Politecnico di Torino

Integrated System Architecture

A RISC-V Processor: Design & Development

Project Report

Master's Degree in Electronic Engineering

Referents: Prof. Martina Maurizio, Prof. Masera Guido

Rovere Enrico, Santoro Dino, Sarda Giuseppe, Sartoni Sandro

September 11, 2019

Contents

Introduction

The aim of the presented work is modeling a processor core compliant with the RISC-V specifications as of the official website[1]. What the group did was choose a subset of the RISC-V Instruction Set Architecture (ISA) and produce the HDL modelling the resultant architecture.

The RISC-V ISA is modular, meaning that, given a defined and fixed word size (32 bits in this case), a given core must support only the **[I] - Integer** subset. Of course, the architecture may be more complex and include optional *extensions*, like **[M] - Integer Multiplication and Division**, and many more. The software to be run on the device is then compiled specifying which subsets the target supports, replacing unsupported instruction with a software library.
Since we wanted to implement an Core as general purpose as possible, we opted for the I and M extension, implementing a **RV32IM** architecture. Following, the set of instructions supported:

Mnemonics	**Mnemonics**	**Mnemonics**
add	lui	lh
addi	suipc	lw
sub	slt	sb
and	slti	sh
andi	sltu	sw
or	sltiu	lbu
ori	beq	lhu
xor	bne	mul
xori	bge	mulh
sll	bgeu	mulhsu
slli	blt	mulhu
sra	bltu	div
srai	jal	divu
srl	jalr	rem
srli	lb	remu

Table 1: Set of Instructions supported by our Core

Originally, the RV32I set of instructions includes some additional instructions needed for hardware threading, we decided not to support Operating Systems and keep the implementation as straightforward as possible, thus not including these instructions.

[1] RISCV Specifications Website: https://riscv.org/specifications/

CHAPTER 1

Control Unit

The Control Unit is effectively the brains of the device, manipulating data flowing through the datapath.

1.1 Overview

The group chose a "hardwired" approach for the present unit, creating a Look Up Table (LUT) of Control Words (CW) indexed by opcode and sequentially fed to the datapath control signals. Once a CW is fetched, it is progressively shifted to separate signals - *cw1, cw2, cw3* - each mapped to the port it is meant to control.
Moreover, R-type and I-type instructions require decoding of their FUNC field, which determines the operation to be performed by the ALU on the operands. Therefore, the CU includes a decoding unit for this purpose, producing an *ALU_control* signal which is then fed to the ALU, delayed by an appropriate amount of clock periods.

1.2 Control Hazards

Despite the fact that this architecture employs a Forwarding Unit, described in Section 2.3.1, some Read-After-Write data dependencies may not be solved at runtime, requiring a stall of the pipeline. For any hazardous pair of instructions, the possible cases are the following:

	IF	**ID**	**EX**	**MEM**	**WB**	**Duration**
1) *any* 2) Branch	S					1 clk cycle
1) Load 2) *any*	S	S				1 clk cycle
1) Load 2) Branch	S					Depends on memory latency

The stages labelled by 'S' are stalled for the specified number of clock cycles, and between them and those unaffected a 'bubble' - a NOP Control Word portion - is inserted. As far as the Load/Branch case is concerned, the number of clock cycles depends on the Data Memory. Once the datum is provided to the processor, it is then able to determine whether to take the branch or not, and therefore resolve the stall.

CHAPTER 2

Datapath

2.1 Instruction Fetch (IF)

The Instruction Fetch module is in charge of fetching at each cycle an instruction (if available).
We decided to implement this module in order to be as efficient as possible, improving its capabilities by means of a **Branch Prediction Unit**, or **BPU**, a Branch Forwarding Unit, or **BFU**, and an **Instruction Cache** with its relative **controller**.
While the specific modules will be thoroughly explained in the Non-Datapath section, below there's a small explanation of the Fetch Unit that collects all of these modules, interfaces them and provides the instruction fetched and whatever may be needed to the other part of our RISCV Core.

2.1.1 Fetch Unit

The Fetch Unit module of the RISCV Core is comprised of the three inner modules described above plus the necessary logic to interface them. Even if not detailed, it's impossible to describe the Fetch Unit without outlining the non-datapath blocks.
The ICache Controller duty is to retrieve from the Instruction RAM the correct set of instructions, store them correctly in the Cache and handle the cache_miss signal. On the other hand, the Branch Prediction Unit and Branch Forwarding Unit are in charge of predicting the behaviour of a branch and providing the jump address (if necessary), same for any JAL instruction for which, if available, the jumping address is provided (no need to predict the jump here). The JALR instruction cannot be handled by these modules as the jumping address is given by the content of the source register, thus making not sensible to store a value that may change inside a table - all of the stored values are constants if there's no aliasing. The BPU is the core logic that predicts on a branch, the BFU provides the correct operands to the BPU - even in case of Forwarding, hence the name - that are needed in order to check whether the prediction was correct or not.
The rationale of the behaviour of the Fetch Unit is the following: whenever the ICache returns a Miss, everything has to stall until the new block of instructions is correctly stored inside the cache: the icache_controller handles the bytes received from the IRAM controller and reconstructs the original set of instructions. During the normal functioning of the core, the Fetch Unit evaluates the **NPC**, or Next Program Counter, at the rising edge of the clk signal it updates the current PC value and fetches the consequent instruction. Such instruction is provided at the output of the module.
In case of Branch/JAL instruction, the BPU drives the PC accordingly; at the following clock cycle, the Fetch Unit generates the branch outcome according to the type of the branch instruction (e.g. BEQ, BNE, BGE etc) along with all of the control signals that are needed by the BPU. If there's a misprediction or the JAL instruction was executed first time, the next fetched instruction is turned to a NOP before

even going to the Instruction Decode stage.

2.2 Instruction Decode (ID)

The Instruction Decode stage consists of the logic intended to decompose the instructions into its fields:

- **Opcode**: the opcode defines the instruction type, whether it's a r-type instruction, an i-type instruction, a load/store instruction etc.
- **RegisterSource1**: this is the field containing the first source register that has to be provided to the Register File
- **RegisterSource2**: this is the field containing the second source register that has to be provided to the Register File
- **RegisterDestination**: this is the field containing the destination register in which the eventual result will be written
- **ImmediateField**: an immediate numerical value that is encoded in some type of instructions

Moreover, the ID stage includes the Control Unit module - plus all the logic to decompose the control word into all of its components - and a Register File that will be described below.

2.2.1 Register File

The Register File is a set of 32 registers, each one of 32 bits, with the only requirement that the *X0* register is a read-only register that contains a 0 value. It comes with two different implementations, one that consists of registers made of latches - if the target technology is ASIC - the other having registers made of flip flops - in case of FPGA.
The interface of the module can be seen in the picture below:

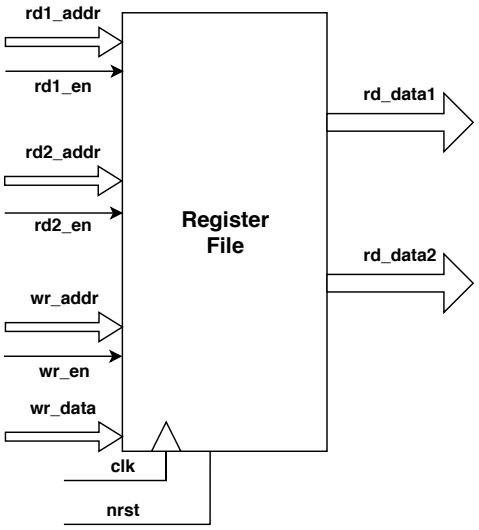

Figure 2.1: Register File Interface

For each input port there's the associated enable signal that controls the related operation: if wr_en is $1'b1$, then it's possible to write the data presented at the wr_data port at the address provided by the wr_addr port (unless the address is equal to $5'h00$, in that case no writing operation is performed). The reading operation is performed in this way: for each port, if the related contorl signal is not enabled then the output will be equal to $32'h00000000$, otherwise the output will be the content of the register requested by rdx_addr (x may be 1 or 2). If there's a read operation in parallel to a write operation and rs1==rd then the register will be written and the value at the wr_data port will be forwarded to the output.

2.3 Execute (EX1)

In the Execute Unit the processor provides the correct operands (even in case of RAW hazard) to the ALU and evaluates the result of the operation required by the current instruction.
This stage is comprised of the two aforementioned modules, the Forwarding Unit and the ALU, and two couples of multiplexers, two of them driven by the Forwarding Unit that selects the correct operand value based on the need to forward or not, the other two needed to select between *op1_execute* and *pc* and between *op2_execute* and *immediate_field*. The output of the latter two MUXs will be the two inputs of the ALU.

2.3.1 Forward Unit

In a pipelined microprocessor, a Forward Unit (FWDU) is mandatory for preventing "bubbles" in the datapath pipeline in the event of a Read-After-Write (RAW) data hazard, that is the present instruction requiring the result of any of the previous two.

To accomplish this task, the FWDU compares source registers of the current instruction with the other two's destinations, and if a collision is detected the appropriate forward path is enabled.

However, the present hazardous instruction may not intend to write a Register File (RF) location and instead simply use the address bits for another purpose, as is the case with load/store or immediate operations: to address this issue, the RF Write Enable signal is also fed to the FWDU.

The FWDU outputs two 2-bit buses, one per ALU input port, each controlling two muxes, each multiplexing between:

- The RF output
- The ALU output
- The output of EX/MEM pipe register

This approach prevents any data hazard blocking the program execution, whereas with control hazards an additional module is integrated in the Control Unit, detailed in Chapter 1.

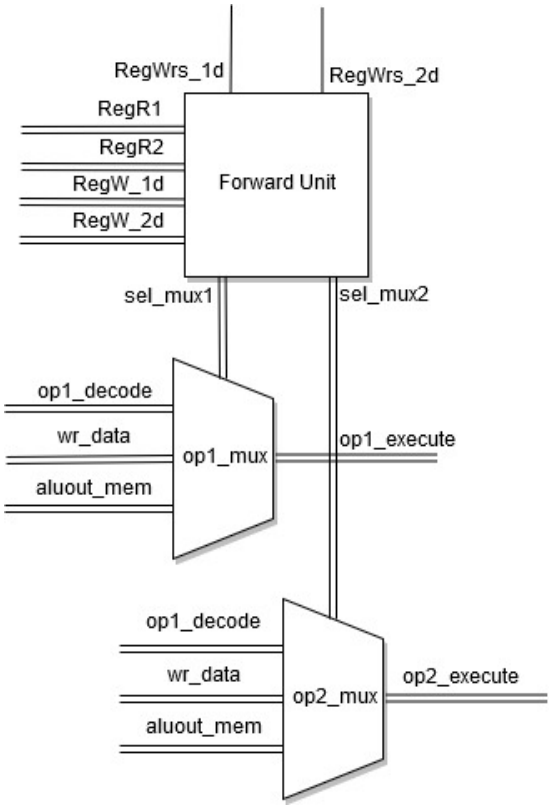

Figure 2.2: Forward Unit

2.3.2 ALU

The unit is used to execute the 29 instructions we implemented that require arithmetic and/or logic operations. Depending on a control signal, named *Control*, we can select the operation the ALU will perform; the organization is presented in the following table:

Mnemonic	Coding	Mnemonic	Coding	Mnemonic	Coding
LUI	4'b0000	XORI	4'b0011	SUB	4'b1001
LB	4'b0001	XOR		SLT	4'b1010
LH		ORI	4'b0100	SLTI	
LW		OR		SLTU	4'b1011
LBU		ANDI	4'b0101	SLTIU	
LHU		AND			
SB SH		SLLI SLL	4'b0110		
SW		SRLI	4'b0111		
ADDI	4'b0010	SRL			
ADD AUIPC		SRAI SRA	4'b1000		

2.4 Multiplication and Division Unit (EX2)

The present unit implements multiplication and division operations and takes a less academic approach to the matter. While almost all algorithms are well described in literature, the implementation still is left to hardware designers, which brings issues about especially having different data formats in the same architecture.

