R[RS1] == R[RS2] else 0	0011 0000 001 101 100
	MOVFP2I	MOV RS1, FS1	R[RS1] = F[FS1]	0011 0000 001 101 100
	MOVI2FP	MOV FS1, RS1	F[FS1] = R[RS1]	0011 0000 001 101 100
I_TYPE	ADDI	ADDI RD,RS1,IMM16	R[RD] = R[RS1] + IMM16	0111 0100 000 000 100
	ADDUI	ADDI RD,RS1,IMM16	R[RD] = R[RS1] + IMM16	0111 0000 000 000 100
	SUBI	SUBI RD,RS1,IMM16	R[RD] = R[RS1] - IMM32	0111 1100 000 000 100
	SUBUI	SUBI RD,RS1,IMM16	R[RD] = R[RS1] - IMM32	0111 1000 000 000 100
	ANDI	ANDI RD,RS1,IMM16	R[RD] = R[RS1] and IMM32	0111 0000 000 100 100
	ORI	ORI RD,RS1,IMM16	R[RD] = R[RS1] or IMM32	0111 0000 001 100 100
	XORI	XORI RD,RS1,IMM16	R[RD] = R[RS1] xor IMM32	0111 0000 010 100 100
	SLLI	SLLI RD, RS1,IMM16	R[RD] = R[RS1] sll R[IMM16]	0111 0000 111 100 100
	SRLI	SRLI RD, RS1,IMM16	R[RD] = R[RS1] srl R[IMM16]	0111 0000 110 100 100
	SRAI	SRAI RD, RS1,IMM16	R[RD] = R[RS1] sra R[IMM16]	0111 0000 100 100 100
	SGEI	SGEI RD, RS1,IMM16	R[RD] = 1 if R[RS1] ≥ R[IMM16] else 0	0111 0000 110 101 100
	SGTI	SGEI RD, RS1,IMM16	R[RD] = 1 if R[RS1] > R[IMM16] else 0	0111 0000 000 101 100
	SLEI	SLEI RD, RS1,IMM16	R[RD] = 1 if R[RS1] ≤ R[IMM16] else 0	0111 0000 100 101 100
	SLTI	SLEI RD, RS1,IMM16	R[RD] = 1 if R[RS1] < R[IMM16] else 0	0111 0000 010 101 100
	SNEI	SNEI RD, RS1,IMM16	R[RD] = 1 if R[RS1] != R[IMM16] else 0	0111 0000 101 101 100
	SEQI	SNEI RD, RS1,IMM16	R[RD] = 1 if R[RS1] += R[IMM16] else 0	0111 0000 001 101 100
	SW	SW RD,RS1,IMM16	MEM[R[RS1] + IMM16] = R[RD]	0011 0000 000 000 110
	LW	LW RD, RS1,IMM16	R[RD] = MEM[R[RS1] + IMM16]	0011 0000 000 000 101
J_TYPE	J	J IMM26	PC = PC + 4 + IMM26	1001 0010 000 000 000
	JAL	JAL LABEL	R31 = PC + 4; PC = PC + IMM26 + 4	1001 0010 000 000 000
	BEQZ	BEQZ RS1, LABEL	if R[RS1] == 0 then PC = PC + IMM16 + 4	1001 0011 001 000 000
	BNEZ	BNEZ RS1, LABEL	if R[RS1] != 0 then PC = PC + IMM16 + 4	1001 0011 101 000 000
F_TYPE	ADDF	ADDF FD, FS1, FS2	F[FD] = F[FS1]+F[FS2]	0011 0100 000 001 000
	SUBF	SUBF FD, FS1, FS2	F[FD] = F[FS1]F[FS2]	1001 0010 000 000 000
	MULTF	MULTF FD, FS1, FS2	F[FD] = F[FS1]·F[FS2]	1001 0010 000 000 000

CHAPTER 6

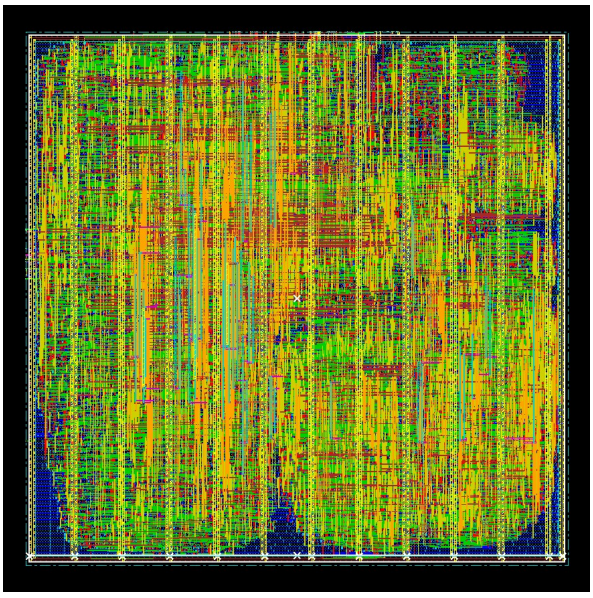
Synthesis and Physical design

6.1 Synthesis

The synthesis is performed using Synopsys. Not all blocks of our design can be implemented, hence only the Decode unit (containing the Control Unit) and the Execute unit (containing the ALU) have been physically designed. The result is running slower than the expectation, but the power consumption is way lower.

