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, Deborah Volpe

Authors: group_4
Giacomo Sansone, Giuseppe Silvestri, Arianna Valenza

July 13, 2023

Contents

CHAPTER 1

Introduction

The implementation of a pipelined RISC microprocessor with advanced functionalities is a remarkable feat in the realm of digital design and computer architecture. Our implementation of the DLX microprocessor, made of a controller and a datapath, includes sophisticated components such as a radix-4 Booth multiplier and a radix-4 SRT divider.

The DLX architecture, inspired by the MIPS, is the foundation of our implementation. Its simple instruction set enables efficient execution of complex programs, making it an ideal choice for a wide range of applications. However, the addition of advanced functionalities takes the DLX microprocessor to new heights, making it capable of handling computationally intensive tasks.

Our processor is a combination of a hardwired control unit and a datapath, working together to execute instructions and perform complex computations. On one hand, the control unit is responsible for generating the control signals that coordinate the operations within the datapath. On the other hand, the datapath is the core processing unit, consisting of arithmetic/logic units, registers and multiplexers. It carries out the actual calculations and manipulations of data under the influence of the control unit. The detailed explanation of the implementation of these components can be found in section 2.2 and section 2.1.

As for the pipeline, a static one has been implemented and consists of five stages: fetch, decode, execute, memory and write-back. By breaking down each instruction's execution into multiple stages, the pipelined datapath enables the concurrent execution of several instructions in different stages, resulting in high throughput.

The inclusion of a radix-4 SRT divider adds to the microprocessor's capabilities. The SRT division algorithm utilises a combination of shifts, additions and subtractions to perform division operations with exceptional speed and precision. We carefully studied and collected the theory behind it and the results of our work can be found in Appendix A. The employment of a radix-4 algorithm further increases the division efficiency by reducing the number of required iterations. The detailed explanation of the implementation can be found in Appendix B.

The radix-4 Booth multiplier integrated into the microprocessor is another noteworthy feature. By employing a combination of shifts and additions/subtractions, the multiplier reduces the number of required intermediate products and minimises the overall latency, quickly computing long multiplications. Its implementation is discussed in Appendix C.

An additional feature of our implementation is the inclusion of automated simulation and synthesis scripts, which streamline the design process. These scripts make it easier to verify and validate the processor's functionality and performance. This automation significantly reduces the manual effort required for simulation and synthesis tasks, enabling the user to focus more on design refinement and optimisation. Further details about the simulation and synthesis automation can be found in section 3.3 and section 3.4, respectively.

CHAPTER 2

Functional Schema

2.1 Datapath

The full picture of the datapath of out DLX can be seen in Figure 2.1; tt can be found in high quality inside `doc/diagrams` folder. In this section we are going to cover the interconnection between the different components of the various stages of the pipeline, highlighting some implementation choices. A full overview of the components can be found in section 3.2.

2.1.1 Fetch stage

During the fetch stage (shown in Figure 2.2), the address stored in the program counter (or PC) is used to access the instruction memory and read the next instruction to be executed. Although in the diagram the instruction memory is shown as if it were part of the datapath, it is external and the same holds for the dram.

The fetched instruction is stored in the `IR_FD` register; the `IR` registers of the pipeline are in charge of moving the instruction that is being executed through the stages, so that it can be used if needed (for example to know the destination register).

In the meantime, the controller receives only the opcode of the fetched instruction and it generates the control word that will be used. The control word is stored in the `CW_FD` register, so that it will be available in the next stage of the pipeline. The control word then flows through the `CW` registers: only the signals needed for the following stages are propagated.

The next program counter (so $PC + 4$) is saved in the `NPC_FD` register and will be propagated as well in order to be available for use in the next stages.

The program counter's content is written during the memory stage: it can be either the next program counter or another address computed during a jump/branch instruction. Th PC's value is updated only if the `PC_enable` signal is one, otherwise it keeps its old value. This is useful for multi-cycle instructions, during which the pipeline has to stop.

For the same reason, all the registers of the pipeline are enabled by the `single_cycle_enable` signal, so that they do not update during the execution stage of a multi-cycle instruction.

2.1.2 Decode stage

During the decode stage, all the possible operands for the execution stage are produced. At this moment, we do not know which operands are going to be needed: they could both come from the register file, one could be an immediate and the other might come from the register file, etc.

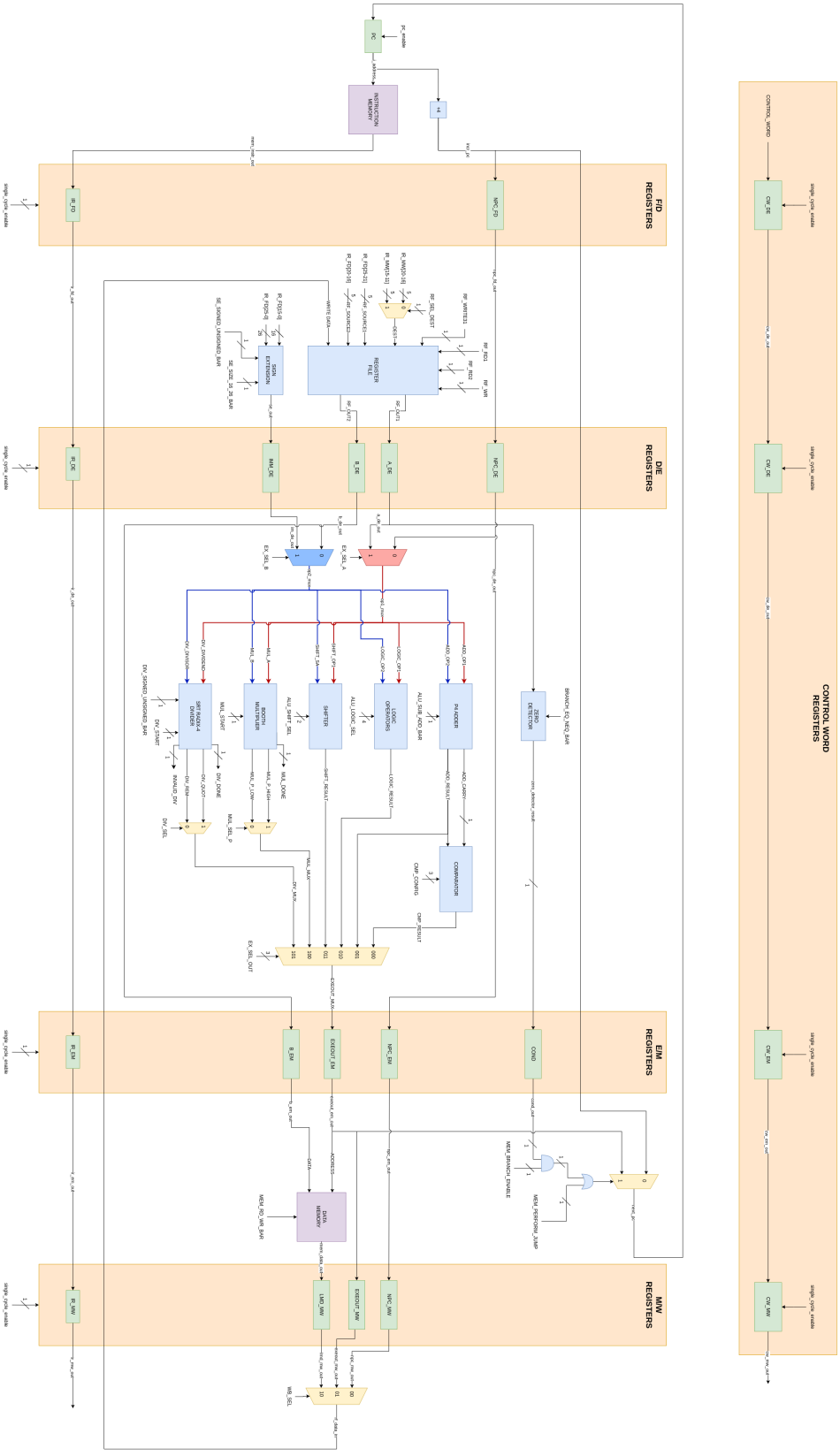

Figure 2.1: Datapath diagram

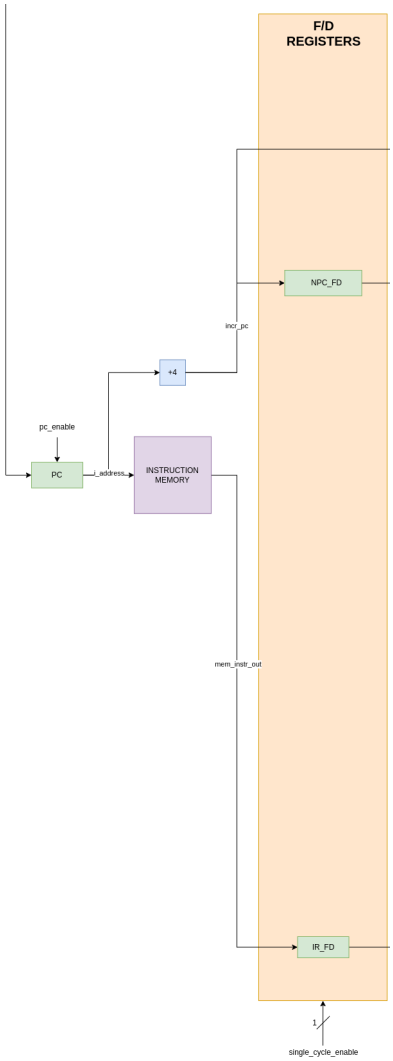

Figure 2.2: Fetch stage of the datapath

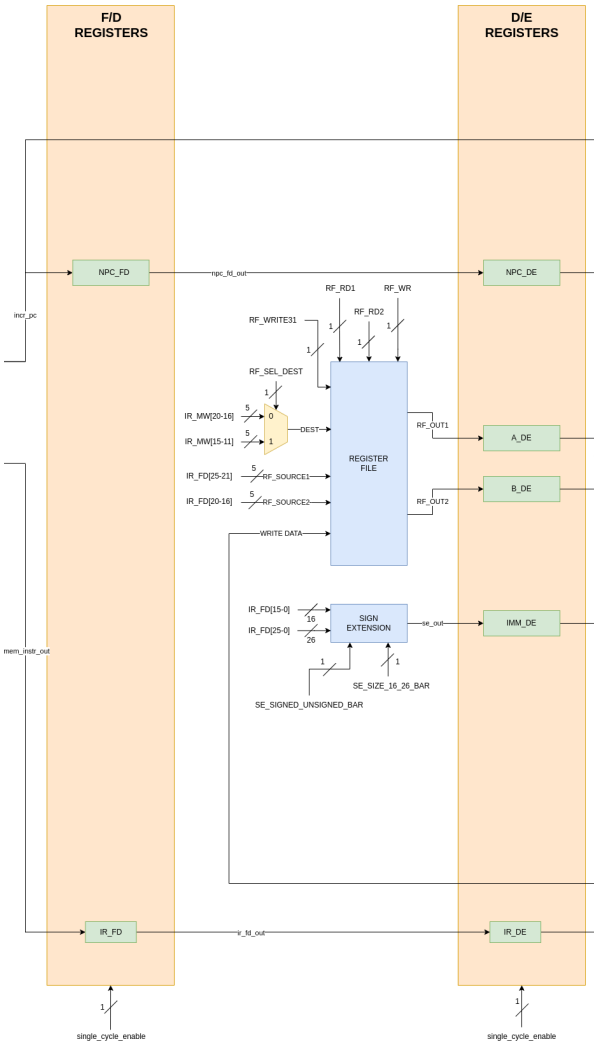

Figure 2.3: Decode stage of the datapath

We use `IR_FD[25:21]` and `IR_FD[20:16]` as read addresses for the register file. The read values are accessed through the `RF_OUT1` and `RF_OUT2` lines, and stored in the `A_DE` and `B_DE` registers. Read operations on the register file are performed only if the control signals `RF_RD1` and `RF_RD2` are one.

In the meantime, the sign extender produces the extension of the immediate value that may be used in the following stage. If we are dealing with an *I-type* instruction, the immediate is located in `IR_FD[15:0]`, and it will be extended as signed or unsigned number depending on the value of the control signal `se_signed_unsigned_bar` (it's sign-extended during arithmetic operations, and zero-extended during logic operations); in case of a *J-type*, the immediate is in `RF_ID[25:0]` and is always sign-extended. The extended immediate is stored in `IMM_DE`.

2.1.3 Execute stage

In the execution stage, shown in Figure 2.4, the operands are used to perform an operation using one of the available components.

The first two multiplexer decide which will be the operands, according to the control signals `EX_SEL_A` and `EX_SEL_B`. When `EX_SEL_A` is one, the first operand is the one read from the first port

of the register file, otherwise it is the next program counter, computed during the fetch stage of the same instruction that is now in the execution stage. When `EX_SEL_B` is one, the second operand is the previously extended immediate, otherwise is the one read from the second port of the register file.

The possible operations that can be performed are: addition, subtraction, shift (left or right, arithmetic or logical), logical operations (or, nor, and, nand, xor, xnor), multiplication or division.

In case a multiplication or a division starts, the pipeline is stopped until a finish signal is raised: should that happen, the controller will resume the pipeline after one clock cycle.

In case of a division, the possible output can be either the quotient or the remainder. Regarding the multiplication, we are multiplying two 32-bit numbers, so the result is 64 bits: for this reason, you can select either the upper part of the result (so its 32 MSBs) or the lower part (so its 32 LSBs).

The result of a subtraction is also used to compare the two inputs using a comparator. According to the `CMP_CONFIG` signals, the result is one if the requested comparison is true, otherwise it is zero. This component is often used to compute a condition based on which a branch might or might not be performed.

The signal to be sent as output is chosen thanks to the last multiplixer, driven by the `EX_SEL_OUT` signal. Both the next program counter and the B operand are also propagated through the pipeline, since they will be used in the following stage.

The value of the register `A_DE` is also used to check the condition of a branch using the zero detector. The output of this component is one if the operand is zero and we are performing a `beqz` operation or if the operand is not zero and we are performing a `bnez` operation (in these cases the branch is taken); in all other cases it is zero (meaning that the branch is not taken).

2.1.4 Memory stage

The memory stage shown in Figure 2.5 is used both to access the dram (in case of a load/store operation) and to compute the next value of the program counter.

The new program counter is:

- $PC + 4 + immediate$ if we are performing a branch and the condition is true (the immediate value stored inside the instruction is an offset relative to $PC + 4$);
- $PC + 4 + immediate$ if we are performing a jump or a jump-and-link;
- the value read from a target register if we are executing a `ret` instruction;
- $PC + 4$ in all the other cases.

If we need to perform a read operation from the dram, the signal `mem_rd_wr_bar` is one, otherwise it is zero. The address is computed in the execution stage by adding the content of a register with the immediate value in the instruction. The value to be written is stored inside `B_EM`; the read value is stored into `LMD_MW`.

2.1.5 Write back stage

In the write back stage, as shown in Figure 2.6, we write to one register in the register file in case the `RF_WR` signal is one. The destination register is located in `IR_MW[20:16]` or in `IR_MW[15:11]`, depending on the type of the instruction.