2.4.1 Specifications

The unit must execute correctly all the RV32-M instructions, so each operation must be capable to handle both signed and unsigned numbers, or even mixed types.

Official RISC-V Foundation specifications also explicitly ask that in case of :

- Consecutive *MULTH[S][U]* and *MULT*
- Consecutive *DIV[U]* and *REM[U]*

Where *rs1* and *rs2* sections are equal, the unit must immediately provide the correct result without performing again the same computation. Lastly *division by zero* and *division overflow* must rise two different exceptions. It's important to notice that the first case is the only arithmetic trap in the standard ISA, for which we are not providing any interaction with the execution environment's trap handlers. Anyway division by zero must cause an immediate control flow change and only a single branch instruction can be inserted before every division one, which will be very predictably not taken.

Condition	Dividend	Divisor	DIVU[W]	REMU[W]	DIV[W]	REM[W]
Division by zero	x	0	$2^L - 1$	x	-1	x
Overflow (signed only)	-2^{L-1}	-1	–	–	-2^{L-1}	0

Table 7.1: Semantics for division by zero and division overflow. L is the width of the operation in bits: XLEN for DIV[U] and REM[U], or 32 for DIV[U]W and REM[U]W.

Figure 2.3: Special cases for division

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

RV64M Standard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	MULW
0000001	rs2	rs1	100	rd	0111011	DIVW
0000001	rs2	rs1	101	rd	0111011	DIVUW
0000001	rs2	rs1	110	rd	0111011	REMW
0000001	rs2	rs1	111	rd	0111011	REMUW

Figure 2.4: RV32M and RV64M instructions

Algorithms chosen

After a first research on how modern microprocessors perform multiplications and divisions, the most common solution was found to be a shared microprogrammed architecture based on sequential algorithms: *Booth encoded multiplication* and *Sweeney-Robertson-Tocher division*. A short description of both can be found on the following sections.
This is mainly for three motivations:

- Both algorithm can share the same basic hardware blocks.
- Working on sequential architectures opens to several possible improvements on the critical path, avoiding any frequency issue inside machines that tends to achieve shorter and shorter periods. Also low-power techniques can be easily implemented during design time. Since both operations share the same architecture any upgrade done for one may be applicable also for the other.
- Algorithms can be modified to elaborate more steps in a single clock cycle with very low timing overhead. Those algorithms are addressed as *higher radix*.

The proposed implementation focuses just on the first two points, while was chosen to keep the simple radix-2 algorithms because higher ones would lead to longer design time. But once an eventual high-radix algorithm is set up for the specific case, few modifications on the radix-2 architecture may be needed.

The use of carry save adder

The kernel of the recipes for multiplication and division is made of additions and shifts repeated for a certain amount of cycles. The first improvement suggested by sources on computer arithmetic was to use a carry save adder and store the result of each iteration in the sum-carry form.
Partial reminder and partial product will be taken as just said. Then final result has to be computed by a "normal" sum, for which any kind of adder can be used, except for the one used in the kernel cycles.
While for the multiplication introducing this amendment is not a big deal, for division brings some issues, since the operations that must be performed in each step of it depends on the value of the partial reminder at the previous one. The point is that there is no way to know it, when it is expressed in the sum-carry form, without performing a sum of the two parts. This would cancel all the efforts.
What can be done is to guess the partial reminder looking just at the first 5 bits of sum and carry and then implement some mechanism that corrects the result in case of one or more mistakes done during the elaboration.

In few words, performing some approximate computations the divisor architecture is capable to produce at the end an exact result.

2.4.2 Multiplication

Multiplication can be considered much easier to handle then division. As previously said, the carry save adder choice doesn't affect much the design.

Theory explanation

Architecture is based on the *right shift* algorithm, in which for each step a partial product is computed by multiplying the multiplicand by a bit of the multiplier starting from the rightmost one, then it is summed to a partial result obtain from all the previous steps and finally, the number got, is divided by 2:

$$p^{(j+1)} = (p^{(j)} + 2^k * x_j a)2^{-1} \tag{2.1}$$

Where:

- $p^{(j)}$ is the partial result at the generic step j.
- x_j is the bit of the multiplier at j position starting counting from the lsb.
- k is the number of bits. In this case 32.
- a is the multiplicand.
- $p^{(0)}$ is equal to **0**.
- $p^{(k)}$ is the product of the multiplication.

Booth encoding Partial products' value may be equal to **a** if the corresponding multiplier bit is equal to 1 or **0** in the other case. Performing a multiplication where most control bit are zeros requires less effort than in the opposite possibility. Booth exploited this simple tip re coding multiplier's bit. This operation consists in replacing a sequence of 1s, all additions, with a subtraction at the least-significant end an addition in the position immediately to the left of its most-significant end. As an example

011110 will become *1000-10*

It's possible to obtain from x_j bit a new y_j re coded one as follows:

x_j	x_{j-1}	y_j	Explanation
0	0	0	No string
0	1	1	End of string
1	0	-1	Beginning of string
1	1	0	Continuation of string

Table 2.1: Booth re coding table

With this re coding we obtain another important feature: now the multiplier can also handle signed operands, while normal right shift one couldn't. In this way is also possible to satisfy

Model

The algorithm for the Booth re coded multiplication can be modeled with the following python code

```
BE_multiplier (multiplicand,multiplier,signed_unsigned_n,parallelism):
    #converting multiplier in it's equivalent string of bits
    mult_string=printer_2s(multiplier,parallelism)
    a=multiplicand*(2**(parallelism-1)) #a*2^k
    #first case in which there is not x(j-1)
    if (int(mult_string[parallelism-1])): #checking LSB
        p=-a
    else:
        p=0
    p=p/2
    for i in range(0,parallelism-1):
        if mult_string[parallelism-i-2]=='0' and mult_string[parallelism-i-1]=='0':
            p=p
        elif mult_string[parallelism-i-2]=='0' and mult_string[parallelism-i-1]=='1':
            p=p+a
        elif mult_string[parallelism-i-2]=='1' and mult_string[parallelism-i-1]=='0':
            p=p-a
        else:
            p=p
        p=p/2
    p=p*2
    return p
```

DataPath

For the realization of the multiplier's datapath only simple basic blocks are needed.

Kernel Logic The logic for the selection of the generic partial product is very simple. Two signals are needed:

- NonZero: obtained with an XOR gate between x_j and x_{j-1}
- Negative: obtained with a direct connection with x_j

It's important to notice that first partial product is directly loaded with the multiplier and it's chosen using its least significant bit.

Final adder The final adder is not implemented. Synthesizer will choose a model of adder and place it after synthesis. This will remain true also for the divisor.

Counter For reasons of space the counter is omitted in the schematic. A signed multiplication would require only 32 steps, while an unsigned one takes always 33 steps because the case in which 32^{nd} bit, most significant bit, is 1 must be taken in account and a 0 must be put at the left end of the number to correctly handle the operation.
In order to simplify control and the finite state machine, the machine always performs 33 steps which will always give the right result. A clock cycle is lost for every signed multiplication.
Considering that the first step is automatically done while loading the multiplier in the *sumH* register, the counter must count only 32 multiplication steps. Latency is always the same.