6.2 Physical Design

The physical design steps are the same used during lab06. The main difference is that the file .sdc was already generated during the synthesis.



APPENDIX A

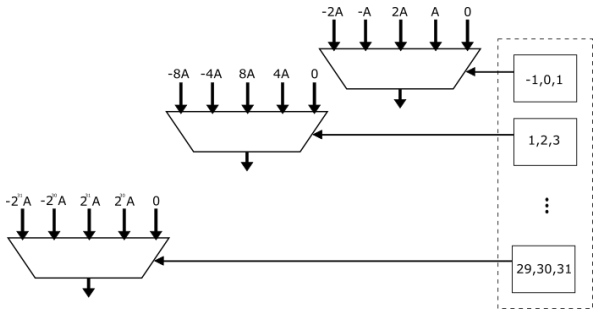
Radix-4 Booth's Algorithm

Booths Algorithm is based on the recoding of multiplied numbers. It is possible to reduce the number of partial products by half, by using the technique of radix 4 Booth recoding. The basic idea is that, instead of shifting and adding for every column of the multiplier term and multiplying by 1 or 0, we only take every second column, and multiply it by ±1, ±2, or 0, to obtain the same results.
Let's say that we want to perform the operation P ← A· B. We need to analyze B to define the partial products of A. In order to do this B is divided in blocks of three bits and analyzed in the method shown in the following table.

-1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

B_{i+1}	B_i	B_{i-1}	V_p
0	0	0	0
0	0	1	A
0	1	0	A
0	1	1	2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

An hardware implementation can be the one shown here below

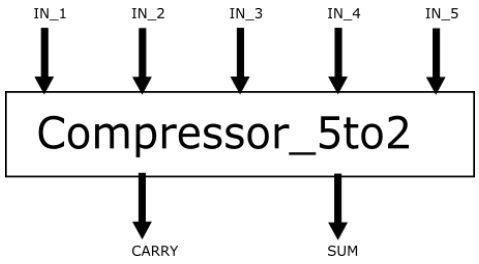


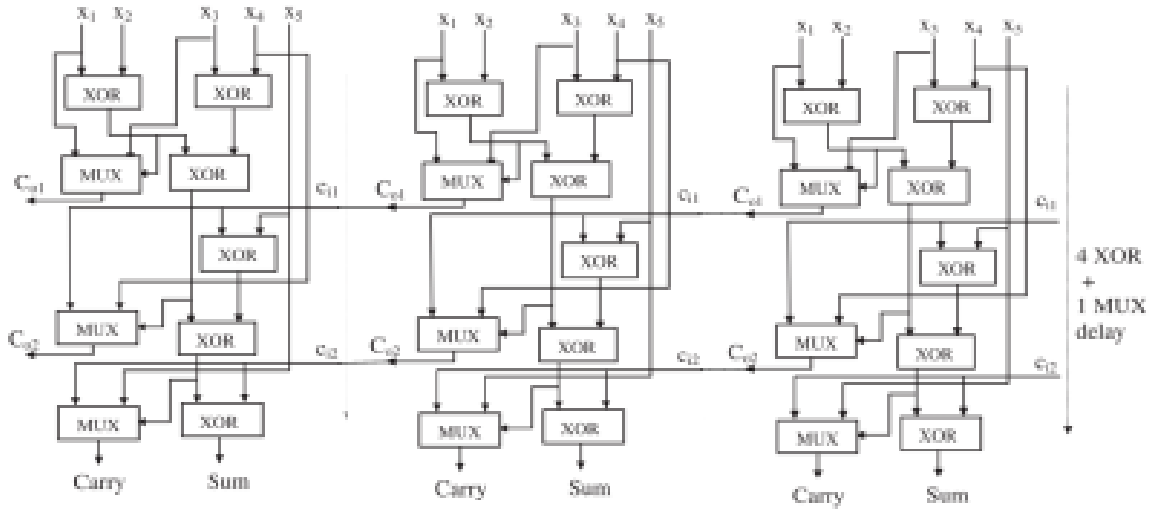
APPENDIX B

Compressor

First of all, it must be clear that this block refers to paper Menon and Radhakrishnan, "High performance 5 : 2 compressor architectures". This is the main component of the Multiplier, responsible to compress five different inputs into two outputs. This section aim is to summarise the internal functioning of the block.

Going inside it's possible to see that it is a cascade of blocks that just sum up the different inputs. What is actually very interesting is how signals are generated and propagated between the cascaded block. Looking at the image below it can be seen that one block will not need to wait for the carry generation of the previous one. This means that the carries are generated in parallel from the starting input and then added together to obtain the final sum and carry. This kind of block is capable to avoid the carry propagation delay, and this cuts a lot of the computation time needed to calculate partial sums in traditional multipliers.





Bibliography

[1] R. Menon and D. Radhakrishnan, "High performance 5 : 2 compressor architectures", *IEE Proceedings - Circuits, Devices and Systems*, vol. 153, no. 5, pp. 447–452, Oct. 2006, ISSN: 1350-2409. DOI: 10.1049/ip-cds:20050152.