The value to be written can be:

- $PC + 4$, in case of a jump-and-link instruction;
- the value read from the memory, in case of a load instruction;
- the result of the execution stage, in all the other cases.

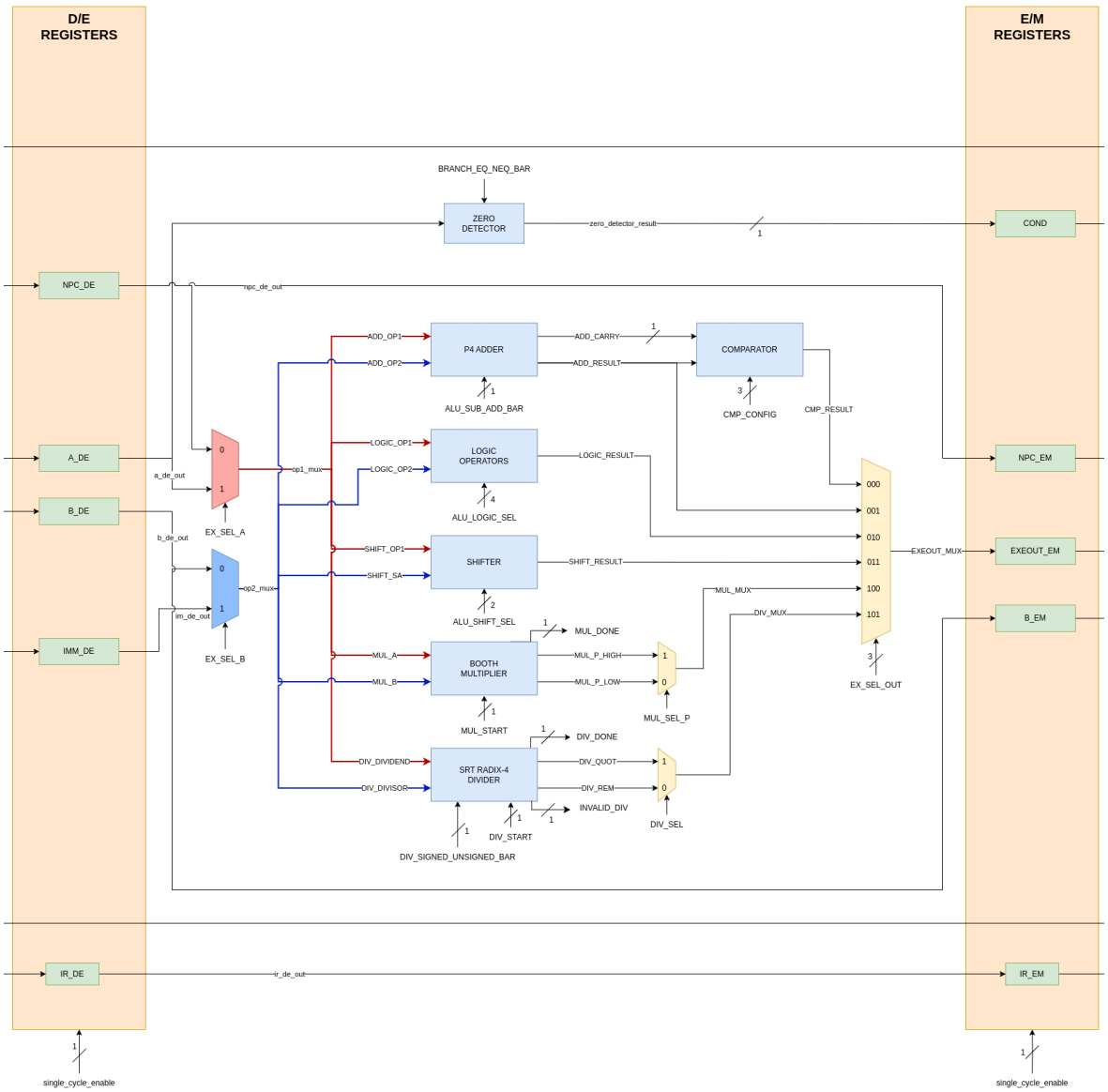

Figure 2.4: Execute stage in the datapath

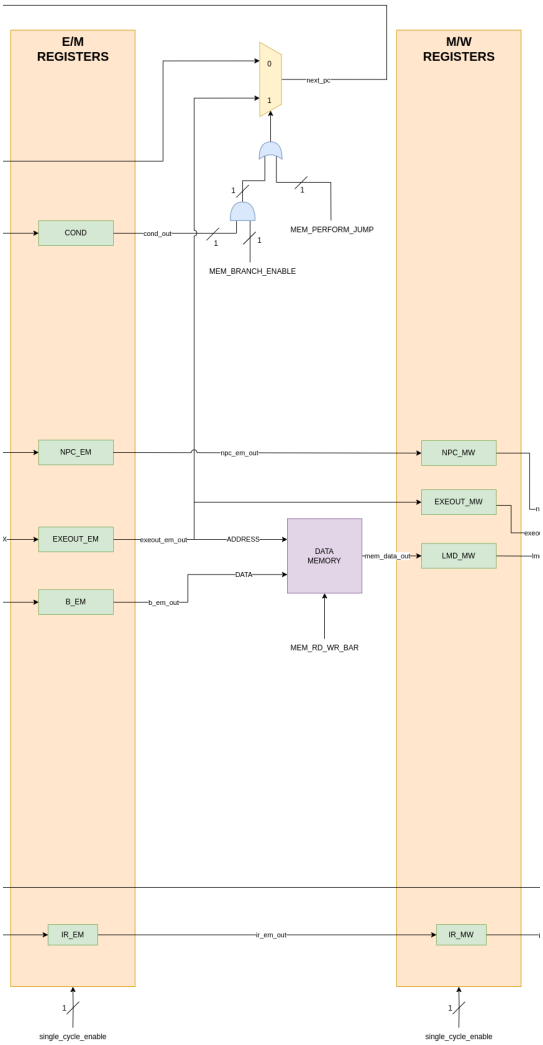

Figure 2.5: Memory stage of the datapath

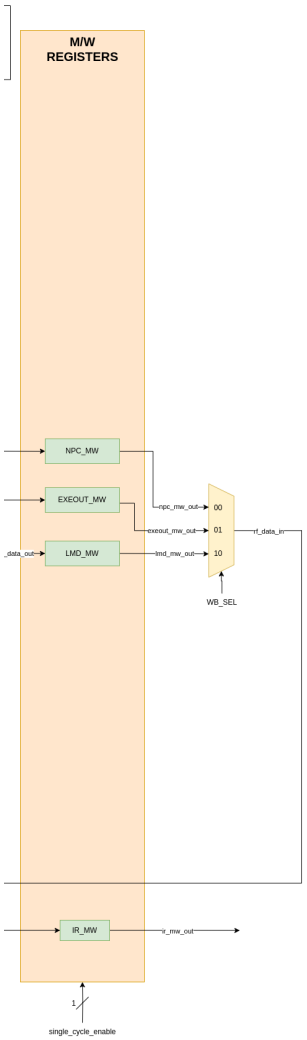

Figure 2.6: Write back stage of the datapath

The register file is configured to perform the writing operation on the falling edge of the clock (technically it writes on the rising edge of the negated clock, since the tech library used doesn't include negative-edge triggered flip-flops). This decision might decrease the overall performance from a delay point of view (by increasing the critical path) but it allows the processor to write and read data from the same destination in the same clock cycle, thus reducing the latency of data-dependent instructions by one clock cycle. In this way, each time there is a RAW dependency between two instructions, there must be only two instructions to separate them, not three.

Since in a `JAL` instruction the destination is implicit (`r31`), the register file has a `RF_WRITE31` signal which forces the write destination to the thirty-second register if it is one.

2.2 Controller

The control unit of the processor communicates directly with the datapath unit and generates the necessary commands to guarantee its correct behaviour and the execution of its nominal functions.

A hybrid FSM-hardwired controller has been used in our processor, as it is the most convenient choice in the presence of a pipelined datapath with support for multi-cycle operations. The manage-

ment of the pipeline is made possible by a series of registers, so that every stage receives the correct control word at each clock cycle.

The controller receives as data inputs the opcode coming from the instruction memory and some status signals from the high-latency components (divider and multiplier), while it outputs the control word associated with the received opcode and the enable signals of some registers. A look-up table is used to store all the control words and is addressed using the instructions' opcodes. The controller's FSM has two possible states (or modes): *single-cycle* and *multi-cycle*.

In single-cycle mode the control word is generated combinationally and according to the opcode received from the IRAM, so the controller behaves like a hardwired control unit. If the opcode of a multiplication or a division is recognised, then the controller will stop the instruction fetching after one clock cycle (by disabling the program counter) and will wait another clock cycle before switching to multi-cycle mode: during this time period it doesn't stop the control word generation. This latency before changing state is necessary to guarantee that the controller is in multi-cycle mode when the high-latency operation is in the execution stage.

In the multi-cycle state the controller stops generating the control words and disables the control word registers and the registers between the stages, effectively halting the pipeline: this is done to allow the multiplication/division components to complete their operations. Once this happens, the high-latency component in use will raise an "operation complete" signal: once that happens the controller will go back to single-cycle mode after one clock cycle, thus allowing the pipeline to resume its operations.

It is important to know that the controller can manage only one multi-cycle instruction at a time: details on how different situations are given in subsection 3.1.3.

CHAPTER 3

Implementation

3.1 ISA

In this section we will briefly highlight the ISA of our DLX implementation. This consists in:

- the control word and the meaning of each bit;
- the implemented instructions and their behavior;
- the way dependencies must be handled in order for the CPU to work.

3.1.1 Control word structure

The control word, generated by the controller at each clock cycle, is made of 33 bits. Here is its description: the different signals are grouped based on the stage in which they are used.

- Decode stage
  - `rf_rd1, cw(32)`: enables the read port 1 of the register file.
  - `rf_rd2, cw(31)`: enables the read port 2 of the register file.
  - `se_signed_unsigned_bar, cw(30)`: when it is zero, the immediate is zero-extended to 32 bits (it's treated as an unsigned number), otherwise it's sign-extended.
  - `se_size_16_26_bar, cw(29)`: when it is zero, the immediate is `IR[25:0]` and will be always zero-extended, otherwise it is `IR[15:0]` and `se_signed_unsigned_bar` will be taken into account.
- Execution Stage
  - `ex_sel_a, cw(28)`: when it is zero, the first operand of the execution stage is the NPC, otherwise it is the output of port 1 of the register file.
  - `ex_sel_b, cw(27)`: when it is zero, the second operand of the execution stage is the extended immediate value, otherwise it is the output of port 2 of the register file.
  - `alu_sub_add_bar, cw(26)`: when it is zero the adder performs an addition, otherwise a subtraction.
  - `alu_logic_sel, cw(25 downto 22)`: defines which logic operation to perform (and, or, xor, nor, etc.).

  - `alu_shift_sel, cw(21 downto 20)`: defines which shift operation to perform (logical left shift, arithmetic right shift, etc.).
  - `mul_start, cw(19)`: when it is one, a multiplication will start after one clock cycle.
  - `div_start, cw(18)`: when it is one, a division will start after one clock cycle.
  - `div_signed_unsigned_bar, cw(17)`: when it is zero, the division operands are treated as unsigned, otherwise they are treated as signed.
  - `div_sel, cw(16)`: when it is zero, the division's remainder will be selected, else the quotient will be returned.
  - `mul_sel, cw(15)`: when it is zero, the 32 LSBs of the product will be selected, else the 32 MSBs will be returned.
  - `cmp_config, cw(14 downto 12)`: defines which compare operation to perform (grater than, grater than or equal, etc.).
  - `ex_sel_out, cw(11 downto 9)`: is used to select which result of which component will be the output of the execution stage.
  - `branch_eq_neq_bar, cw(8)`: when it is zero, the detector will check whether its input is equal to zero, else it will check if it is not equal to zero.

- Memory Stage
  - `mem_rw_wr_bar, cw(7)`: when it is zero, the memory is written to, else it is read from.
  - `mem_branch_enable, cw(6)`: when it is one a branch operation is being executed, so the PC is overwritten if the zero detector previously generated a one.
  - `mem_perform_jump, cw(5)`: when it is one a jump is being executed: the PC is always overwritten.

- Writeback Stage
  - `wb_sel, cw(4 downto 3)`: is used to select the contents of which register will be sent to the register file's write port.
  - `rf_sel_dest, cw(2)`: if zero, the destination address is read from `IR[20:16]`, else it's read from `IR[15:11]`.
  - `rf_write31, cw(1)`: if one, the write destination of the register file is automatically set to `r31` and the write address is ignored.
  - `rf_wr, cw(0)`: if one, enables the register file's write port.

3.1.2 Instruction set

The list of available instructions is in Table 3.1. With respect to the base version of the DLX, we added: nand, nandi, nor, nori, xnor, xnori, slt, slti, sgt, sgti, seq, seqi, smulh, smull, uquot, urem, squot, srem, ret.

We decided to avoid inserting the control words of each instruction in this table for brevity; nonetheless, the complete list of control words can be found in the spreadsheet inside the `doc` folder of the project.