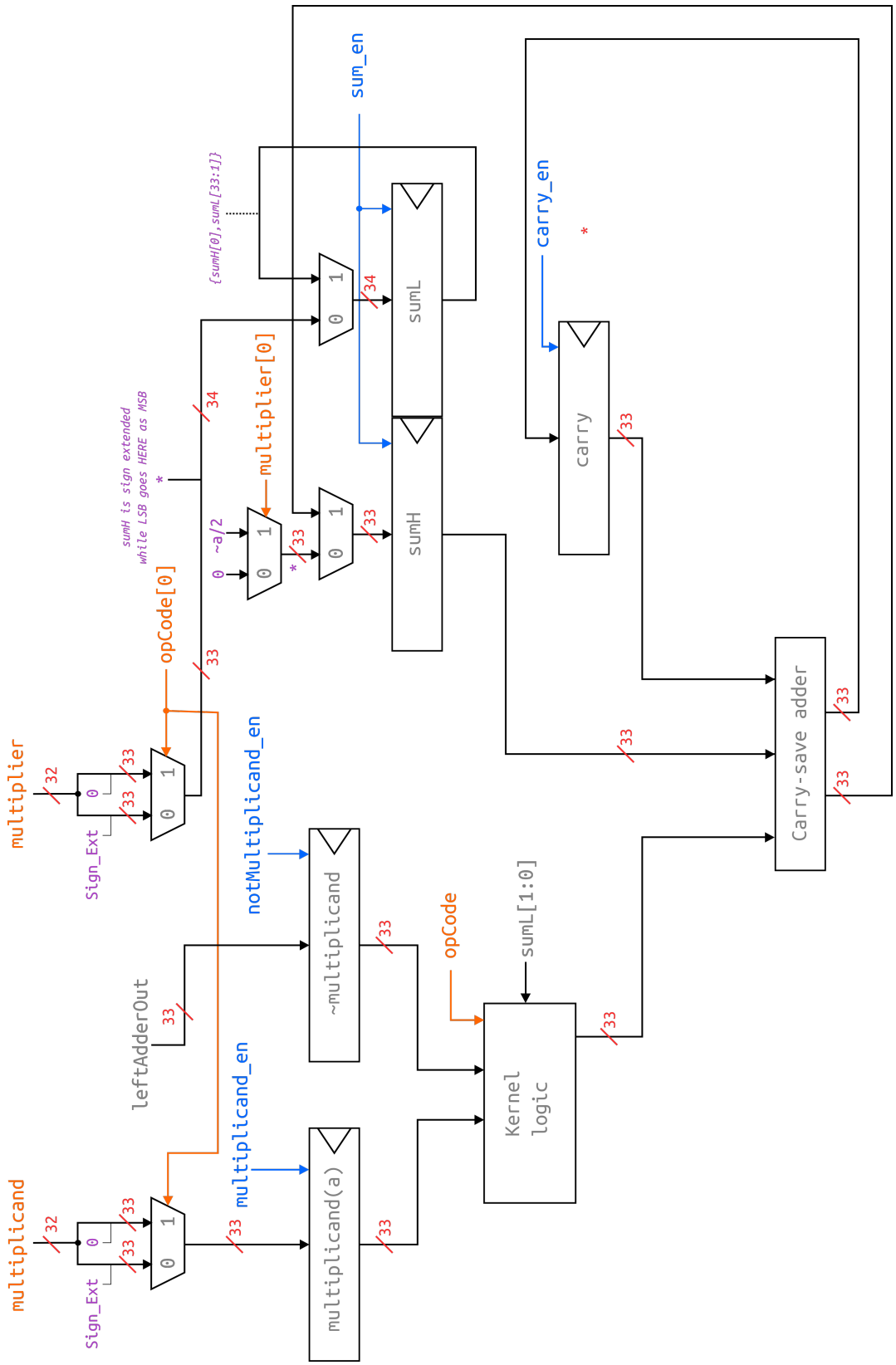

Figure 2.5: Multiplier upper datapath

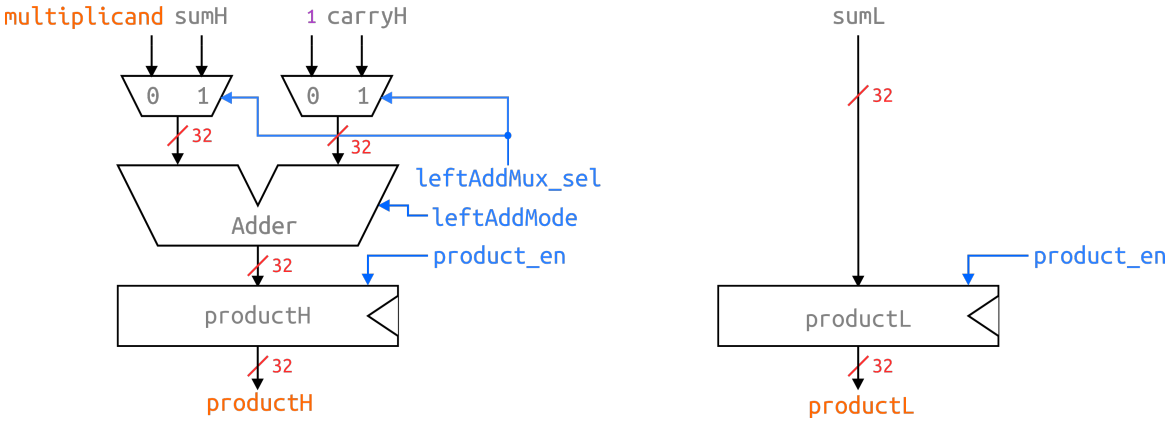

Figure 2.6: Multiplier lower datapath

Finite state machine

Few controls for the whole multiplication are needed.

***idle* State** The machine waits in the idle state until a specific signal is raised by the outer world, communicating that operands are valid and need to be evaluated. The only operation performed is a synchronous clear of the sum and carry registers. This is because an asynchronous reset signal controlled by a finite state machine may lead to problems in the post synthesis circuit, since spurious glitches in the output logic of the commands may reset the entire circuit when unwanted.

***save_multiplicand* State** Only the multiplicand register is enable, before its negative value must be computed before loading the multiplier.

***save_multiplier* State** The multiplexers at the input of the adder in the lower part of the data path must be set correctly in order to compute the *negative multiplicand* $(-a)$. Least significant bit of multiplier will select the correct partial product, that will be loaded with the multiplier in the *sum* registers. Computing *-a* isn't essential for the purposes of the algorithm. In fact it's possible to add the bit-wise not of the multiplicand to the partial result when required, and store the +1 bit for making the 2's complement in another register. Then at the end, all these +1 bits, in the appropriate weight, can be summed together to the final result, saving one state in the finite state machine.
Since for the division is mandatory to compute the negative value of the divisor, the same is done for the multiplication, simplifying a bit the design.

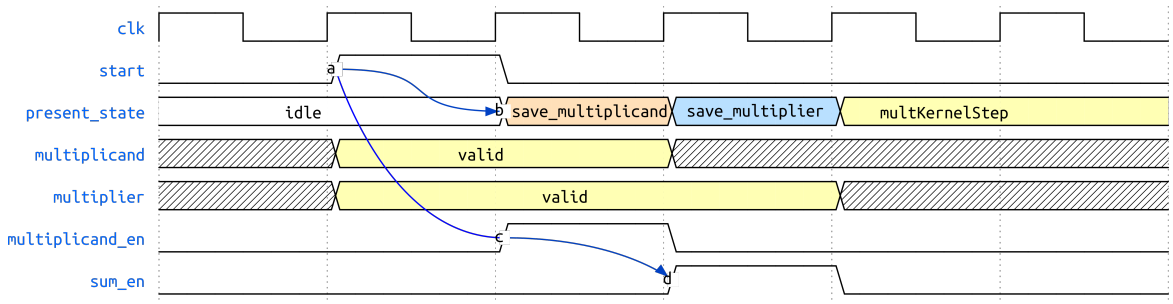

Figure 2.7: Timing of the starting operation

Notice that multiplier must remain valid for at least **three clock cycles** after the rising edge of the *start signal*.

***multKernelStep* State** After first step is performed, 32 always equal are left. For each one of these the correct partial product must be selected and added to the partial result as explained before. The division by 2 is automatically done storing the output of the carry save adder shifted to the right, the least significant bit of the *sum* goes in the most significant one of the register that initially was holding the multiplier, which is also shifted in order to update control bits for the choice of the next partial product.

***save_product* State** Final product will be made up by two words of 32 bits. The most significant one is expressed in the sum-carry form and must be computed by summing the two parts, while the least significant is already ready and stored in the *sumL* register in the range that goes from 33^{rd} to the 2^{nd} bit.

***multDone* State** End of computation is signaled to the outer world.

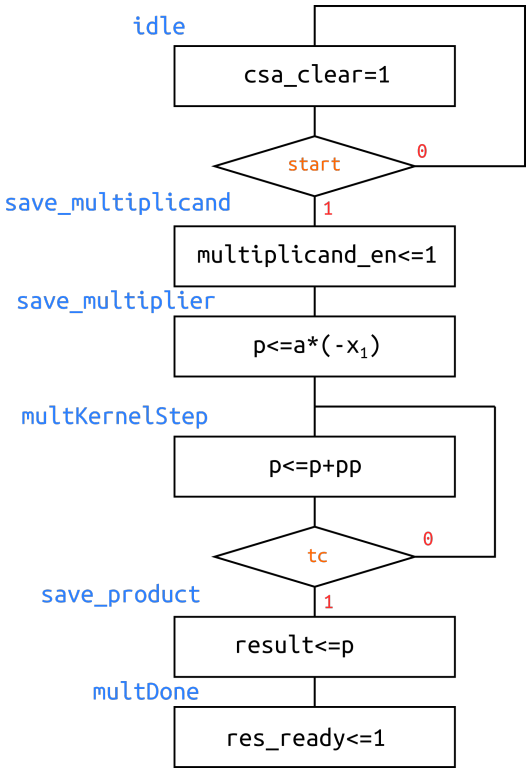

Figure 2.8: Multiplier ASM

	idle	save_muliplicand	save_mulitplier	multKernelStep	save_product	multDone
csa_clear	1	0	0	0	0	0
multiplicand_en	0	1	0	0	0	0
notMultiplicand_en	0	0	1	0	0	0
sumMux_sel	0	0	0	1	0	0
sum_en	0	0	1	1	0	0
carry_en	0	0	0	1	0	0
leftAddMux_sel	0	0	0	0	1	0
count_en	0	0	0	1	0	0
prod_en	0	0	0	0	1	0
res_ready	0	0	0	0	0	1

Figure 2.9: Multiplier control signals

2.4.3 Division

Very complex and tricky was the design of the divisor. There are three main motivations that contributed to make intricate the realization of a division machine:

- available algorithms are not general. Re conducing all possible cases to the specific ones described in literature it's not trivial. What has to be realized is a machine that can handle two operands, signed or unsigned, of 32 bits, but with unknown actual dynamic (e.g. how many bits are sign extension). SRT division is described only for dividend of twice the number of bits respect to the divisor, which must also respect some strict conditions.
- Design, at the beginning, is base on some considerations that may be wrong, but can't be verified at priory. Those assumptions introduce some bug inside a partially working machine that are hard to detect or understand. The only way is to analyze bit by bit the internal behavior, which is a very stress full job, error prone and takes most of the design time.
- all number are expressed in fixed point format, but due to different type of operations done in the division kernel, the number of bits assigned for the integer and the fraction may vary across the architecture in order to avoid overflows and unused resources. For example 2.30 format can be extended to 3.30 which is totally different from 2.31, even if they are bot 33 bits data. Or also 2.30 can be truncated to 2.29 if lsb is not needed, but must be handled assigning the right weight to every bit.
- some tasks in the algorithm can be *"commutative"*, the order doesn't count for the final result, but can make the design much more complex.

Theory explanation

Explaining all the theory that lays under the division algorithm is long and complex, this will be a very short recap. For more information please consult the bibliography.

Basic algorithm The starting point for the model of the final divisor is the *non restoring and signed division.*
The main procedure lays on the these hypothesis:

- Dividend can be expressed in $2k$ bits while the divisor in half of the width, just k bits.

- The value obtained by multiplying the divisor by 2^k must be strictly less than the dividend. If this condition isn't true an overflow can verify at the first step of the algorithm.

We can generally say that

$$z = qd + s \tag{2.2}$$

Where **z** is the dividend, **q** is the quotient, **d** is the divisor and **s** is the reminder.
The operation done in one of the basic steps j out of 32 can be then expressed as

$$s^{(j)} = 2s^{(j-1)} - q_{k-j} * 2^k d \tag{2.3}$$

which is the same as for the restoring algorithm. What changes is that in case the subtraction performed leads to a sign change in the partial reminder, no restore and change of the value of the quotient bit is performed, but in the following step an addition is applied. The effect of this trick can be easily demonstrate.

$$s^{(j+1)} = 2s^{(j)} + 2^k d \tag{2.4}$$

Substitution of $s^{(j)}$ leads to:

$$s^{(j+1)} = 4s^{(j-1)} - 2 * 2^k d + 2^k d \tag{2.5}$$

The only thing that must be done is to keep track of what operation has been performed for each step in the quotient.
If $sign(s^j) = sign(d)$, q_{k-j} is set to *1*. In the opposite case it is set to *-1*.
At the end, it's possible to obtain two numbers, the one made of *1* and the other made of *-1*. To get the quotient is enough subtracting the twos.
Before producing the result is mandatory to check if the reminder is coherent with the dividend, if it's not a correction step must be provided to recover the mistake. If sign(s^k)=sign(d) it means that d must be subtracted to s^k and quotient incremented, if it's not an addition is needed and quotient is decremented.