Mnemonic	Opcode	Example	Operation
add	0x00	add rx, ry, rz	RF[x] = RF[y] + RF[z]
addi	0x01	add rx, ry, imm_16	RF[x] = RF[y] + imm_16
and	0x02	and rx, ry, rz	RF[x] = RF[y] & RF[z]
andi	0x03	andi rx, ry, imm_16	RF[x] = RF[y] & imm_16
nor	0x04	nor rx, ry, rz	RF[x] = !(RF[y] | RF[z])
nori	0x05	nori rx, ry, imm_16	RF[x] = not(RF[y] | imm_16)
nand	0x06	nand rx, ry, rz	RF[x] = !(RF[y] & RF[z])
nandi	0x07	nandi rx, ry, imm_16	RF[x] = not(RF[y] & imm_16)
xnor	0x08	xnor rx, ry, rz	RF[x] = !(RF[y] R̂F[z])
xnori	0x09	xnori rx, ry, imm_16	RF[x] = not(RF[y] îmm_16)
beqz	0x0a	beqz rx, imm_16	PC = (RF[x]) ? PC + 4 : (PC + 4 + imm_16)
bnez	0x0b	bnez rx, imm_16	PC = (RF[x]) ? (PC + 4 + imm_16) : PC + 4
j	0x0c	j imm_26	PC = PC + 4 + imm_26
jal	0x0d	jal imm_26	PC = PC + 4 + imm_26; RF[31] = PC + 4
lw	0x0e	lw ry, imm_16(rx)	ry = MEM[rx + imm_16]
nop	0x0f	nop	Nothing
or	0x10	or rx, ry, rz	RF[x] = RF[y] | RF[z]
ori	0x11	ori rx, ry, imm_16	RF[x] = RF[y] | imm_16
sge	0x12	sge rx, ry, rz	rx = (ry >= rz) ? 1 : 0
sgei	0x13	sgei rx, ry, imm_16	rx = (ry >= imm_16) ? 1 : 0
sle	0x14	sle rx, ry, rz	rx = (ry <= rz) ? 1 : 0
slei	0x15	slei rx, ry, imm_16	rx = (ry <= imm_16) ? 1 : 0
sne	0x16	sne rx, ry, rz	rx = (ry != rz) ? 1 : 0
snei	0x17	snei rx, ry, imm_16	rx = (ry != imm_16) ? 1 : 0
seq	0x18	seq rx, ry, rz	rx = (ry == rz) ? 1 : 0
seqi	0x19	seqi rx, ry, imm_16	rx = (ry == imm_16) ? 1 : 0
slt	0x1a	slt rx, ry, rz	rx = (ry < rz) ? 1 : 0
slt	0x1b	slti rx, ry, imm_16	rx = (ry < imm_16) ? 1 : 0
sgt	0x1c	sgt rx, ry, rz	rx = (ry > rz) ? 1 : 0
sgti	0x1d	sgti rx, ry, imm_16	rx = (ry > imm_16) ? 1 : 0
sll	0x1e	sll rx, ry, rz	rx = (ry << rz[4:0])
slli	0x1f	slli rx, ry, imm_16	rx = (ry << imm_16[4:0])
srl	0x20	srl rx, ry, rz	rx = (ry >> rz[4:0])
srli	0x21	srli rx, ry, imm_16	rx = (ry >> imm_16[4:0])
sub	0x22	sub rx, ry, rz	RF[x] = RF[y] - RF[z]
subi	0x23	subi rx, ry, imm_16	RF[x] = RF[y] - imm_16
sw	0x24	sw imm_16(rx), ry	MEM[rx + imm_16] = ry
xor	0x25	xor rx, ry, rz	RF[x] = RF[y] R̂F[z]
xori	0x26	xori rx, ry, imm_16	RF[x] = RF[y] îmm_16
smulh	0x27	smulh rx, ry, rz	RF[x] = (RF[x]*RF[y])[63:32]
smull	0x28	smull rx, ry, rz	RF[x] = (RF[x]*RF[y])[31:0]
uquot	0x29	uquot rx, ry, rz	RF[z] = RF[y] // RF[z]
urem	0x2a	urem rx, ry, rz	RF[z] = RF[y] % RF[z]
squot	0x2b	squot rx, ry, rz	RF[z] = RF[y] // RF[z]
srem	0x2c	srem rx, ry, rz	RF[z] = RF[y] % RF[z]
ret	0x2d	ret rx	PC = RF[x]

Table 3.1: Instruction set of our DLX

3.1.3 Dealing with dependencies

Here are some highlights on how to handle dependencies in the code. These rules must be followed in order for the CPU to work as expected.

- If a RAW hazard happens, there must be two instructions between the one which writes and the one which reads. In case this does not already happen in the code, some `nop` instructions are necessary.
- For instructions which modify the program counter, the number of required `nop` instructions is three, since the program counter is written during the *memory* stage.
- Two multi-cycle instructions must be separated by other two instructions. `nop` instructions are required if this is not the case in the program. The reason behind this is that the controller can only handle multi-cycle instruction at a time and will only consider the most recently fetched one.

3.2 Components Overview

To obtain a full working DLX processor we developed and used different components. They are:

- IRAM;
- DRAM;
- Controller (implementation explained in section 2.2);
- Zero detector;
- Comparator;
- Pipeline register (single- and multi-bit);
- Register file;
- Sign extender;
- Logic unit;
- Shifter;
- P4 Adder;
- Booth multiplier (implementation explained in Appendix C);
- SRT Radix-4 divider (implementation explained in Appendix B);

3.2.1 IRAM

The IRAM is the static, read-only instruction memory used from which the instructions to be executed are fetched. Its content is initialised by a process which reads from the `sim/IRAM_init_file.mem` file when reset signal is active. During nominal operation it sends out its data on the `mem_instr_out` signal: the datapath receives the full instruction while the controller only receives the opcode (located in the 6 MSBs of the instructions).

3.2.2 DRAM

The DRAM is the data memory of the processor. It is characterised by two different processes: one that is in charge of asynchronously reading, while the other is in charge of synchronously writing it on the positive edge of the clock or resetting it. The signal `rw_bar` discriminates between a read (the signal is one) and a write (the signal is zero) operation.

3.2.3 Zero detector

The zero detector is used to compute whether a branch should be taken or not. Since there are the `beqz` and `bnez` instructions, we use the signal `branch_eq_neq_bar` to discriminate between the two:

- `branch_eq_neq_bar` is zero: `bnez` instruction, so the output is one if the data input is not zero;
- `branch_eq_neq_bar` is one: `beqz` instruction, so output is one if the data input is zero;

3.2.4 Comparator

Exploiting the already implemented adder/subtractor, we added some logic to obtain a comparator. It takes as inputs the calculated difference and carry out of the adder and analyses the relation between the two operands A and B, depending on `config` input (which is responsible of differentiating between the possible types of comparison). In particular:

- `config` is `000`: less than equal comparison, returns one if $A \leq B$;
- `config` is `001`: less than comparison, returns one if $A < B$;
- `config` is `010`: greater than comparison, returns one if $A > B$;
- `config` is `011`: greater than equal comparison, returns one if $A \geq B$;
- `config` is `100`: equal comparison, returns one if $A = B$;
- `config` is `101`: not equal comparison, returns one if $A \neq B$.

The result of the comparison is then zero-extended to 32 bits as it is handled like any other data by the datapath.

3.2.5 Pipeline register

This component is a simple Parallel-In-Parallel-Out N-bit or 1-bit register with an enable signal which is driven by the controller directly and is used to stall the pipeline in case of multi cycle operations. These registers have been implemented as standalone components for the sake of cleaner, more reusable code.

3.2.6 Register file

The implemented register file is composed by 32 registers which are N bits wide, with two reading ports and one writing port. The enable signals are:

- `RD1`: enables reading from reading port 1;
- `RD2`: enables reading from reading port 2;
- `WR`: enables writing;

Function	S_3	S_2	S_1	S_0
and	1	0	0	0
nand	0	1	1	1
or	1	1	1	0
nor	0	0	0	1
xor	0	1	1	0
xnor	1	0	0	1

Table 3.2: Functions computed by the logic unit based on the value of `S`

- `Write31`: when it is one the write destination of the register file is automatically set to r31 and the write address is ignored (it's used used by `jal` instructions).

Reading is asynchronous, while writing is synchronous on the falling edge of the clock to decrease the latency, as explained previously.

3.2.7 Sign extender

Sign-extension or zero-extension has to be performed both for I-type and J-type instructions, depending on the specific operation to be performed. To control the behaviour of the module two control signals have been defined:

- `size_16_26_bar`: when it is at one the immediate is considered on 16 bits, otherwise is on 26 bits;
- `signed_unsigned_bar`: when it is one the immediate is sign-extended, otherwise is zero-extended.

In the case of 26-bit immediate (J-type instructions), this is always extended as signed.

3.2.8 Logic unit

The implemented logic unit is based on the T2 logic unit, in which there is a selection signal `S` on 4 bits that chooses one among different logic operations. This logic unit only computes the following function:

$$Output = S_3 \cdot A \cdot B + S_2 \cdot A \cdot \overline{B} + S_1 \cdot \overline{A} \cdot B + S_0 \cdot \overline{A} \cdot \overline{B}$$

The values of `S` associated with each logic function are summarised in Table 3.2.

3.2.9 Shifter

The unit implements the T2 barrel shifter on N bits, where N is a power of 2. It takes as inputs the value to shift, the amount of positions to shift it by (so only the six LSBs of the second operand are considered) and a 2-bit configuration signal to choose between logic/arithmetic and left/right shifts. The shift is performed by three successive operations: first a set of eight 71-bit masks is generated, then one of these masks is chosen, then only a 64-bit subsection of these masks is selected as the final output.

3.2.10 P4 adder

The adder shares the architecture of the one used in the Pentium 4 processor: using a sparse tree carry generator, only one of them is generated each four bits. The carries are used by eight carry-select adders, in charge of computing 4 bits of the sum each.

The adder produces the result in one clock cycle. In particular, it can be used either as an adder or as a subtractor, depending on the value of the `ALU_SUB_ADD_BAR` signal. This signal, although it's conceptually used for control from an external point of view, internally it's used as a data signal. Due to the way two's complement works:

$$|A - B|_2 = |A + \overline{B} + 1|_2$$

For this reason, the signal `ALU_SUB_ADD_BAR` is used as carry-in and also to select the second operand between B and $\overline{B}$.

3.3 Simulation

In this section we will talk about the way you can simulate the processor, using the scripts we have written.

The simulation consists in providing the CPU some stimuli (i.e. an assembly program) and observing how the system reacts to them. In particular, it might be interesting to look at the internal signals, in order to understand whether the components and the pipeline are working correctly, and to see the state of the register file and of the memory at the end of the simulation.

The simulation is performed using the ModelSim simulation software; the scripts we provide have been tested on version 20.1 and 20.4.

In order to simulate, you need an assembly program. This should be created inside the `sim/asm_example` folder, using the same name both for the folder and for the `.asm` file.

The simulation can be run by executing `scripts/simulate.py`. You can pass many arguments to this script or none at all, in order to make the execution more flexible. It is in charge of calling other scripts and commands: at the end of each of them, it is possible to see the outcome of the process on the terminal. In particular, green text stands for success, while red stands for failure.

Here is a brief explanation of each argument of the script:

- `-c, --compile_only`: compile the VHDL files without executing them;
- `-a ASM, --asm ASM`: which assembler file to execute;
- `-g, --gui`: open ModelSim's GUI to analyse the produced waveforms;
- `-p, --pretty`: prettify memory and register file dump;
- `-v [KEY=VALUE ...], --verify [KEY=VALUE ...]`: define a list of key-expected_value elements to be checked against the register file/dram dump's contents, executed only if `-p` is present.

With the flag `-c` the script just compiles the DLX. The list of files to be compiled is located in the `sim` folder. This can be useful in order to know if the code written does not contain any syntax errors.

With the command `-asm [FILE]` you can assemble a `.asm` file and use it to simulate the processor. The script stops in case the assembler produces some errors.

By default, the simulation does not open the gui of modelsim. All the commands needed for that are provided by `sim/simulation_cli.do`. Basically the simulation is run for 1 µs, then it dumps the contents of register file and dram into the file `sim/dump_rf_dmem.dump`. As a side note, it is highly suggested to add an endless loop at the end of you program to avoid unexpected behaviours after its execution is completed.

The content of the dump file as-is is not very readable, since it is only the list of all the values inside the two components. Using the flag `-p`, another script is run which prettifies the dump. Figure 3.1 shows an example of the register file part of a dump after it has been prettified.

```
REGISTER FILE DUMP:
reg     hex             int
-----------------------------------
r0:     0x00000000      0
r1:     0x00000001      1
r2:     0x00000001      1
r3:     0x00000000      0
r4:     0x00000000      0
r5:     0x00000000      0
r6:     0x00000000      0
r7:     0x00058980      362880
r8:     0x0000000a      10
r9:     0x00000000      0
r10:    0x00375f00      3628800
```

Figure 3.1: Example of dump file (only the first few lines)

```
Register r1 is not 0x001abde7;
->      found 0x00000001 instead
Memory location m0 is 3628800 as expected
```

Figure 3.2: Result of a simulation using the `-v` flag

The last option of the script is the auto-verification of the values stored in the register file and/or dram. Using the `-v` option, you can provide a list of key-value pairs which are expected to be in the dump. Each element of this list is separated by the others using a space. A key can either be a register (using the structure `rX`, with `X` between 0 and 31) or a memory location (using the structure `mX`, with `X` between 0 and 255). The expected value can be written both in hexadecimal (in the form `0xXXXXXXXX`, has to be exactly 8 digits long) or as an unsigned decimal.

Let's say that you want to check whether, at the end of the simulation, the register 1 has value `0x001abde7` while the dram cell 0 has value 3628800. To do so, you would execute `scripts/simulate.py -a your_file.asm -p -v r1=0x001abde7 m0=3628800`.

At the end of the execution, the script will show you whether the checks were successful or not. In Figure 3.2 you can find an example of the output should a check fail.

If you also want to have a look at the generated waveforms, in addition to all of the CLI options provided so far, you can add the `-gui` option when executing the script. After successful compilation, ModelSim will open and the system will ready to be run, using the `run time X` command in software's terminal.

The script automatically adds to the waveform viewer all the important signals of the CPU (basically everything related to the pipeline); the signals are grouped by stage. Different colours are used as well, one per stage, in order to improve readability.

3.4 Synthesis

The synthesis process has been done using Synopys Design Compiler. The process has been done in a simple way, without considering other possible optimisations.

The first step is to analyze the files, in order to see whether they can be synthesised or not. The first time we did so, we discovered that most of our DLX was fine except for the register file: the writing operation happens on the negative edge of the clock, but the technology library we employed does not have a negative-edge triggered flip flop, so there was no way to synthesise it. In order to keep the component's behaviour unchanged, the clock inside the register file is inverted and its rising

Power Group	Internal Power	Switching Power	Leakage Power	Total Power
IO pad	0	0	0	0
Memory	0	0	0	0
Black Box	0	0	0	0
Clock Network	172.7068	$1.7301 \cdot 10^5$	28.7064	$1.7318 \cdot 10^5$
Register	$1.7912 \cdot 10^3$	8.1986	$1.4242 \cdot 10^5$	$1.9418 \cdot 10^3$
Sequential	$1.7210 \cdot 10^{-2}$	$7.2383 \cdot 10^{-4}$	$3.9772 \cdot 10^3$	3.9952
Combinational	44.4411	115.3323	$2.4928 \cdot 10^5$	409.0535
Total	$2.0083 \cdot 10^3$	$1.7313 \cdot 10^5$	$3.9570 \cdot 10^5$	$1.7554 \cdot 10^5$

Table 3.3: Post synthesis power consumption in µW

edge is used to perform a write operation.

The synthesis script located in `syn/syn_script.tcl` shows all the commands we used to perform the synthesis.

We decided not to spend much time at this stage (for instance, we could have uses the switching activity of the system to improve the power analysis, or to adopt the clock gating technique to decrease power consumption).

The final reports show a very slow critical path, around 6.5 ns, which is the carry propagation through the ripple carry adder used inside the multiplier. Because of this, the synthesis has been performed with a global timing constraint of 7 ns, which means that our DLX would run at ~ 142.8 MHz. A possible optimisation may consists of using a 64-bit P4 adder inside the multiplier, in order to improve the critical path. Another possibility would be to replace the 64-bit multiplier with a 32-bit one: this would reduce the functionalities of the CPU, but improve the delay. Another possible optimisation would be to pipeline the multiplier by splitting the 64-bit sum in two subsequent 32-bit sums that would happen in two different clock cycles (this would double the latency of the multiplication).

The results about the estimated power consumption are shown in Table 3.3. The overall area occupation is 19616 μm^2.

3.5 Physical Design

The placing and routing of the DLX processor has been done through the aid of Cadence Innovus Implementation System.

First we created the configuration files for Innovus, such as the `.globals` and `.view` files, so that they pointed to the post-synthesis netlist and SDC files. Then we proceeded to the structuring of the floorplan, defining the aspect ratio of the die and inserting the power rings using high metal layers to avoid congestion (M9 for horizontal lines and M10 for vertical lines).

In order to distribute the power and ground signals across the chip to the cells, vertical and horizontal metal wires were added and routed. The metal layers 1 to 8 were allocated to the cells, avoiding the higher levels which are reserved to for the power grid.

As it was requested to place the memories outside the processor, we needed to place their connection pins along the left and right edges of the chip. Along with these pins we also added the clock and reset ones at the top of the die .

For structural reasons we completed the placement by inserting filler cells in the free spaces, in order not to leave any empty space. Finally we performed the signal routing.

Analyses for timing have been done in order to understand if any violation might have occurred and we verified that we obtained positive slack both for setup and hold times.

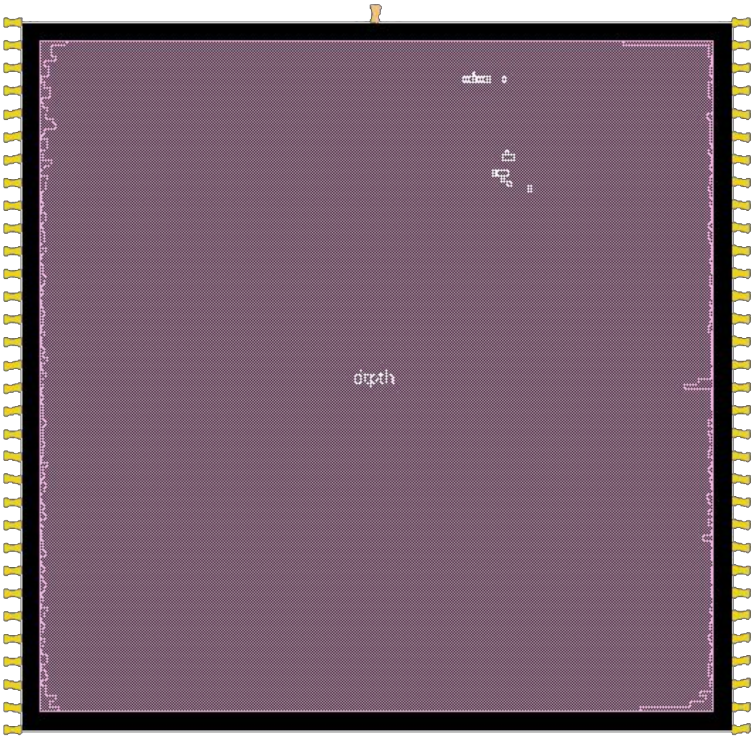

Figure 3.3: Amoeba view of the DLX

The reports generated state that our DLX occupies a total area of 18798.5 μm^2, is made of 23557 gates and 10486 cells.

In Figure 3.3 and 3.4 we can see the amoeba view and the physical view of our processor. It is interesting to notice that the datapath occupies almost the entire die area, while the area occupation of the controller (whose components are the white rectangles in the top-right quarter of the amoeba view) is minimal.

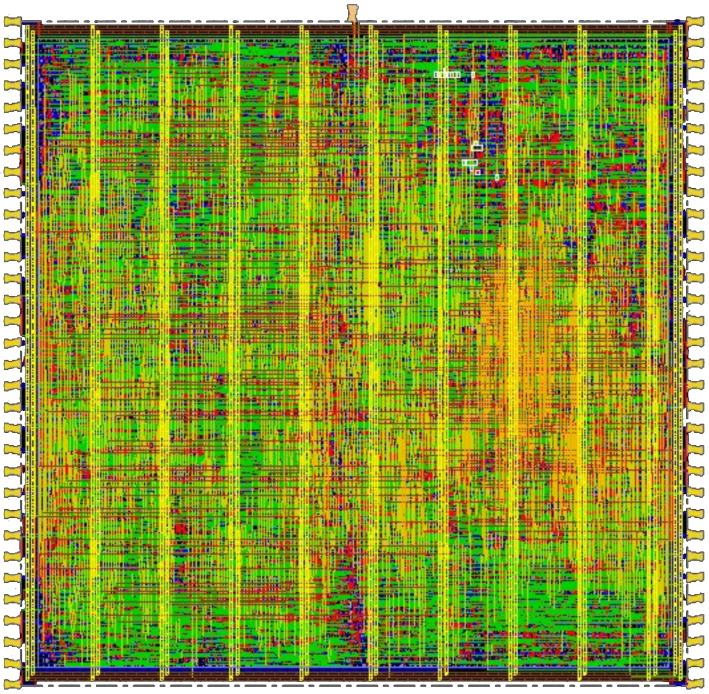

Figure 3.4: Physical view of the DLX

CHAPTER 4

Conclusions

Several possible improvements can be made to our microprocessor, as we did not have the chance to fully develop all the feasible features of a RISC processor.

Pipeline optimisation could be a starting point: we did not design units to handle data and control hazards as we handled the issue via software. However, this is inefficient and proper hardware components should be added: they include a forwarding unit, a stall controller, and a pipeline flush controller. They would greatly help to improve the overall performance and usability of the device. Additional upgrades would include adding a Branch History Table or a Branch Target Buffer to perform branch prediction and reduce branch/jump penalties.

Another desirable feature would be floating point execution units, as currently only integer arithmetic is supported.

Optimisation of the critical path is a very important point, since at the moment it's 6.5 ns due to the 64-bit ripple carry adder inside the multiplier. Possible solutions include:

- using a 64-bit P4 adder;
- implementing a 32-bit multiplier instead of a 64-bit one;
- pipelining the multiplier, increasing its throughput.

One more improvement can be implementing power optimisation techniques, such as clock gating, to reduce the power consumption.

In conclusion, the project has been a challenging yet rewarding journey for the team. The effort has been remarkable, as we delved into the intricacies of microprocessor architecture and digital design. Through this project, we acquired skills in complex functionality integration and the implementation of advanced algorithms like the SRT division. We have refined our abilities in problem-solving, critical thinking, and collaboration, navigating through a high-complexity design and overcoming obstacles. This project not only provided a platform for applying theoretical knowledge but also fostered a deeper understanding of the intricacies involved in developing complex features. The experience gained throughout this project sets a foundation for further exploration in the field of microprocessor design and development.

APPENDIX A

On the SRT Algorithm

The SRT algorithm was independently developed around 1958 by three researchers: Sweeney (IBM), Robertson (University of Illinois) and Tocher (Imperial College London). The first letter of their surnames form the name of the algorithm. In the area of slow division algorithms, which are able to produce k exact digits of the quotient per clock cycle, it is currently the most used.

It became famous in 1990 because of a bug in the floating point division unit of the Pentium 486DX processor . At the end of this appendix, the reader will have the instruments needed to understand what was wrong with their implementation. Before that, we need to start understanding how is it possible to divide binary numbers, in order to gradually introduce the ideas behind SRT.

The following material comes from [3] and [1].

A.1 Simple division algorithms

In division, given a dividend D on m bits and a divisor d on n bits, we want to find two unsigned binary numbers Q and R, with $R < d$, such that

$$D = Q \cdot d + R$$

As it is widely known, before diving into a division we need to check that $d \neq 0$, otherwise the above expression would make no sense. In the context of binary numbers, we also need to check that the final quotient can be expressed on n bits (same size of the divisor). This condition is not always satisfied, so we need more constraints.

We want $Q < 2^n$ and, since it still has to be true that $R < d$, we obtain the following inequality:

$$D = Q \cdot d + R \leq (2^n - 1) \cdot d + d = 2^n \cdot d$$

This condition has to be checked before the division, otherwise the quotient will overflow and its representation will be wrong.

A.1.1 Restoring algorithm

The simplest algorithm we can think of is the *restoring algorithm* as it is identical to the one we learnt during elementary school. We compute one digit q_i of the quotient Q at a time (most significant bit first) by looking at the most significant digits of the current remainder. In the binary case, $q_i \in \{0, 1\}$, so what we need to check is whether d is greater or less than the $N/2$ most significant digits of the current remainder A.

In hardware we cannot guess as we do on paper, so we try to perform a subtraction between A and $2^{N/2} \cdot d$ (d is left-shifted by $2^{N/2}$ to subtract it from A's most significant digits, due to A being twice as long as d): if the result is positive, then the correct digit of the quotient q_i is 1, otherwise it is 0.

The following code block is an implementation of the above algorithm in C++, using a library defined specifically to simulate the behaviour of VHDL's `std_logic_vector` type. It will be used later on for the same purpose.

```
std::pair<SLV::std_logic_vector, SLV::std_logic_vector>
Division::restoring_division
(SLV::std_logic_vector& dividend, SLV::std_logic_vector&  divisor){

    int N = divisor.size;

    if(dividend.to_base_10() >= (divisor.to_base_10() << N/2))
        throw std::invalid_argument();

    SLV::std_logic_vector R(N/2);
    SLV::std_logic_vector Q(N/2 + 1);
    SLV::std_logic_vector A(N + 1);
    SLV::std_logic_vector B(N + 1);

    A = SLV::std_logic_vector(1) &
        dividend;

    B = SLV::std_logic_vector(1) &
        divisor.extract(N/2 - 1, 0)  &
        SLV::std_logic_vector(N);

    for(int i = N/2; i >= 0; i--){
        A = A - B;
        if(A.get_msb() == 0)
            Q.set(i),
            A = A << 1;
        else
            Q.reset(i),
            A = (A + B) << 1;
    }

    R = A.extract(N, N/2 + 1);
    Q = Q.extract(N/2 - 1, 0);

    return std::pair<SLV::std_logic_vector, SLV::std_logic_vector>
    (Q, R);
}
```

It is clear from the code that, in order to decide which is the current digit of the quotient q_i and calculate the current remainder, we first need to perform $A - B$: since in an hardware implementation this difference is stored in a register that we first write to and then read from, to compute one digit of the quotient we need 2 clock cycles.

A.1.2 Non-restoring algorithm

The *non-restoring algorithm* solves this problem: as we can see from Figure A.1, depending on the current value of the shifted remainder we either subtract or add $2^{N/2} \cdot d$ and choose the digit of the quotient based on the sign of the current remainder A.

Here is an implementation of the algorithm:

```
std::pair<SLV::std_logic_vector, SLV::std_logic_vector>
Division::non_restoring_division
```

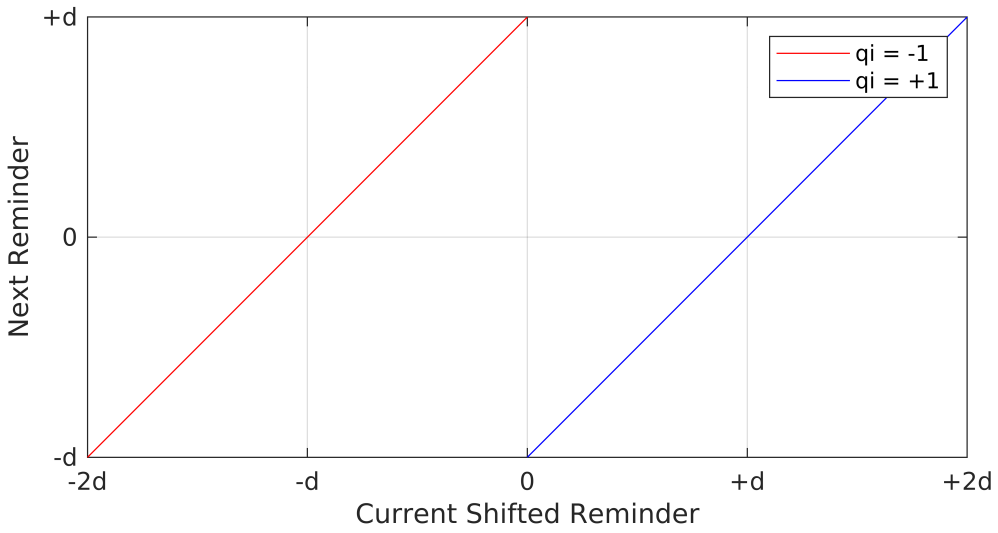

Figure A.1: Non restoring division: operation to perform at each cycle

```
(SLV::std_logic_vector& dividend, SLV::std_logic_vector & divisor){

    int N = divisor.size;

    if(dividend.to_base_10() >= (divisor.to_base_10() << N/2))
        throw std::invalid_argument();

    SLV::std_logic_vector Q(N/2 + 1);
    SLV::std_logic_vector R(N/2);
    SLV::std_logic_vector A(N + 2);
    SLV::std_logic_vector B(N + 2);

    A = SLV::std_logic_vector(2) &
        dividend;

    B = SLV::std_logic_vector(2)  &
        divisor.extract(N/2 - 1, 0) &
        SLV::std_logic_vector(N/2);

    for(int i = N/2; i >= 0; i--){
        if(A.get_msb() == 0)
            Q.set(i),
            A = (A - B) << 1;
        else
            Q.reset(i),
            A = (A + B) << 1;
    }

    Q = Q - Q.complement();

    if(A.get_msb() == 1)
        Q = Q - 1,
        A = ((A >> 1) + B) << 1;

    R = A.extract(2 * N/2, N/2 + 1);
    Q = Q.extract(N/2 - 1, 0);

    return std::pair<SLV::std_logic_vector, SLV::std_logic_vector>
    (Q, R);
}
```

In this algorithm, we can see something which will be very useful in the SRT division: a non-standard encoding of the digits of the quotient. Instead of using $\{0, 1\}$, at each clock cycle we choose q_i in $\{-1, 1\}$. The value of a number that uses this special digit set can be computed as $\sum a_i \cdot 2^i$ where $a_i \in \{-1, 1\}$.

At the end of the algorithm, we need to re-code the quotient to the more standard $\{0, 1\}$ digit set. This is not complicated, depending on the way we implement the $\{-1, 1\}$ encoding. For instance, we can represent -1 as a 0 in the quotient register, and then compute $Q = Q - \bar{Q}$. This is what happens in the example above. Another possibility is to use two registers, Q and C: we set Q_i to 1 when the final quotient's digit q_i is 1, else we set C_i, and then compute $Q = Q - C$. Later on we will see that this computation can be also done *on the fly*.

Another issue is that, by accepting to have remainders which are not correct, we may end up with a negative remainder once all of the quotient's digits have been calculated. In this case a further correction is required: we decrease Q by one and add the shifted divisor to the remainder. Again, this final correction is shown in the code.

The loop now requires $N/2$ or $N/2 + 1$ clock cycles (depends whether the quotient and remainder need to be corrected at the end), since we can set one digit of the quotient and compute the remainder at the same time.

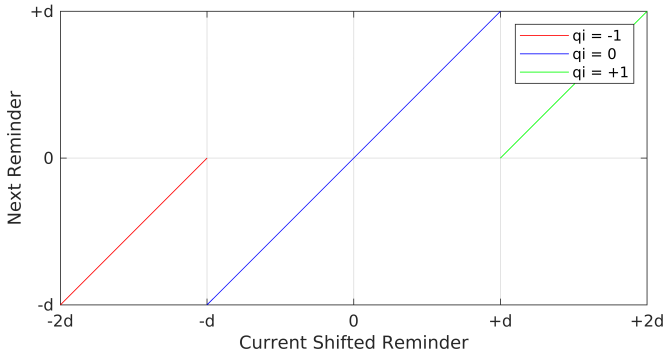

Figure A.2: SRT: operation to perform at each cycle without normalisation

A.2 The SRT division

It is difficult to use one of the previously analysed approaches to design a division process with a radix higher than 2: an algorithm in radix 2^k is able to compute k digits of the final result at each iteration, so the higher the radix the faster the result is generated.