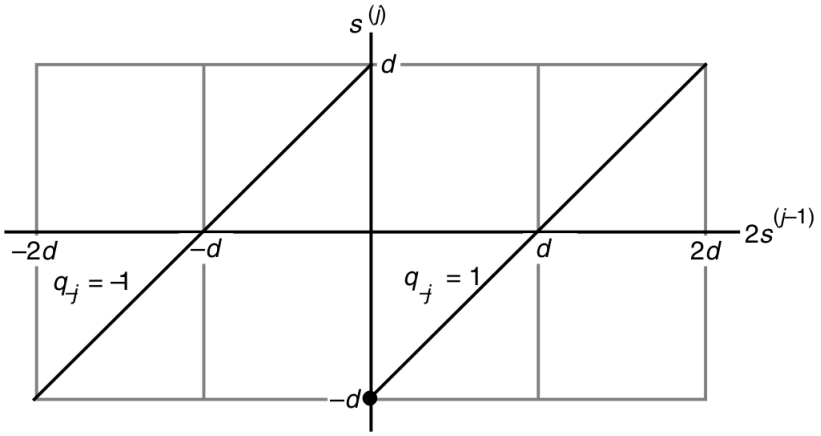

Figure 2.10: Ranges for q_{k-j} choice in the set {-1,1}

SRT division The *Sweeney-Robertson-Tocher division* is based on the idea that it's possible to avoid any addition or subtraction in a normal kernel step if the partial reminder is enough "small". If it is in the range $[-d, d)$, multiplying it by two, it's possible to choose q_{k-j} equal to 0.

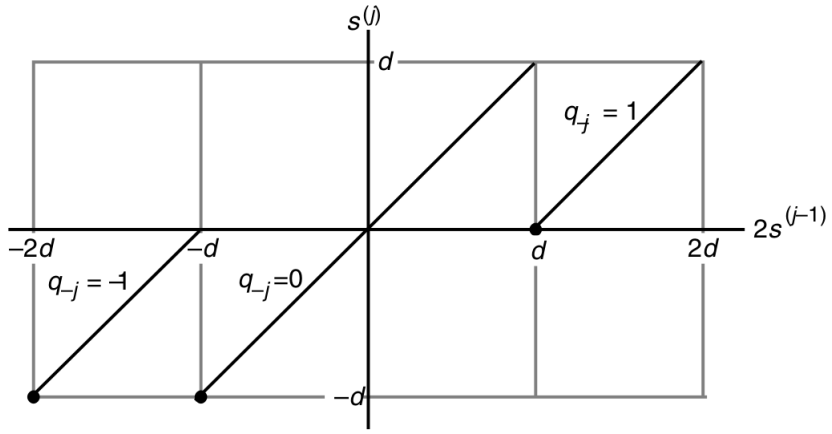

Figure 2.11: Ranges for q_{k-j} choice in the set {-1,0,1}

The problem of this concept is that is not possible to compare the partial reminder with the divisor without subtracting them and then analyze the result with extra logic. This would slow down dramatically the architecture.
The only way to undo the knot is to choose easy fixed thresholds for the comparison, it's possible to achieve this by expressing the operands in fixed point 1.31. Then two conditions must hold:

- $d \geq 0.5$, if it's not true divisor must be shifted left. The number of shifts will become the number of kernel step that must be performed. It's important to remember that the reminder must be corrected at the end by shifting it right the same amount of time.
- $-d \leq z < d$, if it's not true dividend must be shifted right once, then result must also be realigned at the end. In the design z is always divided by 2, so that this condition is always true.

Two comparison are still needed to select the correct quotient digit, but now they can be done with two constants: -0.5 and 0.5. This implies just a logic gate layer.

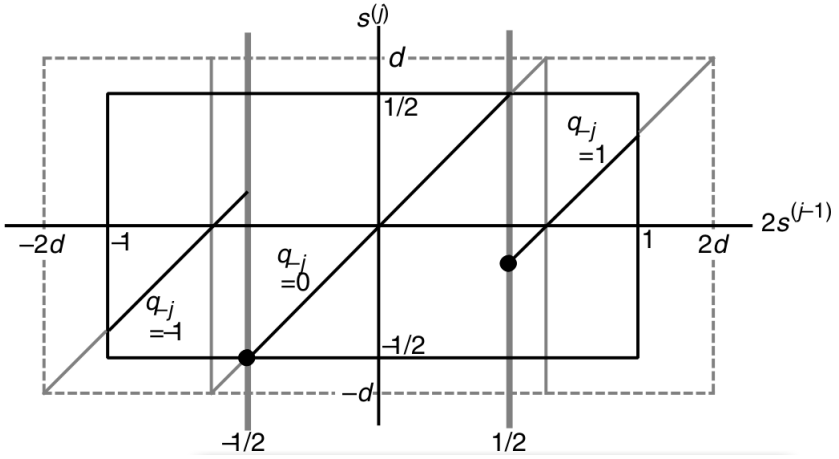

Figure 2.12: Ranges for q_{k-j} choice in the set {-1,0,1}, with new thresholds

Carry Save adder Carry save adder, as for the multiplication, speeds up the computation done each cycle.
But imagine if partial reminders must be stored in the sum-carry form. The only way to determine

it's actual is to compute it. This would null all the advantages obtained introducing the carry save adder. What can be done is to sum just the most significant bits up to second decimal place and modify thresholds to handle the case in which the actual number falls outside the range with an approximate computation. The sum can be implemented with a look-up table or a PLA in order to in crease the access speed.
In literature, it's written that only two guard bits are needed to avoid overflows after the sum, but what was found during the design is that there are cases, one over two thousand, in which one more is needed for the carry register. So in the schematic 5 bits are used for the quotient bit selection.

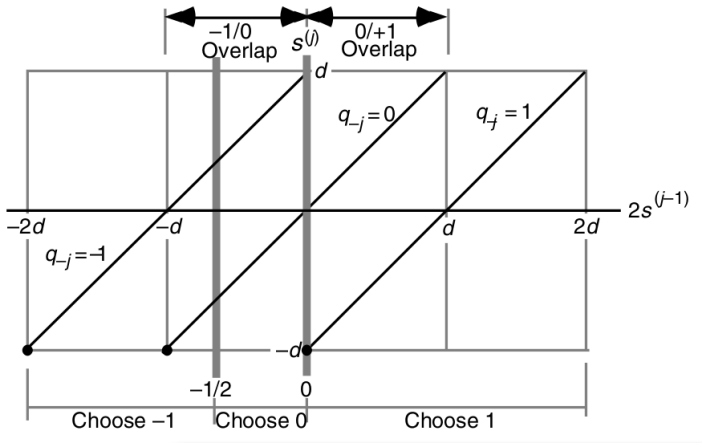

Figure 2.13: Ranges for q_{k-j} choice in the set {-1,0,1} with overlapping regions

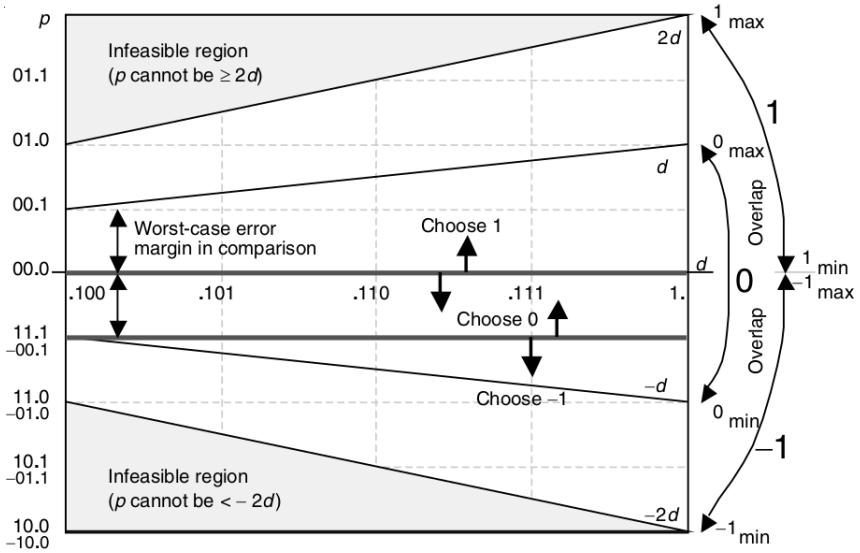

Figure 2.14: p-d plot for the final design

Model

All the steps described in the previous part are resumed in the following python code, which is as close as possible to what the hardware machine really do.

```
#thr1=0 and thr2=-0.5 in this case
def SRTr2_divisor(dividend,divisor,inPar,thr1,thr2):
```

```
#division by zero case
if (divisor==0):
    return "divisorIs0"
#fixed point format 1.33
#division inPar+1 for both signed and unsigned case, -d<=z<d always true
z=dividend/2**(inPar+1)
d=divisor/2**(inPar+1)
#first step s(0)=2*z
s=z*2
#shifting left divisor until more than 0.5
loop_iteration=0
while ((d<0.5 and d>0) or (d>=-0.5 and d<0)):
    loop_iteration+=1
    d=d*2
#standard kernel iterations
q=""
for i in range(0,loop_iteration):
    if(s>=thr2):
        _q=1
        if (d>0):
            s=s-d
        else:
            s=s+d
    else:
        if (s>=thr1):
            _q=0
            s=s
        else :
            _q=2
            if (d>0):
                s=s+d
            else:
                s=s-d
    s=s*2
    q=q+str(_q)
#last step doesn't require *2, so a correction is needed
s=s/2
#quotient conversion
quotient=0
for i in range(0,loop_iteration):
    if (q[len(q)-1-i]=="1"):
        quotient=quotient+2**i
    else :
        if (q[len(q)-1-i]=="2"):
            quotient=quotient-2**i
        else:
            quotient=quotient
#sign of q correction
#we need to change sign before correction!
if (divisor<0):
    quotient=-quotient
#last step correction
if(((s>0) and (z<0)) or ((s<0) and (z>0))):
    if(((s>0) and (d<0)) or ((s<0) and (d>0))):
        s=s+d
        quotient=quotient-1
    else:
        s=s-d
        quotient=quotient+1
#final correction
s=s*(2**(inPar+1-loop_iteration))
res=[]
```

```
res+=[quotient]
res+=[s]
return res
```

Datapath

In the schematic proposed for each wire is indicated the parallelism and where significant also the fixed format digits. As for the multiplier all the blocks are very simple ones. Only the kernel logic require some attention.

Kernel Logic The kernel logic consist in block that given 5 most significant bit of the sum and the 5 of the carry produces an exact sum. As already said, this can be implemented with a 10 bit entry look-up table or PLA. Depending on the output of the first stage *SignSel* and *Non0* signals are driven, and they will determine next quotient bit.

Sum	SignSel	Non0	Data to be summed
$(-\infty, -0.5)$	1	1	$-d * sign(d)$
$[-0.5, 0)$	-	0	0
$[0, +\infty)$	0	1	$d * sign(d)$

Table 2.2: Division kernel logic output table

Reminder shift register The reminder shift register is actually made of 34 bits, because it's already loaded divided by two, if the machine shifts it as many time as the divisor was shifted, it will result wrong. The solution was found in saving it with a 0 in the least significant bit so that taking first 32 bits of the register there won't be any mistakes.