The SRT algorithm tries to solve this problem by requiring $d \in \left[\frac{1}{2}, 1\right)$ before the division can be performed: this is particularly simple in binary representation because it means that d's most significant bit has to be 1. It should be noted that, should a normalisation take place, the dividend D will have to be normalised too. It may be useful to recall that, given a fixed-point fractional binary number $0.a_{-1}a_{-2}...a_{-n}$ we can compute its value in base ten using the expression

$$A = \sum_{i=1}^{n} a_{-i} \cdot 2^{-i}$$

For example, the decimal conversion of 0.101 is $1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8} = \frac{5}{8}$. The integer part, if different from 0, can be calculated using the standard binary-to-decimal conversion method. This constraint on the operands is particularly convenient for IEEE-754 floating point numbers, since their mantissa is already represented as $1.xxxx...$ (so it just needs to be shifted one position to the right to be normalised). However, the algorithm (and its implementation) can be used for signed/unsigned integer numbers as well, provided that we normalise them and then adjust the result of the division.

A.2.1 Radix-2 algorithm

Knowledge of the radix-2 version is preliminary for the radix-4, as the basic principles are the same for both.

We want to work with the $\{-1, 0, 1\}$ digit set. Figure A.2 shows how to compute the next remainder and quotient digit q_i starting from the current remainder and using the above digit set: this graph is not very useful, as it requires comparing the value of the current remainder both with d and $-d$ before knowing what to do, which is way more complex than checking the remainder's most significant bit.

This is where the normalisation of the divisor proves useful: in this way the remainder is forced in the $[-1, 1]$ interval. We can now use a new a p-d plot as a graphical tool for understanding the

Table A.1: Possible look up table for radix-2 SRT

A_0	A_{-1}	A_{-2}	**Value**	**Interval**	q_i
0	0	0	00.0xxx	$0 < A < \frac{1}{2}$	0
0	0	1	00.1xxx	$\frac{1}{2} \leq A < 1$	1
0	1	0	01.0xxx	$1 \leq A < \frac{3}{2}$	1
0	1	1	01.1xxx	$\frac{3}{2} \leq A < 2$	1
1	0	0	10.0xxx	$-2 < A \leq -\frac{3}{2}$	-1
1	0	1	10.1xxx	$-\frac{3}{2} < A \leq -1$	-1
1	1	0	11.0xxx	$-1 < A \leq -\frac{1}{2}$	-1
1	1	1	11.1xxx	$-\frac{1}{2} < A < 0$	0

quotient digit selection process.

In Figure A.3, the horizontal axis shows the normalised divisor's values, the vertical axis shows the current remainder's values, the vertical segments on the right represent the quotient digit q_i associated with the given remainder range and the red areas represents invalid (A, d) couples. We represent said remainder in 2's complement, so the first digit is the sign.

As an example, q_i can be 0 only if the remainder is between the two green lines ($-d \leq A \leq d$): if we were not follow this rule, then when we sum d to or subtract it from A we would end up with a remainder which is higher than 1 or lower then -1 (without normalisation, this would be equivalent to saying that $R \geq d$ or $R \leq -d$). In the next subsection we will briefly cover how to build a plot like this given a radix and a digit set for the quotient.

Each point (A, d) on the plot might be associated with more than one possible q_i digit: for instance, having $(0.111, 00.11)$ leads to q_i being either to 1 or 0. For this reason we need to set some boundaries in order to have a single possible q_i digit given a couple of A and d.

To do so, we can draw some diagonal lines: they partition the diagram into areas, and to each of those a unique q_i digit is associated. This also makes the decision about the q_i digit be dependent exclusively on A's value, without having to take d into consideration. Here are three possible partitioning schemes:

$$q_i = \begin{cases} 1 & \text{if } A \geq 0 \\ 0 & \text{if } 0 > A \geq -\frac{1}{2} \\ -1 & \text{if } A < -\frac{1}{2} \end{cases}$$

$$q_i = \begin{cases} 1 & \text{if } A \geq 0 \\ -1 & \text{if } A < 0 \end{cases}$$

$$q_i = \begin{cases} 1 & \text{if } A \geq \frac{1}{2} \\ 0 & \text{if } \frac{1}{2} > A \geq -\frac{1}{2} \\ -1 & \text{if } A < -\frac{1}{2} \end{cases}$$

Since we are comparing A with fixed values, we can just use the first three digits of A to determine into which range does A belong and which q_i to choose. This is extremely simple to implement in hardware using a look-up table, which associates a digit q_i to each combination of the first three digits A. In Table A.1 we can see an example of the look-up table, valid for the last scheme of the previous list. Graphically, the same values can be observed from Figure A.4

Once q_i has been chosen, the algorithm computes the next remainder as $A_{i+1} = 2 \cdot (A_i - q_i \cdot d \cdot 2^{N/2})$. Regarding the representation of the quotient, there are many possible ways to store it. It can be stored

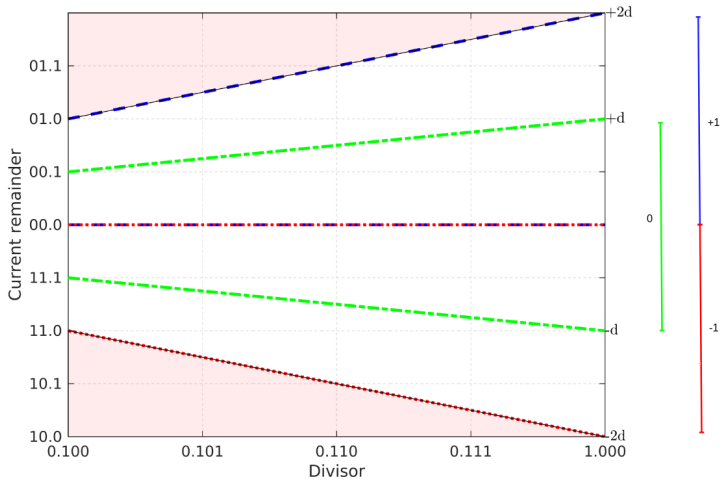

Figure A.3: p-d plot for SRT radix-2

in the standard binary representation by computing the translation from the $\{-1, 0, 1\}$ digit set *on the fly*, without the need to go back to a $\{0, 1\}$ digit set at the end of the algorithm. This *on the fly* conversion is performed at each step by computing $Q = 2 \cdot Q + q_i$. From an implementation point of view, this is means having the possibility to sum or subtract one to the shifted current quotient.

Before introducing the code, we need to cover one last element: the normalisation of the input values. In order to have the divisor in the range $[\frac{1}{2}, 1)$, we need to remove all the leading zeros, which means *shifting left as many times as the number of leading zeroes*. From an hardware point of view, this operation can be performed with a barrel shifter and a combinational circuit to detect the number of leading zeros k. Once we have shifted the divisor, we apply the same left shift to the dividend: after this normalisation the final quotient will be correct but to get the correct remainder we will need to shift it to the right by k positions (the quotient is unsigned so we would push zeroes as most significant bits).

Here is a the code to execute this algorithm. The function `srt_division_radix_ 2_lut` implements Table A.1.

```
std::pair<SLV::std_logic_vector, SLV::std_logic_vector>
Division::srt_division_radix_2
(SLV::std_logic_vector& dividend, SLV::std_logic_vector & divisor){

    int N = divisor.size;

    if(dividend.to_base_10() >= (divisor.to_base_10() << N/2)
        throw std::invalid_argument();

    SLV::std_logic_vector A (N   + 2);
    SLV::std_logic_vector B (N/2 + 2);
    SLV::std_logic_vector Q (N/2 + 2);

    unsigned int k =
        divisor.extract(N/2 - 1, 0).count_leading_zeros();

    A = SLV::std_logic_vector("00") & (dividend << k);
    B = SLV::std_logic_vector("00") & (divisor.extract(N/2 - 1, 0) << k);

    for(int i = 0; i <= N/2; i++){
        SLV::std_logic_vector A_upper  =
            A.extract(N + 1, N/2);
```

```
        SLV::std_logic_vector A_bottom =
            A.extract(N/2 - 1, 0);
        SLV::std_logic_vector A_top    =
            A.extract(N + 1, N - 1);

        SLV::std_logic_vector partial_R(N/2 + 2);

        int current_q =
            srt_division_radix_2_lut(A_top);

        if(current_q == 0)
            Q =  (Q << 1),
            partial_R = A_upper;

        if(current_q == 1)
            Q = ((Q << 1) + 1),
            partial_R = A_upper - B;

        if(current_q == -1)
            Q = ((Q << 1) - 1),
            partial_R = A_upper + B;

        A = (partial_R & A_bottom) << 1;
    }

    if(A.get_msb()){
        Q = Q - 1;
        A = A >> 1;
        A = (A.extract(N + 1, N/2) + B) & A.extract(N/2 - 1, 0);
        A = A << 1;
    }

    SLV::std_logic_vector R = A.extract(N + 1, N/2 + 1) >> k;
    Q = Q.extract(N/2 - 1, 0);

    return std::pair<SLV::std_logic_vector, SLV::std_logic_vector>
    (Q, R);
}
```

As it was for the non restoring algorithm, after the iterations are completed we may end up with a negative remainder: in this case we decrease Q by 1 and correct A.

A.2.2 Radix-4 algorithm

Once that we have understood how does the radix-2 algorithm works, it is easy to apply the same concepts to higher radixes. The radix-4 implementation is able to compute 2 digits of the quotient ad a time: in order to do so, the digit set of the quotient can either be $\{-3, -2, -1, 0, +1, +2, +3\}$ (called *maximally redundant digit set*) or $\{-2, -1, 0, +1, +2\}$ (called *minimally redundant digit set*). None of them is unconditionally better than the other: the *maximally redundant* set has a small look-up table with $2^5 = 32$ entries, but the required hardware to perform the computations is more complex; the *minimally redundant* set is simpler to implement, but the look-up table has $2^9 = 512$ entries.

Our final implementation of the circuit uses the *minimally redundant* digit set: at each step of the loop, once we have chosen the q_i digit, what we need to do is compute $Q_{i+1} = 4 \cdot Q_i + q_i$ and $A_{i+1} = 4 \cdot (A_i - q_i \cdot d \cdot 2^n)$.

The main problem is coming up with the look up table, but again the p-d plot can be used for that purpose. First of all we need to define it, then we need to set the boundaries to use for the digit selection and then we use all this information to extract a look up table.

We want the partial remainder for a given iteration i to be in $[-hd, hd]$ (with h being a scaling

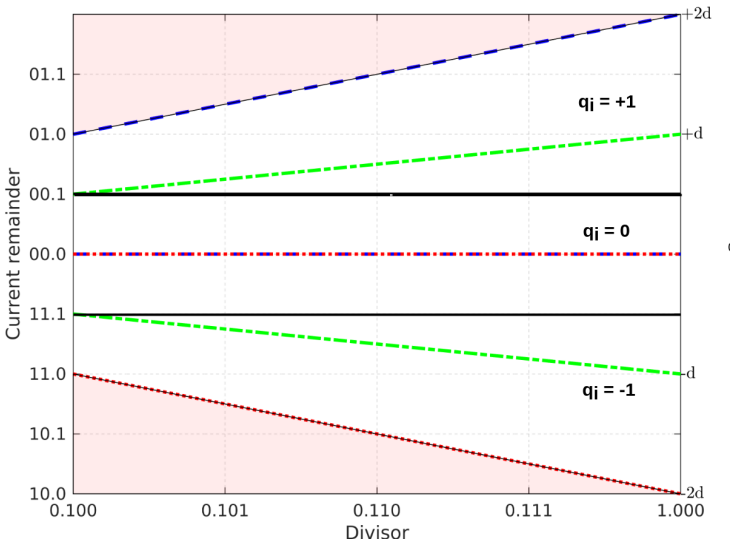

Figure A.4: p-d plot for radix 2 showing the boundaries to select the digits

factor), while the remainder produced by the previous iteration is in $[-4hd, 4hd]$ due to the fact that it was shifted left by two positions. Since at most we can add or subtract $2d$ to that remainder, we need to satisfy:

$$4hd - 2d \geq hd \Rightarrow h \leq \frac{2}{3}$$

If we choose $h = \frac{2}{3}$, we can compute the limits of each digit selection interval the following way:

$$\begin{aligned}
2_{upper} &= +\frac{2}{3} \cdot d + 2 \cdot d = +\frac{8}{3} \cdot d \\
2_{lower} &= -\frac{2}{3} \cdot d + 2 \cdot d = +\frac{4}{3} \cdot d \\
1_{upper} &= +\frac{2}{3} \cdot d + 1 \cdot d = +\frac{5}{3} \cdot d \\
1_{lower} &= -\frac{2}{3} \cdot d + 1 \cdot d = +\frac{1}{3} \cdot d \\
0_{upper} &= +\frac{2}{2} \cdot d + 0 \cdot d = +\frac{2}{3} \cdot d \\
0_{lower} &= -\frac{2}{3} \cdot d - 0 \cdot d = -\frac{2}{3} \cdot d \\
-1_{upper} &= +\frac{2}{3} \cdot d - 1 \cdot d = -\frac{1}{3} \cdot d \\
-1_{lower} &= -\frac{2}{3} \cdot d - 1 \cdot d = -\frac{5}{3} \cdot d \\
-2_{upper} &= +\frac{2}{3} \cdot d - 2 \cdot d = -\frac{4}{3} \cdot d \\
-2_{lower} &= -\frac{2}{3} \cdot d - 2 \cdot d = -\frac{8}{3} \cdot d
\end{aligned}$$

Thanks to these values, we end up with the plot in Figure A.5, whose structure is the same as the one of Figure A.3, which was previously defined.

The regions associated with each digit cannot be delimited using horizontal lines, which means that we need to take into account the digits of the divisor d as well as those of A and that the boundaries

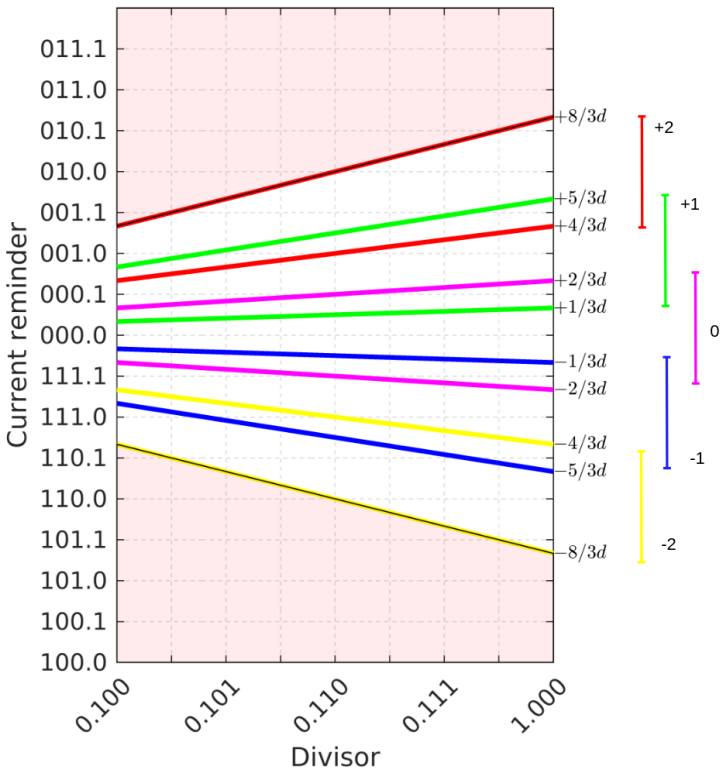

Figure A.5: p-d plot for SRT radix-4, with minimum binary set

are now broken lines. We can thus partition the plot, and associate a digit of the quotient to each partition. This is what is done in Figure A.6, using the black lines as boundaries. Thanks to this separation, it is possible to build the look-up table using as inputs 6 digits from A and 3 digits from d. Since there is no available look-up table on the internet for this algorithm, radix and digit set, its declaration can be found in its entirety in Table A.2. Using these few principles, we can come up with an SRT division algorithm with radix 2^k.

As it happened in the previous case, the areas above $\frac{8}{3} \cdot d$ and below $-\frac{8}{3} \cdot d$ are not feasible, due to the constraints we put on the operands. In spite of that, we still need to insert the points in those areas in the look-up table and associate them to $q_i = 2$ or $q_i = -2$, respectively. The code which implements this algorithm can be found in Appendix B, since it is fundamental to understand the structure of the division circuit.

Now we have all the knowledge to understand what happened in the Pentium division unit in 1990 [2]: Intel used a radix-4 SRT division with a look-up table similar to the one presented. However, while defining this table using the p-d plot, the boundaries were computed incorrectly and some areas were associated to the wrong digits: when an (A, d) couple belonged to these areas, the result ended up being incorrect. These issues had not been discovered during the verification of the chip due to fact that the probability of using these incorrect entries was very low (one in ten billion). Regardless, the problem was noticed by the users and Intel had to pay $475 million to recall the defective processors and fix the issue.

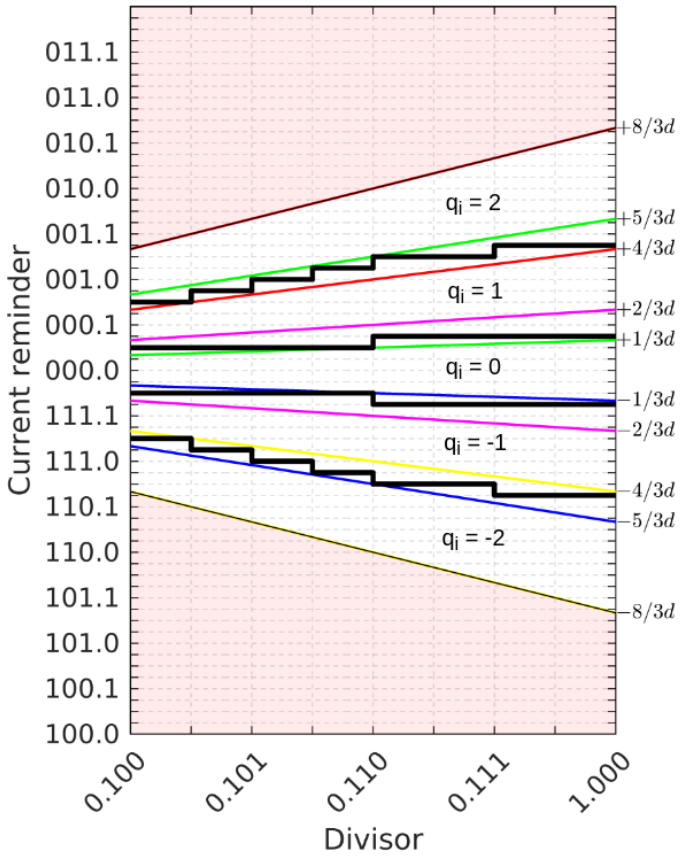

Figure A.6: p-d plot for radix 4 showing the boundaries to select the digits

Table A.2: Possible look up table for radix-4 SRT using minimally redundant digit set $\{-2, -1, 0, +1, +2\}$

B	A	q_i	B	A	q_i	B	A	q_i	B	A	q_i	B	A	q_i	B	A	q_i	B	A	q_i	B	A	q_i
000	000000	0	001	000000	0	010	000000	0	011	000000	0	100	000000	0	101	000000	0	110	000000	0	111	000000	0
000	000001	0	001	000001	0	010	000001	0	011	000001	0	100	000001	0	101	000001	0	110	000001	0	111	000001	0
000	000010	1	001	000010	1	010	000010	1	011	000010	1	100	000010	0	101	000010	0	110	000010	0	111	000010	0
000	000011	1	001	000011	1	010	000011	1	011	000011	1	100	000011	1	101	000011	1	110	000011	1	111	000011	1
000	000100	1	001	000100	1	010	000100	1	011	000100	1	100	000100	1	101	000100	1	110	000100	1	111	000100	1
000	000101	1	001	000101	1	010	000101	1	011	000101	1	100	000101	1	101	000101	1	110	000101	1	111	000101	1
000	000110	2	001	000110	1	010	000110	1	011	000110	1	100	000110	1	101	000110	1	110	000110	1	111	000110	1
000	000111	2	001	000111	2	010	000111	1	011	000111	1	100	000111	1	101	000111	1	110	000111	1	111	000111	1
000	001000	2	001	001000	2	010	001000	2	011	001000	1	100	001000	1	101	001000	1	110	001000	1	111	001000	1
000	001001	2	001	001001	2	010	001001	2	011	001001	2	100	001001	1	101	001001	1	110	001001	1	111	001001	1
000	001010	2	001	001010	2	010	001010	2	011	001010	2	100	001010	2	101	001010	2	110	001010	1	111	001010	1
000	001011	2	001	001011	2	010	001011	2	011	001011	2	100	001011	2	101	001011	2	110	001011	2	111	001011	2
000	001100	2	001	001100	2	010	001100	2	011	001100	2	100	001100	2	101	001100	2	110	001100	2	111	001100	2
000	001101	2	001	001101	2	010	001101	2	011	001101	2	100	001101	2	101	001101	2	110	001101	2	111	001101	2
000	001110	2	001	001110	2	010	001110	2	011	001110	2	100	001110	2	101	001110	2	110	001110	2	111	001110	2
000	001111	2	001	001111	2	010	001111	2	011	001111	2	100	001111	2	101	001111	2	110	001111	2	111	001111	2
000	010000	2	001	010000	2	010	010000	2	011	010000	2	100	010000	2	101	010000	2	110	010000	2	111	010000	2
000	010001	2	001	010001	2	010	010001	2	011	010001	2	100	010001	2	101	010001	2	110	010001	2	111	010001	2
000	010010	2	001	010010	2	010	010010	2	011	010010	2	100	010010	2	101	010010	2	110	010010	2	111	010010	2
000	010011	2	001	010011	2	010	010011	2	011	010011	2	100	010011	2	101	010011	2	110	010011	2	111	010011	2
000	010100	2	001	010100	2	010	010100	2	011	010100	2	100	010100	2	101	010100	2	110	010100	2	111	010100	2
000	010101	2	001	010101	2	010	010101	2	011	010101	2	100	010101	2	101	010101	2	110	010101	2	111	010101	2
000	010110	2	001	010110	2	010	010110	2	011	010110	2	100	010110	2	101	010110	2	110	010110	2	111	010110	2
000	010111	2	001	010111	2	010	010111	2	011	010111	2	100	010111	2	101	010111	2	110	010111	2	111	010111	2
000	011000	2	001	011000	2	010	011000	2	011	011000	2	100	011000	2	101	011000	2	110	011000	2	111	011000	2
000	011001	2	001	011001	2	010	011001	2	011	011001	2	100	011001	2	101	011001	2	110	011001	2	111	011001	2
000	011010	2	001	011010	2	010	011010	2	011	011010	2	100	011010	2	101	011010	2	110	011010	2	111	011010	2
000	011011	2	001	011011	2	010	011011	2	011	011011	2	100	011011	2	101	011011	2	110	011011	2	111	011011	2
000	011100	2	001	011100	2	010	011100	2	011	011100	2	100	011100	2	101	011100	2	110	011100	2	111	011100	2
000	011101	2	001	011101	2	010	011101	2	011	011101	2	100	011101	2	101	011101	2	110	011101	2	111	011101	2
000	011110	2	001	011110	2	010	011110	2	011	011110	2	100	011110	2	101	011110	2	110	011110	2	111	011110	2
000	011111	2	001	011111	2	010	011111	2	011	011111	2	100	011111	2	101	011111	2	110	011111	2	111	011111	2
000	100000	-2	001	100000	-2	010	100000	-2	011	100000	-2	100	100000	-2	101	100000	-2	110	100000	-2	111	100000	-2
000	100001	-2	001	100001	-2	010	100001	-2	011	100001	-2	100	100001	-2	101	100001	-2	110	100001	-2	111	100001	-2
000	100010	-2	001	100010	-2	010	100010	-2	011	100010	-2	100	100010	-2	101	100010	-2	110	100010	-2	111	100010	-2
000	100011	-2	001	100011	-2	010	100011	-2	011	100011	-2	100	100011	-2	101	100011	-2	110	100011	-2	111	100011	-2
000	100100	-2	001	100100	-2	010	100100	-2	011	100100	-2	100	100100	-2	101	100100	-2	110	100100	-2	111	100100	-2
000	100101	-2	001	100101	-2	010	100101	-2	011	100101	-2	100	100101	-2	101	100101	-2	110	100101	-2	111	100101	-2
000	100110	-2	001	100110	-2	010	100110	-2	011	100110	-2	100	100110	-2	101	100110	-2	110	100110	-2	111	100110	-2
000	100111	-2	001	100111	-2	010	100111	-2	011	100111	-2	100	100111	-2	101	100111	-2	110	100111	-2	111	100111	-2
000	101000	-2	001	101000	-2	010	101000	-2	011	101000	-2	100	101000	-2	101	101000	-2	110	101000	-2	111	101000	-2
000	101001	-2	001	101001	-2	010	101001	-2	011	101001	-2	100	101001	-2	101	101001	-2	110	101001	-2	111	101001	-2
000	101010	-2	001	101010	-2	010	101010	-2	011	101010	-2	100	101010	-2	101	101010	-2	110	101010	-2	111	101010	-2
000	101011	-2	001	101011	-2	010	101011	-2	011	101011	-2	100	101011	-2	101	101011	-2	110	101011	-2	111	101011	-2
000	101100	-2	001	101100	-2	010	101100	-2	011	101100	-2	100	101100	-2	101	101100	-2	110	101100	-2	111	101100	-2
000	101101	-2	001	101101	-2	010	101101	-2	011	101101	-2	100	101101	-2	101	101101	-2	110	101101	-2	111	101101	-2
000	101110	-2	001	101110	-2	010	101110	-2	011	101110	-2	100	101110	-2	101	101110	-2	110	101110	-2	111	101110	-2
000	101111	-2	001	101111	-2	010	101111	-2	011	101111	-2	100	101111	-2	101	101111	-2	110	101111	-2	111	101111	-2
000	110000	-2	001	110000	-2	010	110000	-2	011	110000	-2	100	110000	-2	101	110000	-2	110	110000	-2	111	110000	-2
000	110001	-2	001	110001	-2	010	110001	-2	011	110001	-2	100	110001	-2	101	110001	-2	110	110001	-2	111	110001	-2
000	110010	-2	001	110010	-2	010	110010	-2	011	110010	-2	100	110010	-2	101	110010	-2	110	110010	-2	111	110010	-2
000	110011	-2	001	110011	-2	010	110011	-2	011	110011	-2	100	110011	-2	101	110011	-2	110	110011	-2	111	110011	-2
000	110100	-2	001	110100	-2	010	110100	-2	011	110100	-2	100	110100	-2	101	110100	-2	110	110100	-2	111	110100	-2
000	110101	-2	001	110101	-2	010	110101	-2	011	110101	-2	100	110101	-2	101	110101	-2	110	110101	-1	111	110101	-1
000	110110	-2	001	110110	-2	010	110110	-2	011	110110	-2	100	110110	-1	101	110110	-1	110	110110	-1	111	110110	-1
000	110111	-2	001	110111	-2	010	110111	-2	011	110111	-1	100	110111	-1	101	110111	-1	110	110111	-1	111	110111	-1
000	111000	-2	001	111000	-2	010	111000	-1	011	111000	-1	100	111000	-1	101	111000	-1	110	111000	-1	111	111000	-1
000	111001	-2	001	111001	-1	010	111001	-1	011	111001	-1	100	111001	-1	101	111001	-1	110	111001	-1	111	111001	-1
000	111010	-1	001	111010	-1	010	111010	-1	011	111010	-1	100	111010	-1	101	111010	-1	110	111010	-1	111	111010	-1
000	111011	-1	001	111011	-1	010	111011	-1	011	111011	-1	100	111011	-1	101	111011	-1	110	111011	-1	111	111011	-1
000	111100	-1	001	111100	-1	010	111100	-1	011	111100	-1	100	111100	-1	101	111100	-1	110	111100	-1	111	111100	-1
000	111101	-1	001	111101	-1	010	111101	-1	011	111101	-1	100	111101	0	101	111101	0	110	111101	0	111	111101	0
000	111110	0	001	111110	0	010	111110	0	011	111110	0	100	111110	0	101	111110	0	110	111110	0	111	111110	0
000	111111	0	001	111111	0	010	111111	0	011	111111	0	100	111111	0	101	111111	0	110	111111	0	111	111111	0

APPENDIX B

Division implementation

B.1 Introduction

The purpose of this chapter is to introduce and explain the hardware implementation of the Radix-4 SRT division algorithm using the *minimally redundant digit set*. The algorithm will first be presented in the form of high-level code and only some information about the operations to be performed will be given: the detailed explanation of all the assumptions and the reason why some operations are performed can be found in Appendix A.