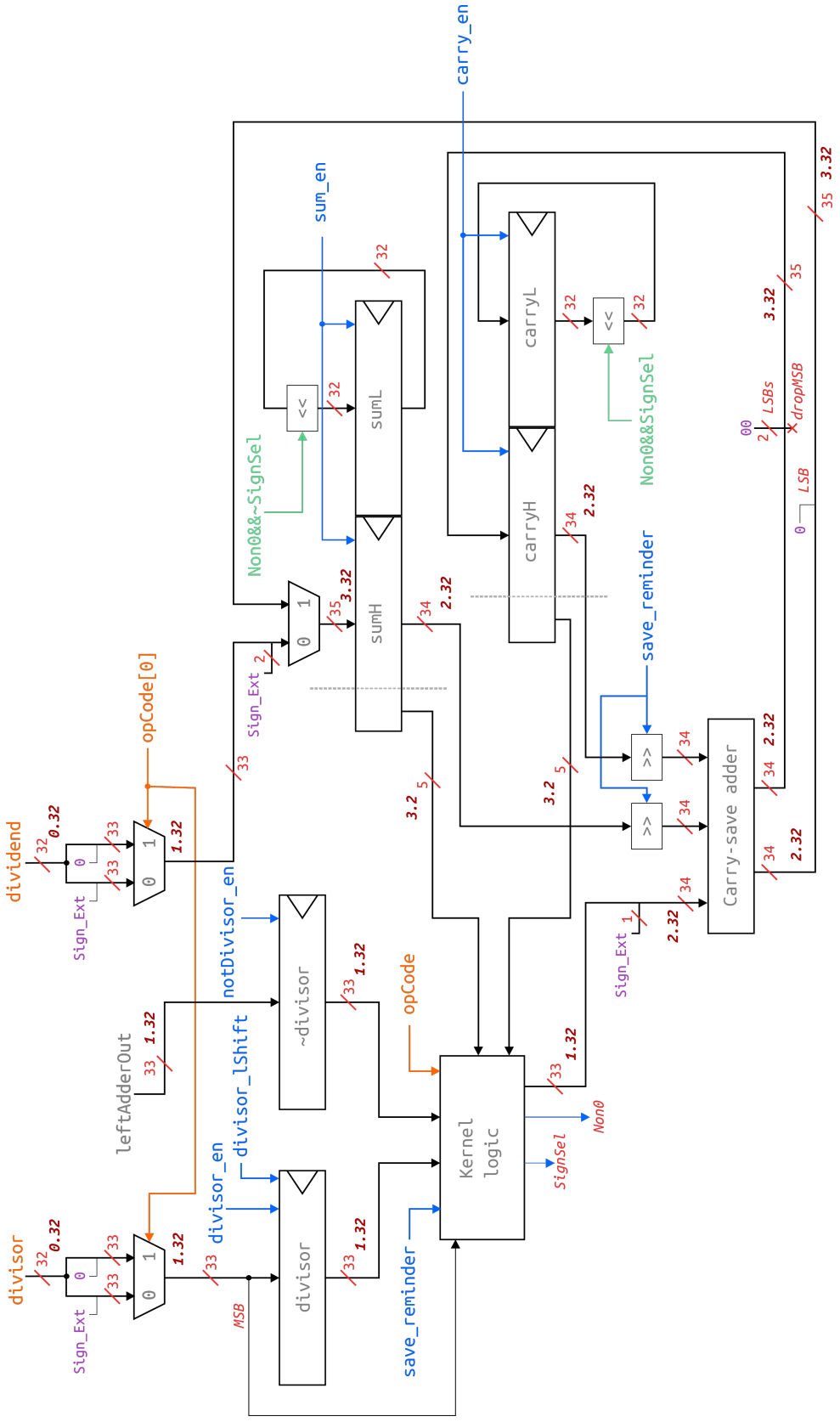

Figure 2.15: Divisor upper datapath

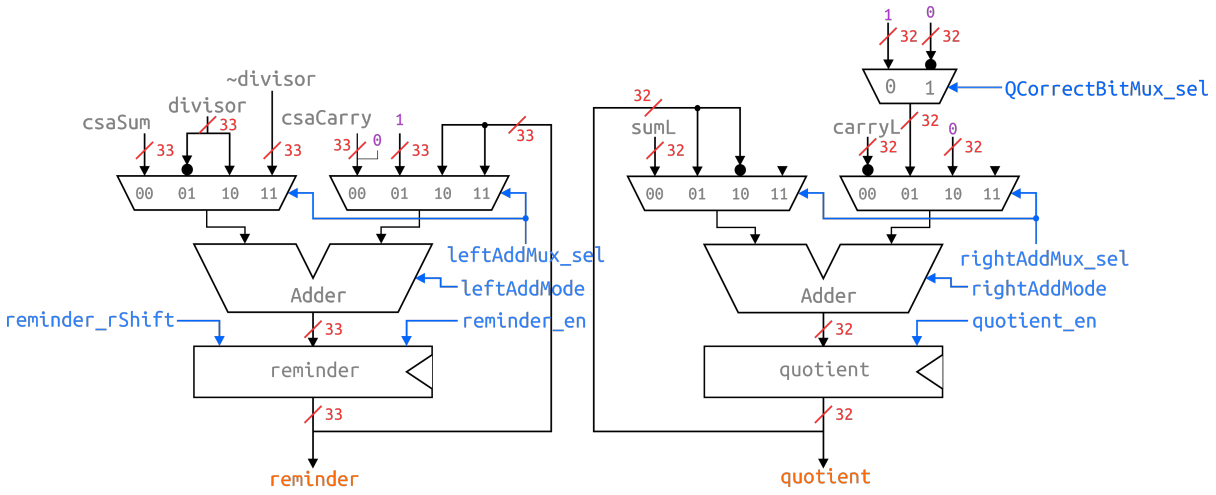

Figure 2.16: Divisor lower datapath

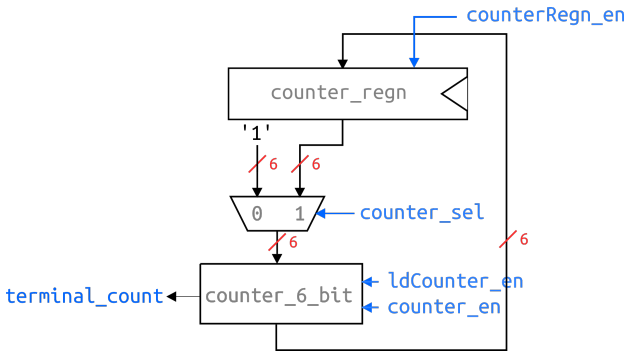

Figure 2.17: Divisor counter

Finite State Machine

Here is done a fast description of what is done in each state of the FSM. For what concerns the *idle* state and the start timing please check the multiplier section.

***idle* State** The machine waits for a rising edge on the start signal.

***loadData* State** Operands are loaded into the appropriate registers multiplied by 2. Divisor is checked on the fly to understand if it needs left shifts. In case it's ready, *1* is loaded in the counter, since we need to perform just one step. This never verifies in case of signed division because 33^{rd} bit will always be a sign extension.

***divisorLShift* State** Divisor is shifted and then a change of trend is checked by xoring 32^{nd} and 31^{st}, since the shift will be updated in the successive cycle. The counter is enabled for counting how many kernel step must be performed, starting from 1 because divisor was loaded multiplied by 2.

***saveIterloop* State** Once the divisor is properly aligned it's important to keep track of how many multiplications by 2 have been done. Counter output is stored inside a dedicated feedback-register, the value will be read at the end few state later.

***divKernelStep* State** The main operation consists of a sum between the number produced by the proper logic and the previous partial reminder, expressed in sum-carry form. The result is automatically multiplied by two and feeds the registers in upper data path.
The two registers dedicated to the quotient also sample the new bit.

***computeQ* State** From now on only the two output adder are going to be used. Firstly they are set to compute a early version of the quotient and the reminder, the results will be then stored in the output registers used as accumulators. In this state the counter is re loaded with the previously store numeber of total shifts.

***waitSignals* State** Since the logic for the new state selection is driven from the output of the output, a clock cycle must be lost to let the signals propagate across the circuit.

***qInv* State** Before going ahead, the sign of the quotient must be corrected. If the divisor is negative we must switch the sign performing the 2's complement of it's value.

***correctDown* and *correctUp* State** These two states correct the results when needed.

***RemCorrectio* State** Lastly it's needed to shift right the reminder for a number of time that was stored in the counter in the *computeQ* state. It must be noticed that the reminder is loaded already divided by 2 so, one step less is needed. For this purpose the shift register is made of 34 bits so that the actual least significant bit is not drop during last step.

***divDone* State** End of computation is signaled to the outer world.

	idle	loadData	divisorLShift	saveIterLoop	divKernelStep	computeQ	waitSignals	correctDown	correctUp	qInv	remCorrection	divDone
csa_clear	1	0	0	0	0	0	0	0	0	0	0	0
divisor_en	0	1	0	0	0	0	0	0	0	0	0	0
divisor_lShift	0	0	1	0	0	0	0	0	0	0	0	0
notDivisor_en	0	0	0	1	0	0	0	0	0	0	0	0
saveReminder	0	1	0	0	0	1	0	0	0	0	0	0
sumHMux_sel	0	0	0	0	1	0	0	0	0	0	0	0
sum_en	0	1	0	0	1	0	0	0	0	0	0	0
carry_en	0	0	0	0	1	0	0	0	0	0	0	0
leftAddMux_sel	0 0	0 0	0 0	0 1	0 0	0 0	0 0	1 0	1 1	0 0	0 0	0 0
rightAddMux_sel	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 1	0 1	1 0	0 0	0 0
QCorrectBitMux_sel	0	0	0	0	0	0	0	1	0	0	0	0
leftAddMode	0	0	0	1	0	0	0	0	0	0	0	0
rightAddMode	0	0	0	0	0	1	0	0	0	1	0	0
reminder_en	0	0	0	0	0	1	0	1	1	0	0	0
reminder_rShift	0	0	0	0	0	0	0	0	0	0	1	0
quotient_en	0	0	0	0	0	1	0	1	1	1	0	0
counterMux_sel	0	0	0	0	0	1	0	0	0	0	0	0
count_upDown	0	0	1	0	0	0	0	0	0	0	0	0
count_load	0	1	0	0	0	1	0	0	0	0	0	0
count_en	0	0	1	0	1	0	0	0	0	0	1	0
counterReg_en	0	0	0	1	0	0	0	0	0	0	0	0
done	0	0	0	0	0	0	0	0	0	0	0	1