The starting point of the circuit is the algorithm to be implemented, and its code is presented in Listing B.1. The division circuit is split into two sub-units: a controller which keeps track of what operation should be performed and a datapath which actually performs the necessary operations.

```
1  std::pair<SLV::std_logic_vector, SLV::std_logic_vector>
2  Division::srt_division_radix_4_digit_2
3  (SLV::std_logic_vector& dividend, SLV::std_logic_vector & divisor){
4
5      // PREPROCESSING
6      int N = dividend.size;
7
8      // Also detects divide by zero operation
9      if(dividend.to_base_10() >= (divisor.to_base_10() << N/2)
10         throw std::invalid_argument("Can't compute with these values");
11
12     // Temporary reminder
13     SLV::std_logic_vector A (N   + 3);
14     // Divisor
15     SLV::std_logic_vector B (N/2 + 2);
16     // Quotient
17     SLV::std_logic_vector Q (N/2 + 2);
18
19     // Leading zeros of the divisor
20     unsigned int k = divisor.extract(N/2 - 1, 0).count_leading_zeros();
21
22     A = SLV::std_logic_vector("000") & (dividend << k);
23     // Since the divisor and the dividend are both N long, the real divisor is in the
24     // bottom part of "divisor"
25     B = SLV::std_logic_vector("00") & (divisor.extract(N/2 - 1, 0) << k);
26
27     // DIVISION
28     for(int i = 0; i <= (N/2 >> 1) ; i++){
29         SLV::std_logic_vector A_upper  = A.extract(N + 1, N/2);
30         SLV::std_logic_vector A_bottom = A.extract(N/2 - 1, 0);
31
```

```
        SLV::std_logic_vector partial_R(N/2 + 2);

        int current_q = srt_division_radix_4_lut_digit_2(
            B.extract(N/2 - 2, N/2 - 4) &
            A.extract(N + 2, N - 3)
        );

        if(current_q == 2){
            Q = (Q << 2) + 2;
            partial_R = A_upper - (B << 1);
        }
        if(current_q == 1){
            Q = (Q << 2) + 1;
            partial_R = A_upper - B;
        }
        if(current_q == 0){
            Q = Q << 2;
            partial_R = A_upper;
        }
        if(current_q == -1){
            Q = (Q << 2) - 1;
            partial_R = A_upper + B;
        }
        if(current_q == -2){
            Q = (Q << 2) - 2;
            partial_R = A_upper + (B << 1);
        }

        A = (SLV::std_logic_vector("0") & partial_R & A_bottom) << 2;
    }

    // SIGN CORRECTION
    if(A.get_msb() == 1){
        Q = Q - 1;
        A = (SLV::std_logic_vector("0") &
             ((SLV::std_logic_vector("0") & A.extract(N + 2, N/2 + 2)) + B)
             & A.extract(N/2 + 1, 2) ) << 2;
    }

    Q = Q.extract(N/2 - 1, 0);
    SLV::std_logic_vector R = A.extract(N + 1, N/2 + 2) >> k;

    return std::pair<SLV::std_logic_vector, SLV::std_logic_vector>
    (Q, R);
}
```

Listing B.1: The C++ code implementing the Radix-4 SRT division

B.2 Controller

The basic operations to be performed are:

1. Preprocess the dividend and the divisor;
2. Divide;
3. Correct the quotient and the remainder, if necessary;
4. Postprocess the quotient and the result.

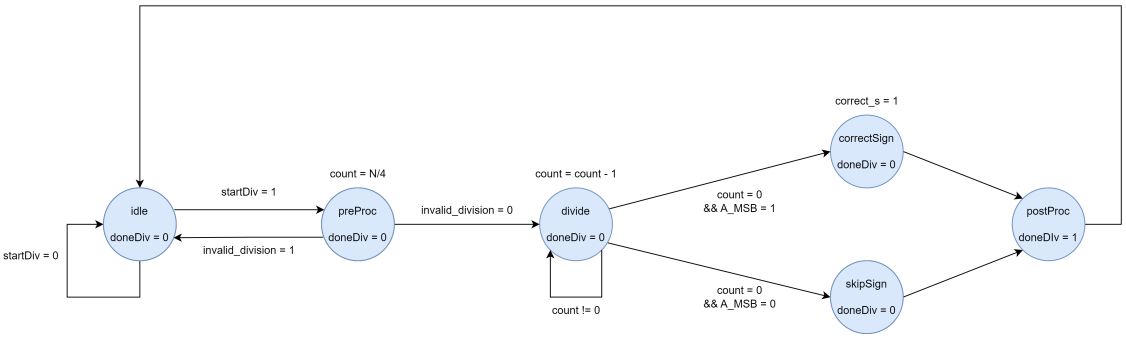

Figure B.1: Simplified sate diagram of the controller

The details of these procedures are given in section B.3.

The controller has one state for each activity plus the `idle` state, and a simplified state diagram (most of the control signals are omitted for brevity) is in Figure B.1. It's important to know that N represents the length of the **dividend**, which in AFTAB is the width of a register, so 32 bits. Upon reset, the controller is initialised to the `idle` state and waits for the processor's control unit to assert the `startDiv` signal, thus requesting a division to be performed.

After that, the controller moves into the `preProc` state and asserts the control signals to perform the preprocessing of the operands.

The next state depends on the `invalid_division` signal, which is generated by the datapath and represents, as the name implies, whether the division is valid or not due to the values of dividend and divisor (which also includes the case of a division by 0). Assuming the division is feasible, the controller moves to the `divide` state: here the quotient and remainder are iteratively calculated. Since two digits of the quotient are computed per clock cycle and the quotient is $N/2 = 16$ bits long, the controller stays in this state for 8 clock cycles.

After all the iterations of the division have been performed, the controller's next state depends on the remainder's sign: if it's negative both the quotient and the remainder itself need to be corrected, so the controller moves to the `correctSign` state and asserts some signals to instruct the datapath to perform the sign-correction operations. If the remainder is not negative then the quotient and remainder are correct, so the division is concluded, and all that is missing is the postprocessing. Nonetheless, in this case the controller moves to the `skipSign` state for consistency: it's undesirable for an operation's execution to take a variable amount of clock cycles depending on its inputs' values. Thus, to make sure the division has a constant duration in terms of clock cycles, the datapath remains idle while the controller is in the `skipSign` state.

After the division is completed the operands need to be postprocessed, so regardless of whether the sign correction was performed or not the controller moves to the `postProc` state. In this state that final outputs of the division circuit are made ready to be used by any unit outside of the device and a signal that warns the processor's control unit is asserted.

These results are available for one clock cycle (technically less than one, since postprocessing takes some time), and after that the controller goes back to the `idle` state and will wait to be instructed to run the division again.

B.3 Datapath

The datapath is split into a wrapper and the actual division unit: the former is built around the latter because the SRT division requires its operands to be normalised (so preprocessed), and then will produce a quotient and a remainder that will have to be postprocessed to match some characteristics

of the original input operands (like signs).

B.3.1 Wrapper

The diagram of the wrapper is shown in Figure B.2 (most control signals are omitted for brevity): all the blocks above the `SRT Divider` unit are employed in the preprocessing phase, while all the blocks below it are used while postprocessing. The registers (represented by rectangles with a triangle inside them) are used to persist some information generated during preprocessing that is needed while postprocessing.

Preprocessing

The purpose of this phase is to turn the input operands, which can be signed or unsigned integers, into numbers that comply with the assumptions made by the SRT divider: the dividend has to be a 32-bit unsigned number, while the divisor has to be a 16-bit unsigned number with 1 as its MSB. Before normalising the operands, they need to be converted (if necessary) to unsigned numbers by complementing them. To do so a copy of the operands is complemented and both the original values and the complemented ones flow into two multiplexers: these select their output depending on whether the division to be performed is between signed operands (`signedUnsignedBar`, a signal that is controlled by the processor's control unit and that depends on the instruction to be executed, is asserted) and on their inputs' signs (so the MSBs). Specifically, an operand is considered negative if `signedUnsignedBar` and its MSB are both 1: in this case its complement is selected; in all other cases the unchanged input goes through the multiplexer because it's either unsigned or signed positive.

After the inputs have been converted to unsigned numbers, the check on the division's validity is performed: if $2^N \cdot divisor \geq dividend$ and a division should be performed (`startDiv` is asserted), then the division is invalid and the corresponding signal is asserted; this will cause the controller to change state (as previously explained) and the processor's control unit will handle this situation. In the meantime, in order to normalise the divisor, its $N/2$ MSBs are truncated and a component counts the leading zeroes of the resulting sub-string: this quantity is called `shift_amount` and is used by a left-shifter to move the first 1 of the shortened divisor to its MSB. In order to preserve the ratio between the orders of magnitude of the dividend and the divisor, we also left-shift the dividend by `shift_amount`, which is also saved in the register `shiftAmountReg`: we need to keep this information as we'll need it to compensate for these left-shifts during postprocessing.

We also need to save information about what the sign of quotient and remainder should be once they're calculated, and we need to do this now as only during preprocessing do we have access to the original inputs: the sign of the quotient is the product of the dividend's and divisor's signs (logically speaking it's an XOR), while the sign of the remainder is the same as dividend's. This information is stored as two bits in the register `negResult`: its MSB is 1 if the final quotient should be negative and the LSB is 1 if the final remainder should be negative. Once all the operations explained so far have been executed, the preprocessed operands are passed to the division circuit.

Postprocessing

The division circuit will output quotient an remainder as 16-bit unsigned numbers. In order to restore the ratio between the orders of magnitude of the dividend/divisor and the remainder, the latter first needs to be right-shifted by `shift_amount` (which was saved in `shiftAmountReg`). The quotient, on the other hand, is of the right order of magnitude and doesn't need to be changed at this time.

After this first operation, both quotient and remainder are resized to match the length of the inputs by simply pushing $N/2$ zeroes as their MSBs, so that the resized numbers are still unsigned and their value is unchanged.

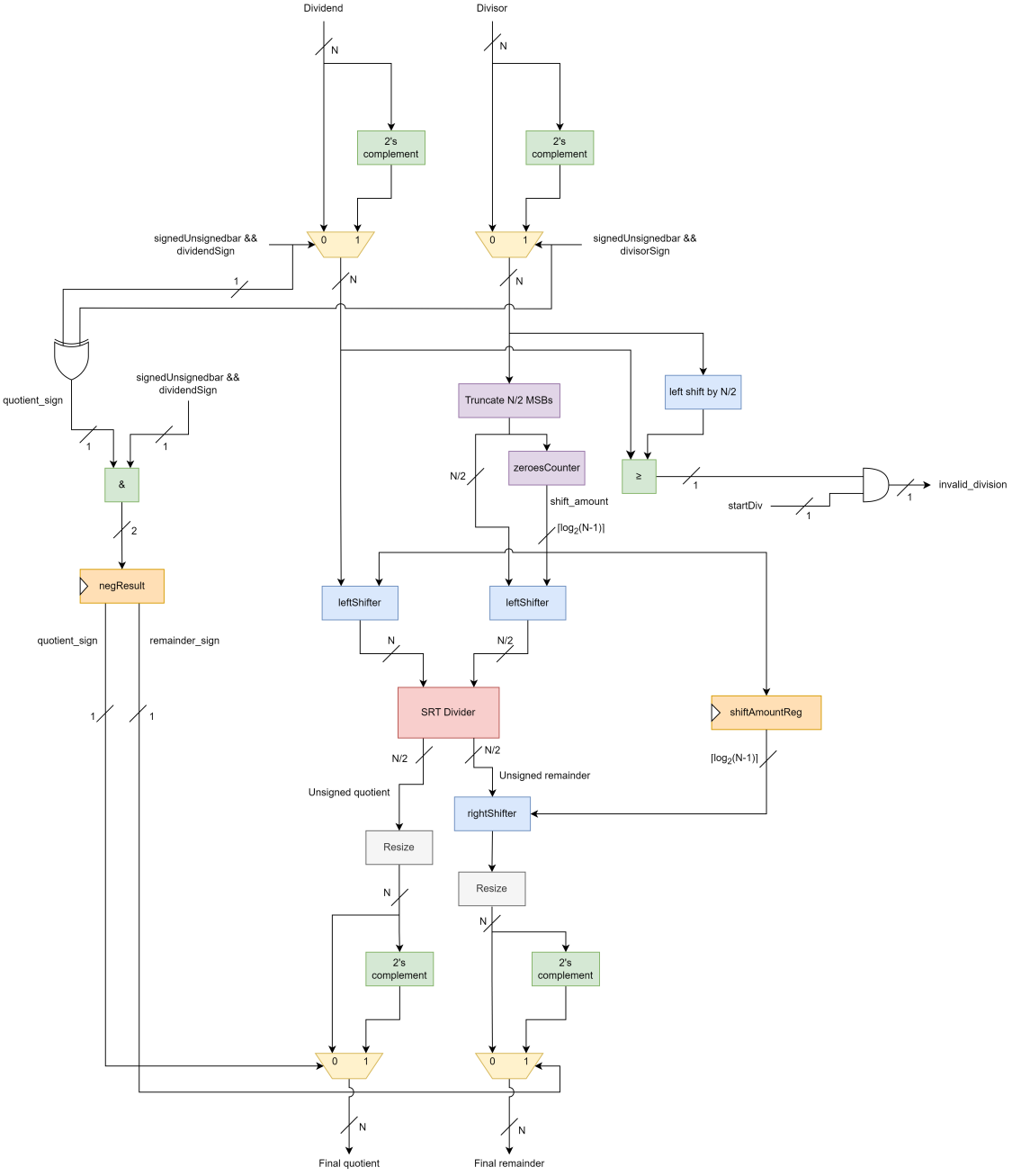

Figure B.2: Diagram of the wrapper

The last step of postprocessing consists in changing the quotient and remainder's sign, if needed. The way this operation is performed is similar to the way the dividend's and the divisor's signs were handled: copies of the resized quotient and remainder are complemented, and both the original values and their complements (which are always negative) flow into two multiplexers controlled by the information that was saved in `negResult`: if the MSB of this register is 1 the quotient has to be negative (its complement is chosen), else the positive quotient is chosen; if the LSB of this register is 1 the remainder has to be negative, else the positive remainder is chosen.

Thus we obtain the final quotient and remainder: they are passed to the processor which will use them as needed.

B.3.2 Divider

The diagram of the divider is shown in Figure B.3 (most control signals are not shown for brevity).

In this component there are three registers (represented as they were in the wrapper):

- `A` will contain the dividend at first, and then the intermediate remainders while they're calculated;
- `B` will contain the divisor;
- `Q` will contain the quotient while it's calculated.

Let's begin by analysing register `B`: its value is only written with the preprocessed divisor after two zeroes have been pushed as its MSB, and this happens when the controller is in the `preProc` state.

Register `Q`'s value is initialised to zero before each division, and it's iteratively computed by adding ± 1, ± 2 or 0 to its current value left-shifted by 2. The aforementioned numbers are the digits computed during the current iteration of the division and are generated by the `digit LUT` component, which is a memory implementing the digit selection table as shown in Table A.2. The address used to access this memory is obtained by concatenating register `B`'s bits with indices in the range $[N/2-2, N/2-4]$ with register `A`'s bits with indices in the range $[N+2, N-3]$. This means that this address is 9 bits wide and that the memory is made of 512 cells. If the correction of the remainder's sign is needed at the end of the division (so the `correct_s` signal is asserted), `Q`'s value is decremented by 1. Its value is finally provided to the wrapper after the two MSBs are truncated.

`A`'s handling, on the other hand, is more complex: its next value depends on both the controller, which will choose which operation should be performed by using the `sel_A` signal, and on the quotient digit q that has been calculated during the current iteration. The possible operations to be performed depend on the controller's state:

- when `sel_A` is "00" the controller is in the `preProc` state, so the preprocessed divisor is saved into `A` after three zeroes are pushed as its MSB;
- when `sel_A` is "01" the controller is in the `divide` state, so `A` is loaded with the value of the next intermediate remainder;
- when `sel_A` is "10" the controller is in the `correctSign` state, so `A` is loaded with the sign-corrected remainder.

The quotient digits q play a role during the division iterations, as they select what number should be added to `A`'s current value. The way this is implemented is not trivial: in the purely mathematical

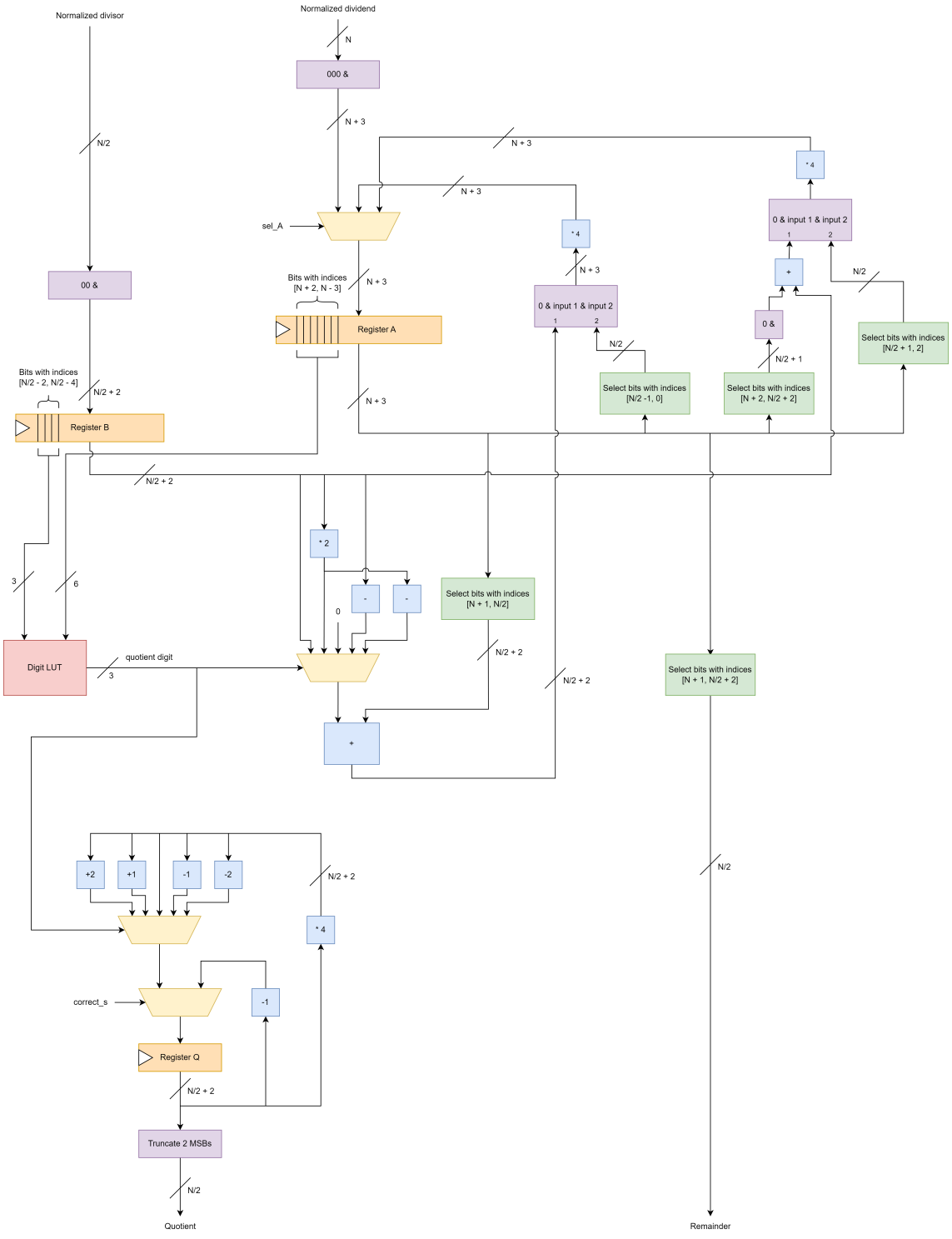

Figure B.3: Diagram of the divider

algorithm, the current remainder is added to a *factor* whose value depends on q:

$$factor = \begin{cases} -2 \cdot (divisor \cdot 2^{N/2}) & \text{if } q = 2 \\ -(divisor \cdot 2^{N/2}) & \text{if } q = 1 \\ 0 & \text{if } q = 0 \\ divisor \cdot 2^{N/2} & \text{if } q = -1 \\ 2 \cdot (divisor \cdot 2^{N/2}) & \text{if } q = -2 \end{cases}$$

The result of this sum is then multiplied by 4. The implementation of this operation can be simplified because of the multiplication by $2^{N/2}$: this means that the $N/2$ LSBs of this product are all zeroes, so by adding it to the current remainder we can see that the $N/2$ LSBs of the sum are just those of the current remainder, unchanged. So we can compute the *factor* without multiplying the divisor by $2^{N/2}$, and then add it to the current remainder after excluding its $N/2$ LSBs. To the result of this sum are then concatenated the bits that were just excluded; a 0 is pushed as MSB (to restore the sum's length to that of the register), and then the concatenation result is multiplied by 4 (this will shift out the 0 we pushed as MSB, making its value unimportant): the final result of this process is the same as the previous one, but the operands of the sum are half as long. We decided that the next remainder value saved into `A` should be computed using this optimised strategy.

The procedure described thus far is used to compute the next remainder, but is not the one used to correct its sign should the final one be negative: this can happen since, as seen in Figure A.5, several digits might be associated to the same (*remainder*, *divisor*) couple: this might cause the remainder to be negative and the quotient to be one more than expected. It should be noted that this is not mathematically wrong, it's just an uncommon way of representing the results of a division: for example, $15/6 \rightarrow (q = 2, r = 3)$ or $(q = 3, r = -3)$. In order to represent quotient and remainder in the canonical form we need to decrease the quotient by one (as already stated while analysing the `Q` register's values) and subtract `B` to the remainder that was calculated previously, before it was shifted left by 2. The implementation of the remainder correction can once again be optimised: we can undo the shift by selecting the right bits of `A`, which are the bits in the range $[N + 2, 2]$; push a 0 as the MSB; isolate the least significant $N/2$ bits because we need to add `B` to the most significant ones; concatenate the isolated part of `A` to the result of the sum and push a 0 as the MSB; multiply this result by 4.

The remainder thus obtained requires a final truncation before it can be passed to the wrapper: since the circuit's outputs are expected to be $N/2$ bits long, from `A`'s value we remove the MSB, which is the sign, and the $N/2 + 2$ LSBs.

APPENDIX C

Multiplication implementation

C.1 Introduction

The purpose of this chapter is to introduce and explain the hardware implementation of Booth's signed multiplication algorithm. This algorithm is presented in the form of pseudo-code in Listing C.1, where N is the length of multiplicand A and multiplier B, so their product P will be $2 \cdot N$ bits long; in our processor $N = 32$. The contents of the look-up table (LUT) are shown in Table C.1.

The multiplication circuit is split into two sub-units: a controller which keeps track of how many intermediate products still need be computed, and a datapath which actually performs the necessary operations.

```
int booth_mul(int A, int B) {
    i = 0;
    P = 0;

    B = B << 1;

    while (i < N/2){
        factor = LUT(B[2:1])

        P = P + A * factor;
        A = A << 2;
        B = B >> 2;
        i = i + 1;
    }

    return P;
}
```

Listing C.1: The C-like psuedo code implementing the radix-4 booth multiplication

C.2 Controller

A simplified view of the states of the controller is shown in Figure C.1 (some control signals are not shown for brevity). It's a very simple controller because the operations executed at each clock cycle are always the same.

Upon reset, the controller is initialised to the `idle` state and waits for the processor's control unit to assert the `startMul` signal, thus requesting a multiplication to be performed.

B[2]	B[1]	B[0]	factor
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Table C.1: Contents of the LUT used in Booth's algorithm

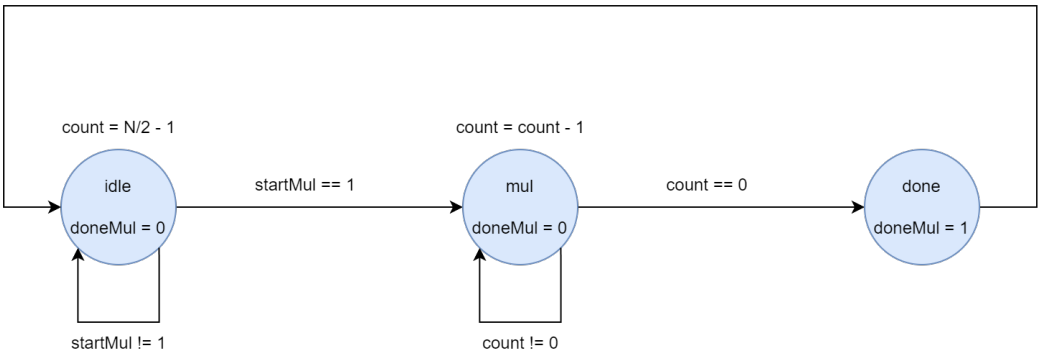

Figure C.1: Simplified sate diagram of the controller

After that, the controller moves into the `mul` state and asserts the control signals needed to compute the intermediate products. Since the multiplicand and multiplier are $N = 32$ bits long, the controller stays in this state for $N/2 - 1 = 15$ clock cycles.

After the final product has been computed the controller moves to the `done` state: the purpose of this state is to let the controller know that the product is ready, and to do so the `doneMul` signal is raised (no other operation is performed).

The result is available for one clock cycle: after that the controller goes back to the `idle` state and will wait to be instructed to run the multiplication again.

C.3 Datapath

The datapath, just like the controller, is very simple and is shown in Figure C.2. Its purpose is to compute the intermediate products, so it implements the body of the *while* loop as it was shown in the code.

While the controller is in the `idle` state, the multiplicand A is sign-extended to $2 \cdot N = 64$ bits to match the length of the product P while a zero is appended to the multiplicand B's LSB, which is equivalent to left-shifting B ny one position.

The main difference between this implementation and the code is in the absence of the LUT: it is replaced by computing in parallel 0, $\pm A$, $\pm 2 \cdot A$ and using B's LSBs as the selection signal of a multiplexer, without involving a memory component. The output of the multiplexer, which is $A \cdot \text{factor}$, is summed to P's current value to generate the next partial product, which will be written to the product's register at the next rising edge of the clock.

It's important to know that both A's and B's registers are shift registers: A's register left-shifts

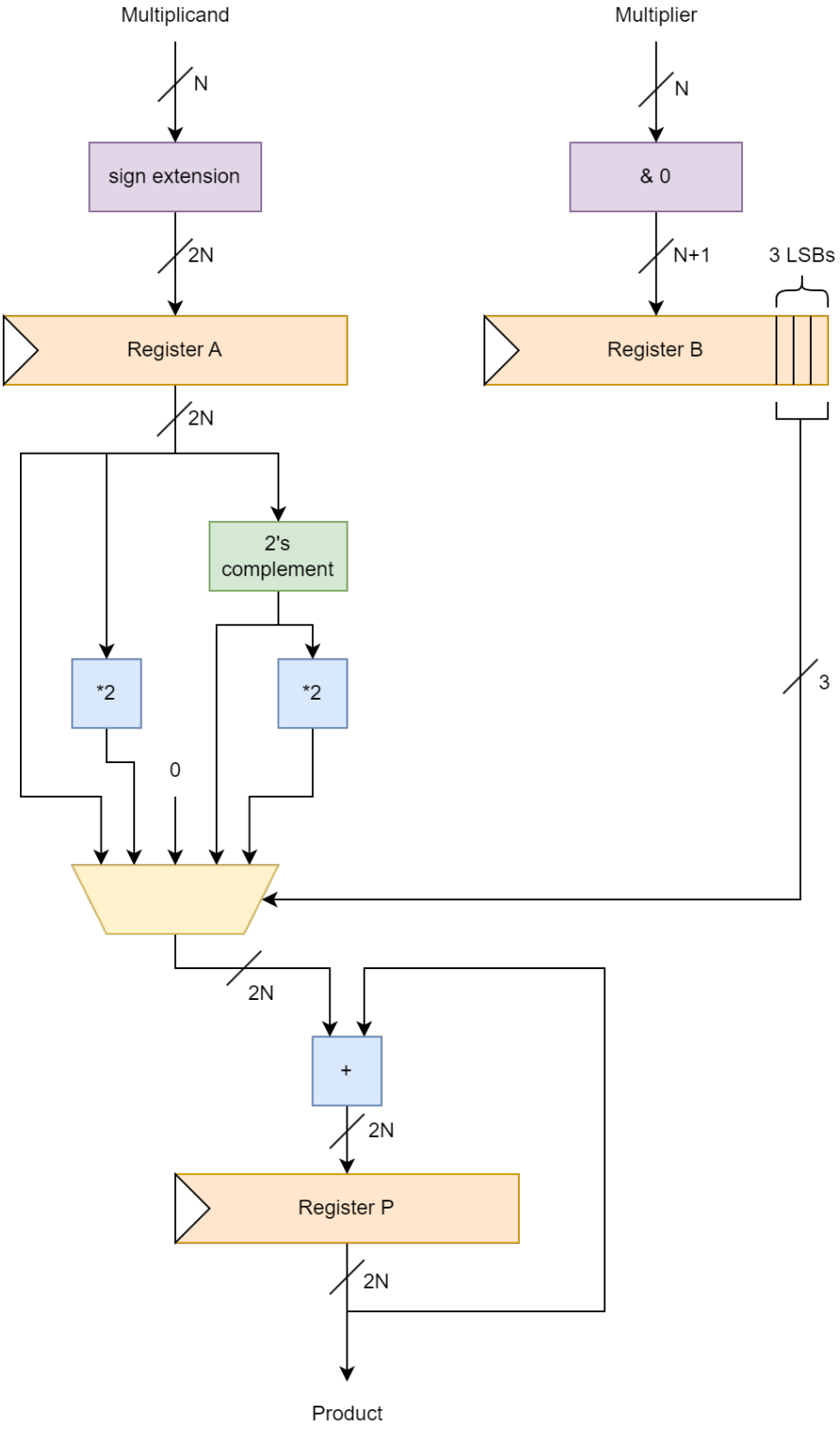

Figure C.2: Diagram of the datapath

the multiplicand by two positions at each clock cycle, while B's register right-shifts the multiplier by the same amount.

The value in the register P is accessed as-is from the outside.

Bibliography

[1] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford series in electrical and computer engineering. Oxford University Press, 2010. ISBN: 9780195328486. URL: https://books.google.it/books?id=tEo%5C_AQAAIAAJ.

[2] Vaughan Pratt. "Anatomy of the Pentium bug". In: *TAPSOFT '95: Theory and Practice of Software Development*. Ed. by Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 97–107. ISBN: 978-3-540-49233-7.

[3] A.L. Ruiz et al. *Arithmetic and Algebraic Circuits*. Intelligent Systems Reference Library. Springer International Publishing, 2021. ISBN: 9783030672669. URL: https://books.google.it/books?id=7OQfEAAAQBAJ.