Figure 2.18: Divisor Control State Table

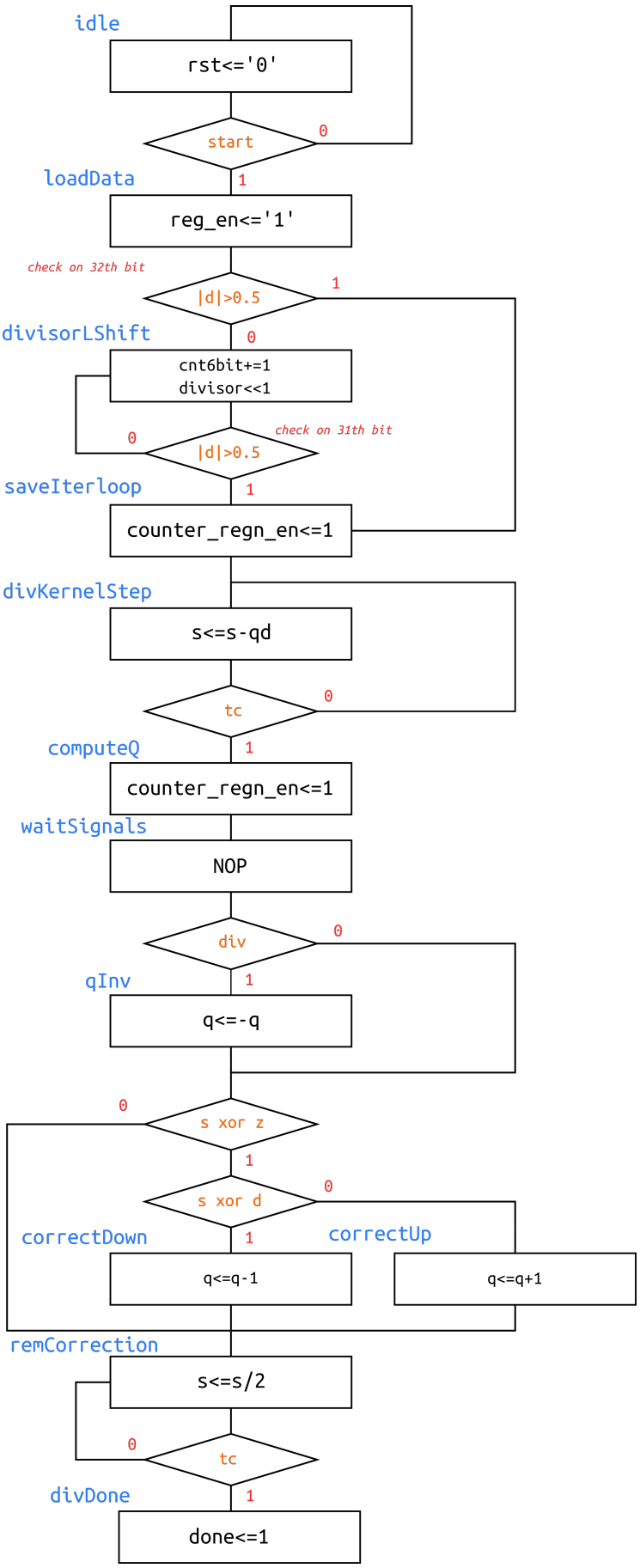

Figure 2.19: Divisor ASM

2.4.4 Complete multiplication and division unit

After the study of the single multiplication and division, it was not difficult to integrate everything in a single architecture.
Main difference is represented by a conditions checking circuit in which the following special case are tested:

- Two consecutive instructions require the same computation, which is avoided and the result is directly given at the output.
- In case of division, divisor value is 0.
- In case of signed division, divisor value is -1 and dividend one is -2^{k-1}, so that the resulting quotient will overflow the capability of 32 bits.

Datapath

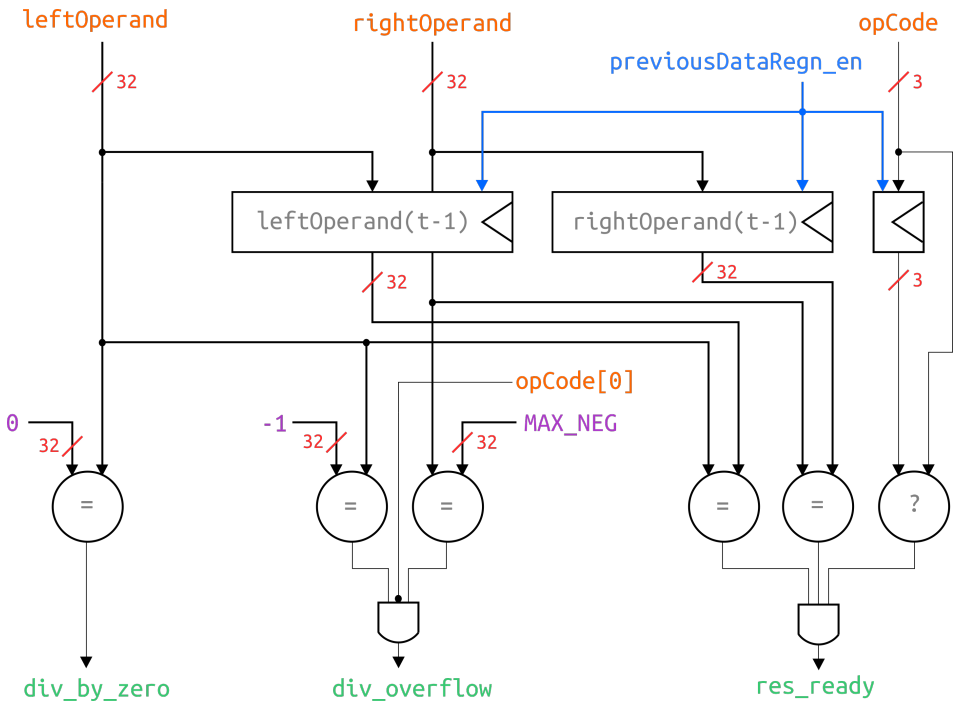

Figure 2.20: Control net for special cases

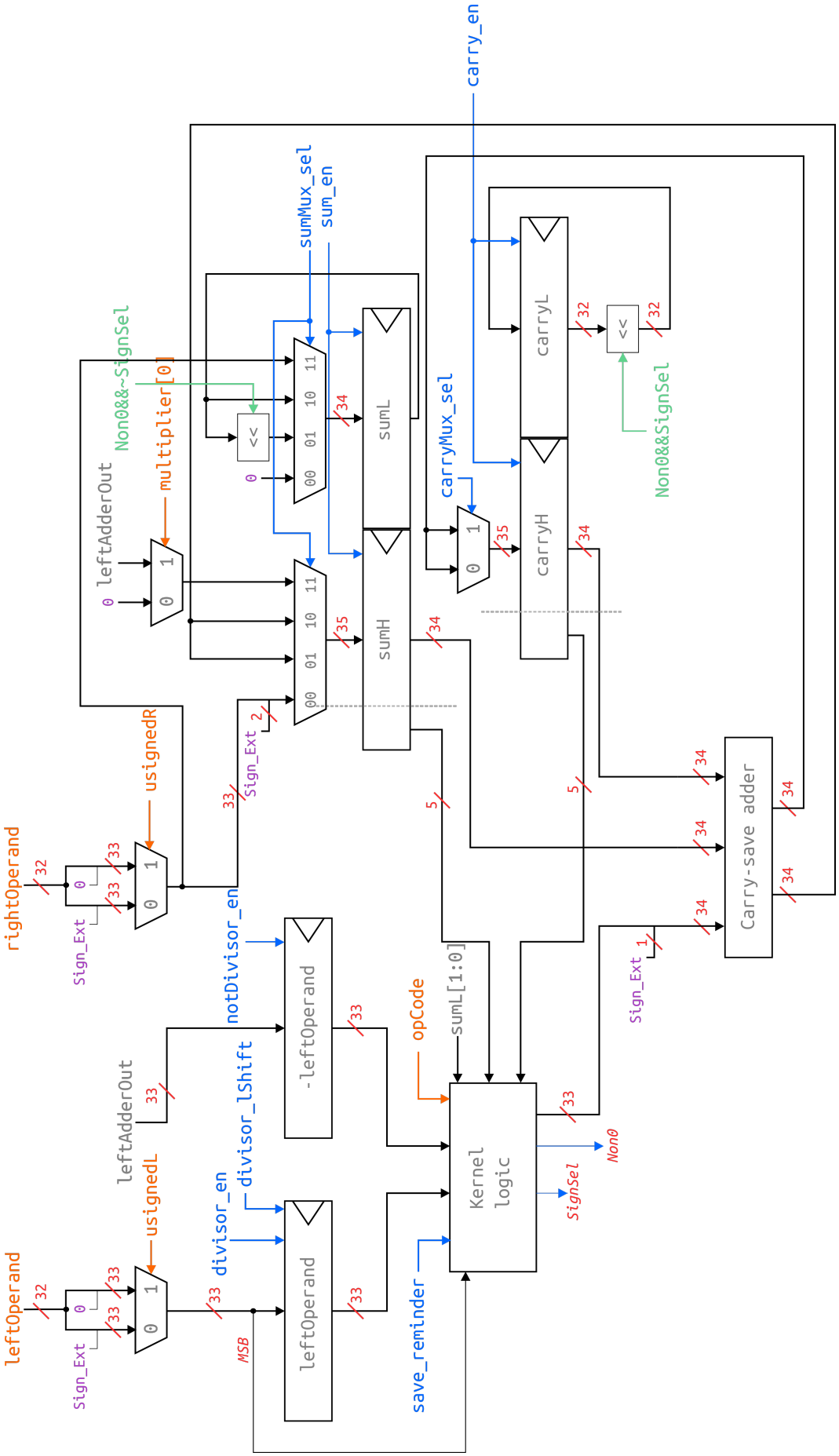

Figure 2.21: Upper data path

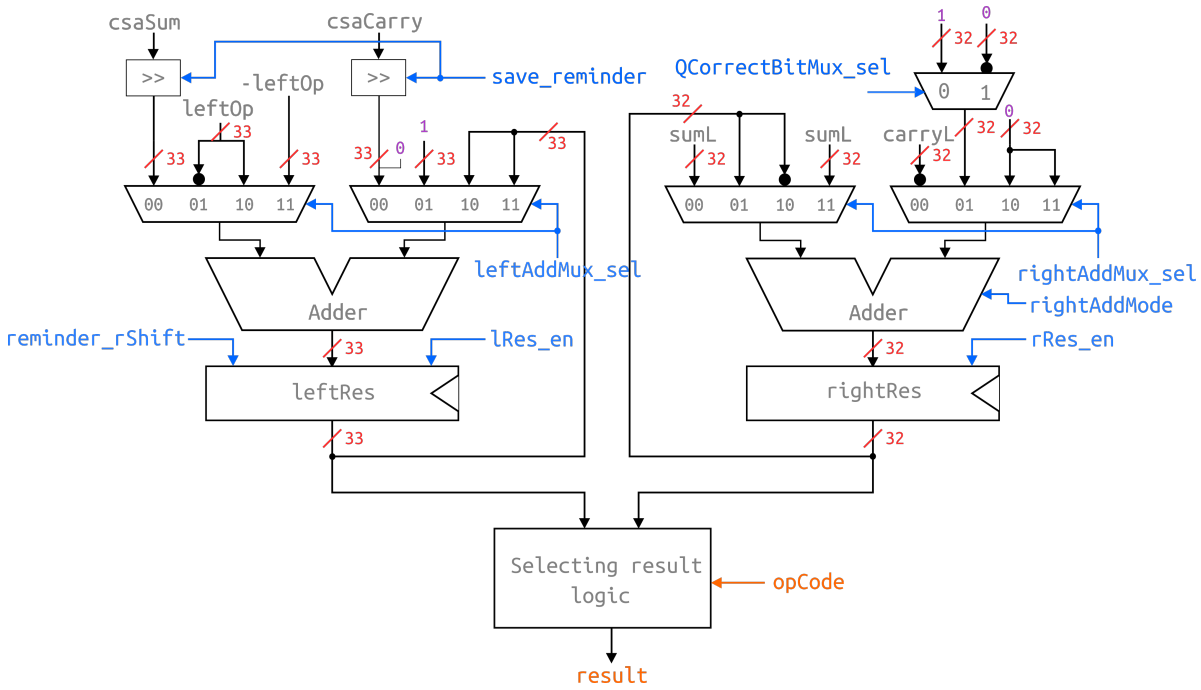

Figure 2.22: Lower data path

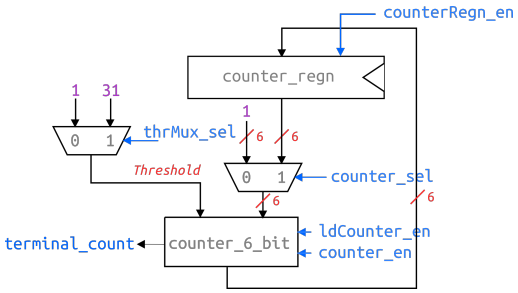

Figure 2.23: Counter and related controls

Finite State Machine

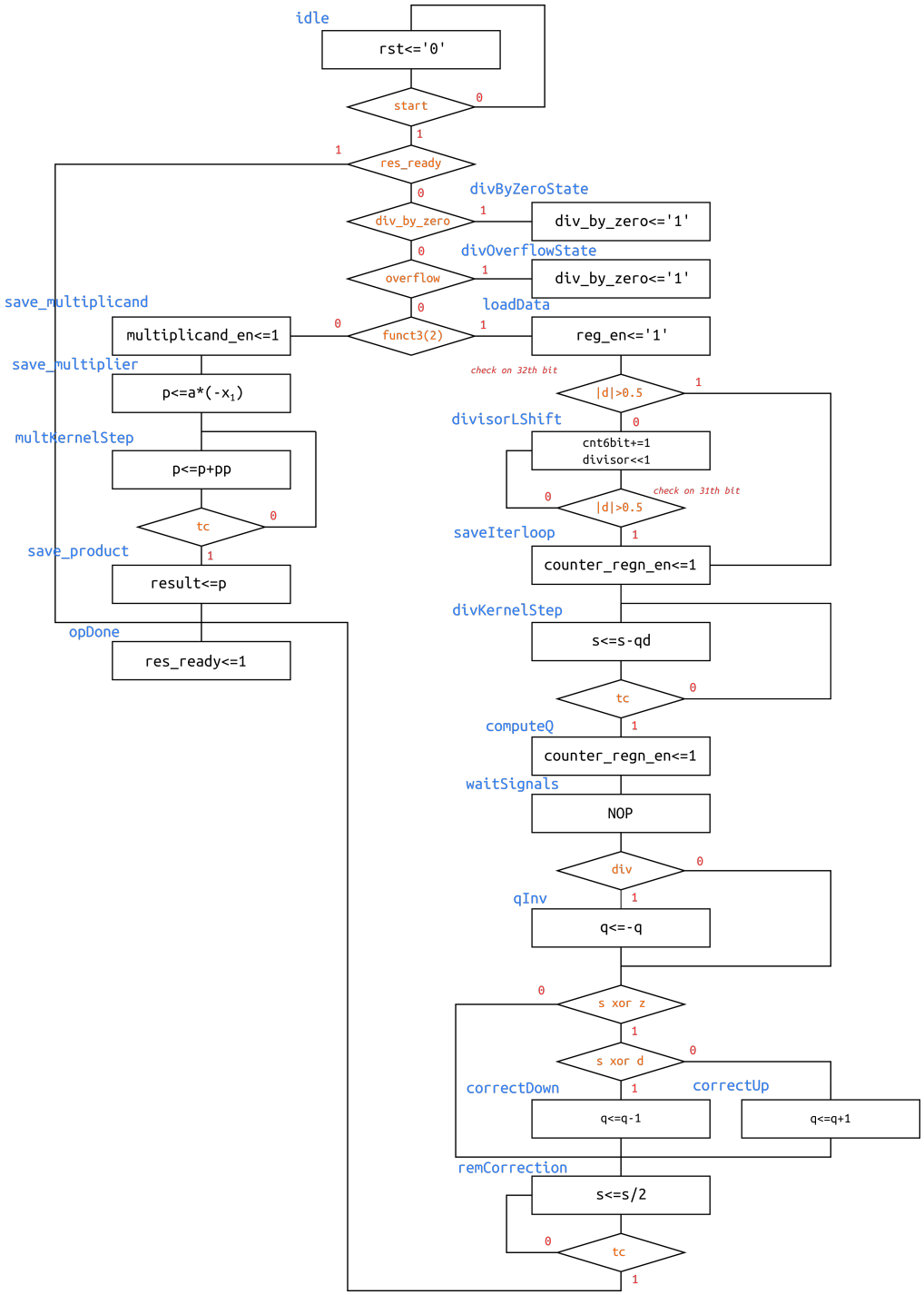

Figure 2.24: Complete machine ASM

2.5 Memory (MEM)

The Memory stage of our RISCV Core is mainly composed of the DRAM Controller that handles the DRAM.

We decided to keep things simple in this stage, without replicating the cache structure we adopted in the Fetch Unit, so we opted for a basic combinatory unit that is supposed to write/read into/from the DRAM in the same clock cycle in which the operation is required. The Data RAM, on the other hand, has not been introduced inside the processor core since we assumed it to be an outer (and non synthetizable) module. The controller is capable of correctly handling the sign extension of the operand retrieved from memory in case we've a signed load instruction *lb* or *lh* which load a signed byte or half-word respectively from memory or an unsigned load instruction *lbu* or *lhu*.
An improved version of our core with respect to this stage may implement a similar version of the Instruction Cache, with a *ready* signal to tell the CU when to stall the core in case of a miss; it will also need to handle the data coherency with the main RAM since the content of the two memories may not be the same at any given time instant.

2.6 Write Back (WB)

The Write Back stage is not explicitly reported on the RISCV Core Module and it just consists of the wr_en signal, the wr_addr and wr_data fields that stem from the pipeline registers and go into the related Register File ports in order to allow the writing of the result of the instruction completed.

CHAPTER 3

Non-Datapath Logic

In this section a more detailed explanation about the sub-modules not belonging to the Datapath will be given. In particular, these blocks belong to the Fetch Unit and are the *BranchPredictionUnit*, *BranchForwardingUnit* and the *InstructionCache* with its *Controller*.

3.1 Branch Prediction Unit

The BPU is a module in charge of determining the behaviour of the Fetch Unit of the RISCV Core when dealing with Branch or Jump instructions. This block is comprised of a 512 entry, 2-bit predictor table used for every branch instruction; the BPU, once it recognizes the instruction checks the table at the address provided by the lower 9 bits of the Program Counter to decide what to fetch, according to the scheme below

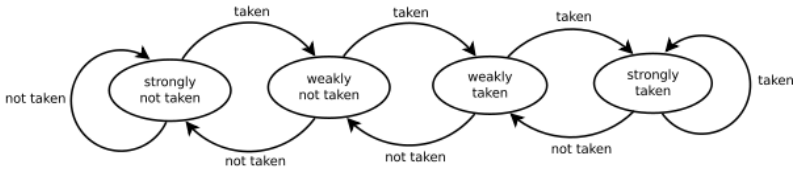

Figure 3.1: 2-bits Predictor Scheme

where the states of this FSM correspond to the following logic values:

- **2'b00**: Strongly Not Taken
- **2'b01**: Weakly Not Taken
- **2'b10**: Weakly Taken
- **2'b11**: Strongly Taken

The prediction is then given by the Most Significant Bit of the predictor.
If the BPU assumes that the branch has to be taken, then it's its responsibility to provide the jumping address. This is why the BPU is equipped with a 512 entry table, each entry having 32-bits to store the PC values. Along with this table there's a validity bit that signals whether the stored value is valid or not.
Here there's a very basic representation of the logic that drives the Program Counter value:

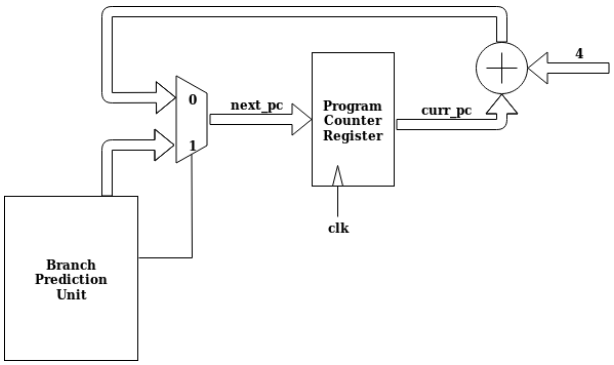

Figure 3.2: PC value logic

At every cycle, if the BPU finds a branch instruction, it checks the prediction bit: if it's taken, then the mux is driven such that the *next_pc* value is the one predicted, otherwise the mux will select the *curr_pc*+4. In case of misprediction, the BPU will correct itself in the subsequent clock cycle, otherwise everything will be leaved as it is.
A similar flow occurs when there's a JAL instruction: in this case there's no need for prediction, and, if the validity bit is set, the BPU will drive the PC accordingly without any further check (aliasing excepted).

3.2 Branch Forwarding Unit

The Branch Forwarding Unit is a rather simple module which task is to provide at any time the correct operands value to the BPU.
The idea behind this block is that when there's a *Read After Write* hazard, or RAW, at the same time of a branch instruction, once the hazard is solved it's necessary to forward the new value of the required operand in order to determine the correctness of the previous prediction.
Its internal functioning logic is quite similar to the one described for the Forwarding Unit in the datapath.

3.3 Instruction Cache and its Controller

The Instruction Cache was modeled as a set-associative cache of 4 sets, each one having 8 entries, each entry having 64 Bytes, i.e., 16 instructions. For each entry there's a *TAG* field - used to determine whether in a set there's one *HIT* or *MISS* - as well as a validity bit to determine if the entry is valid or not (in case of aliasing) or first time accessing the cache after reset.
The Instruction Cache comes with a Controller that handles the *miss* scenario and the byte reception from the IRAM. We assumed that the IRAM worked at one half the frequency of the processor core and that it stored the data in Bytes: the role of the *icache_controller* module is to recompose the received bytes in order to recreate the instructions that will be stored in the cache. Once the entry has been filled, the controller returns transparent and the instructions go from the cache to the rest of the core.

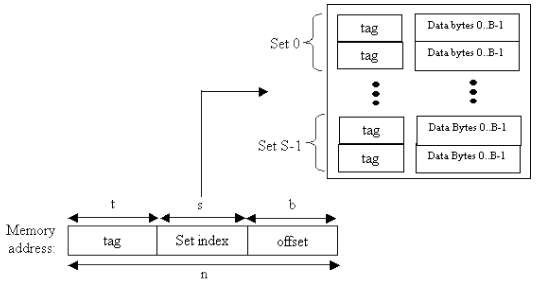

Figure 3.3: Set-Associative Cache Basic Structure

CHAPTER 4

Verification and Synthesis

In order to test our processor core we went through some steps that allowed us to develop a testbench that replicates as much as possible the real situation in which a CPU works.

4.1 GCC Toolchain and Test Programs

The first thing we worked on was to find a toolchain that could compile our code into an executable file and then something to convert the executable file into a text file containing the bytes to be loaded into the instruction cache.
In particular, among all the RISCV GCC toolchains that already exist, we were looking for something that would work for bare-metal systems, since we did not want to implement the necessary HW and FW support for OS. What we found was a flexible compiler[1] that allows to select the desired architecture and extension in order to determine which subset of instructions to use when compiling the *.c* file.
For our purposes, though, we decided to start from an assembly file in order to check directly whether the instructions work or not. Once the file was compiled using the command *riscv-none-embed-gcc -Wall -nostdlib -march=rv32i -mabi=ilp32 test_file.s -o test_file*, we used a basic C program, named *bin2hex.c* in order to convert the executable file into a text file containing all the bytes in which the converted instructions are formed. The text file we obtained in this way is

```
.text
.align 2

.global _start

_start:

# ITYPE Instructions

        addi    x1,x0,0x0E      # 0x0000000E
        andi    x2,x1,0x02      # 0x00000002
        ori     x3,x0,0x03      # 0x00000003
        xori    x4,x0,0x04      # 0x00000004
        slli    x5,x3,0x05      # 0x00000060
        addi    x1,x0,0         # 0x00000000
        sub     x1,x1,x2        # 0xFFFFFFFE
        srai    x6,x1,0x01      # 0xFFFFFFFF
        srli    x7,x1,0x01      # 0x7FFFFFFF
        slti    x8,x1,8         # 0x00000001
        sltiu   x9,x1,9         # 0x00000000

# RTYPE Instructions

        add     x10,x5,x2       # 0x00000062
        sub     x11,x5,x2       # 0x0000005E
        and     x12,x4,x1       # 0x00000004
        or      x13,x4,x2       # 0x00000006
        xor     x14,x4,x3       # 0x00000007
        sll     x15,x3,x4       # 0x00000030
        sra     x16,x1,x2       # 0xFFFFFFFF
        srl     x17,x1,x2       # 0x3FFFFFFF
        slt     x18,x1,x2       # 0x00000001
        sltu    x19,x1,x2       # 0x00000000
        sb      x1,0(x0)        # no effect
        sh      x1,4(x0)        # no effect
        sw      x1,8(x0)        # no effect
        lb      x20,8(x0)       # 0xFFFFFFFE
        lh      x21,8(x0)       # 0xFFFFFFFE
        lw      x22,8(x0)       # 0xFFFFFFFE
        lbu     x23,8(x0)       # 0x00000FE
        lhu     x24,8(x0)       # 0x000FFFE

# LUI/AUIPC Instructions

        lui     x25,0x800       # 0x00800000
        auipc   x26,0x10        # 0x0001007C

# Branch instructions and JALR

        beq     x25,x26,beq_lab #  FsDEMW
bne_instr:
        bne     x25,x26,bne_lab
blt_instr:
        blt     x1,x2,blt_lab
bge_instr:
        bge     x1,x2,bge_lab
```

Figure 4.1: instr_tester first part

[1] RISCV Toolchain: https://gnu-mcu-eclipse.github.io/toolchain/riscv/

the one that will be loaded into the behavioural IRAM in the same testbench in which the RISCV Core is instantiated.

We wrote three different files to show our core's capabilities: the first one is named *instr_tester.s* and its purpose is to test every instruction from the *I* extension our core can execute in order to test its correct functionality. As we can see in the pictures on the right, first we went through all of the i-type instructions, then the r-type, load, store and lui/auipc instructions and, ultimately, we tried all of the branch and jump instructions. This test was interesting not only because we can make sure almost every submodule and all of their interconnections work correctly, but we could also check the behaviour of the processor when there're instructions in the pipe and the cache has a miss. In fact, since there're many instructions, the cache had a few misses and we were able to see if the processor was able to handle correctly the instructions while fetching the new instruction block from the IRAM.

```
bltu_instr:
        bltu    x1,x2,bltu_lab
bgeu_instr:
        bgeu    x1,x2,bgeu_lab

jalr_lab:
        addi x2,x2,10           # 0x0000000C
        jalr x3,x1,0            # 0x0000009C
beq_lab:
        addi x27,x27,27
        j bne_instr
bne_lab:
        addi x28,x28,28         # 0x0000001C
        j blt_instr
blt_lab:
        addi x29,x29,29         # 0x0000001D
        j bge_instr
bge_lab:
        addi x30,x30,30
        j bltu_instr
bltu_lab:
        addi x31,x31,37
        j bgeu_instr
bgeu_lab:
        addi x27,x27,27         # 0x0000001B
        jal x1,jalr_lab         # 0x000000CC

# End of Program

_exit:
        j _exit
```

Figure 4.2: instr_tester second part

Next, we wrote *branch_stall_tester.s*, a test file that is used to check the stall/forwarding logic in every possible situation, together with the BPU test in order to show its features when dealing with a *while loop*.

```
.text
.align 2

.global _start

_start:
        addi x1,x0,0x0A
        sw x1,0(x0)
        lw x2,0(x0)     # FDEMW         # First stall: one cycle because of load instruction (next instruction should not be a branch one)
        addi x3,x2,0x0A #   FDsEMW
        beq x3,x1,_exit #    FssDEMW    # Second stall: one cycle because the branch instruction needs the operands in the Decode Stage
        sw x3,4(x0)

wrong_loop:
        lw x4,4(x0)     #       FDEMW   # Last stall: two cycles when having load-branch instruction
        beq x4,x2,wrong_loop #    FssDEMW

correct_loop:
        addi x1,x1,0x01         # Showing BPU capabilities
        bne x1,x3,correct_loop

_exit:
        j _exit
```

Figure 4.3: branch_stall_mem program

Lastly, we wrote a test file to check the functioning of the Multiplier/Divider module that comprises all of the instructions of the *riscv32m* extension, that are: **mul**, **mulh**, **mulhsu**, **mulhu**, **div**, **divu**, **rem** and **remu**.

```
.text
.align 2

.global _start

_start:
        lui x1,0x80000
        addi x1,x1,0x0A
        lui x2,0x80000
        addi x2,x2,0x0A
        mul x3,x1,x2
        mulh x4,x1,x2
        mulhsu x5,x1,x2
        mulhu x6,x1,x2

        addi x1,x0,0x04
        lui x2,0xFFFFF
        addi x2,x2,2046
        addi x2,x2,2047
        div x7,x1,x2
        divu x8,x1,x2
        rem x9,x1,x2
        remu x10,x1,x2


_exit:
        j _exit
```

Figure 4.4: muldiv program

4.2 Synthesis

The Synthesis process has been conducted by means of a very basic script that is shown below:

```
set PATH ../../HardwareDescription

# Analyze all of the RISCV Modules
### Constants ###
analyze -f sverilog -lib WORK $PATH/Constants/constants.sv
#################

### Fetch Unit ###
# Branch Prediction Unit
analyze -f sverilog -lib WORK $PATH/FetchUnit/BranchPredictionUnit/bpu.sv

# Branch Forwarding Unit
analyze -f sverilog -lib WORK $PATH/FetchUnit/BranchForwardingUnit/bfu.sv

# Instruction Cache
analyze -f sverilog -lib WORK $PATH/FetchUnit/InstructionCache/i_cache.sv
analyze -f sverilog -lib WORK $PATH/FetchUnit/InstructionCache/icache_controller.sv

# FetchUnit Block
analyze -f sverilog -lib WORK $PATH/FetchUnit/fetch_unit.sv
##################

### Register File ###
analyze -f sverilog -lib WORK $PATH/RegisterFile/reg_file.sv
#####################

### Arithmetic Logic Unit ###
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/mux2to1.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/mux4to1.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/register.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/shiftRegister.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/syncCounter.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/fullAdder.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/divZeroDetect.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/adder.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/carrySaveAdder.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/comparator.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/divOvfDetectBlock.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MulDivComponents/kernelLogic.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MultDivUnitDP.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MultDivUnit.sv
analyze -f sverilog -lib WORK $PATH/ALU/MultiplierDivider/MultDivUnitWrapper.sv

analyze -f sverilog -lib WORK $PATH/ALU/alu.sv
#############################

### Forward Unit ###
analyze -f sverilog -lib WORK $PATH/ForwardUnit/forwardunit.sv
####################

### Control Unit ###
```

```
analyze -f sverilog -lib WORK $PATH/ControlUnit/cu.sv
##################

### Data RAM Controller ###
analyze -f sverilog -lib WORK $PATH/DRAMController/dram_controller.sv
############################

### RISCV Core ###
analyze -f sverilog -lib WORK $PATH/RISCVCore/riscv_core.sv
##################

# Elaborate top entity module
elaborate riscv_core -lib WORK

# Create clock
create_clock -name riscv_clk -period 0 clk
set_dont_touch_network riscv_clk

# Compile
compile

# Saving reports
report_timing > report_timing.txt
change_names -hierarchy -rules verilog
write -f verilog -hierarchy -output riscv_syn.v
```

As we can see, no further optimization was required and we set the clock period to 0 in order to see how small the slack could be.
The synthesis process was carried out both for an ASIC and an FPGA target.
In the case of an FPGA, we used the *Quartus Prime* tool, targeting the *Cyclone IV EP4CE75F23C6* (as it was the one with the smallest number of Logic Elements, or LEs, above the minimum required by our core). Here are the obtained results:

Total LEs	60,331/75,408 (80%)
Total Registers	47,349
Pins	119/243 (41%)
Slack (Fast Model) [ns]	11.788
Slack (Slow Model, 85C) [ns]	20.907
Slack (Slow Model, 0C) [ns]	18.797

Table 4.1: FPGA-based synthesis report

As for the ASIC implementation, these are the features of our core:

Slack [ns]	2.15
Area [μm^2]	425295.6875
Power [mW]	7.4807

Table 4.2: ASIC-based synthesis report

Depending on the goal - whether to optimize power consumption, speed or area - one can choose which parameter to optimize, in order to get even better results in at least one of these fields.

4.3 Pre and Post-Synthesis Simulation

We've conducted two kind of simulations, the pre-synthesis one that was intended to check the logic functioning of the hardware description of our CPU core, the post-synthesis one was intended to check that the synthesis process went smoothly and the functionality was kept as we intended.
The three files presented above have been tested succesfully in the pre-synthesis phase, we checked that the synthetized core was still working properly by running *instr_tester.s* for the non-Multiplier/Divider datapath and *muldiv_tester.s* for the Multiplier/Divider unit.

Conclusion

The proposed core has been shown and described in its details, tested before and after - at least for the I extension - the synthesis and so far it shows a correct behaviour in every scenario.
There is still room for improvement, such as the development of a Cache module and Controller for the DRAM as well, but in our opinion the dynamism of the RISCV environment make it worthy to continue developing CPUs, as well as moving to the usage of SystemVerilog as a HDL.