Universitat Autònoma de Barcelona

A Thesis for the

Master in Telecommunication Engineering

RISC-V Core Instruction Extension Sets M and F

By

Francisco Javier Fuentes Diaz

Supervisors: Raimon Casanova Mohr
Lluis Teres Teres

Microelectrònica i Sistemes Electrònics (MiSE) Department

Escola Tècnica Superior d'Enginyeria (ETSE)
Universitat Autònoma de Barcelona (UAB)

June 2021

El sotasignant, *Raimon Casanova Mohr*, Professor de l'Escola Tècnica Superior d'Enginyeria (ETSE) de la Universitat Autònoma de Barcelona (UAB),

CERTIFICA:

Que el projecte presentat en aquesta memòria de Treball Final de Master ha estat realitzat sota la seva direcció per l'alumne *Francisco Javier Fuentes Diaz*.

I, perquè consti a tots els efectes, signa el present certificat.

Bellaterra, dilluns 21 de Juny del 2021.

Signatura: *Raimon Casanova Mohr*

Resum:

Aquest projecte de tesi presenta el disseny dels components integrals dels processadors de 5 etapes RV32I, RV32IM amb expansió de multiplicació i divisió d'enters, i RV32IMF amb suport parcial de coma flotant de simple precisió.

Aquests han estat desenvolupats utilitzant Verilog HDL i donen suport al sets d'instruccions RISC-V. A més, aquests dissenys han estat verificats i sintetitzats en "bare metal" mitjançant la plataforma FPGA de la placa de desenvolupament DE0.

Addicionalment, s'ha dissenyat una varietat de mòduls divisors, oferint diversos rendiments en freqüència d'operació, assignació de recursos i nombre de cicles de rellotge en operacions de divisió. La selecció entre aquests mòduls aporta opcions d'implementació per ajustar el producte a les necessitats del client.

Resumen:

Este proyecto de tesis presenta el diseño de los componentes integrales de los procesadores de 5 etapas RV32I, RV32IM con expansión de multiplicación y división de enteros, y RV32IMF con soporte parcial de coma flotante de simple precisión.

Estos han sido desarrollados utilizando Verilog HDL y apoyan sets de instrucciones RISC-V. Además, estos diseños han sido verificados y sintetizados en “bare metal” mediante la plataforma FPGA de la placa de desarrollo DE0.

Adicionalmente, se ha diseñado una variedad de módulos divisores, ofreciendo diversos rendimientos en frecuencia de operación, asignación de recursos y número de ciclos de reloj en operaciones de división. La selección entre estos módulos aporta opciones de implementación para ajustar el producto a las necesidades del cliente.

Summary:

This thesis project presents the hardware design of the components capable of implementing a 5-stages core RV32I, RV32IM with integer multiplication and division expansion, and RV32IMF with partial single-precision floating-point support.

These have been developed using Verilog HDL and based on the RISC-V ISA. Furthermore, these designs have been verified and synthesised on “bare-metal” using the FPGA platform from the DE0 development board.

In addition, a custom variety of division modules have been produced to offer performance diversity on frequency of operation, resource allocation and number of clock cycles per division operations. The selection of these modules provides implementation options that allow to personalize the product to the customer needs.

Table of Contents

List of Figures

List of Tables

1 | Introduction

The rapid expansion of computers over the world has been a global milestone that has changed humanity. This has been primarily thanks to the advancements on the Integrated Circuit (IC) manufacturing field, providing the technological means to produce microprocessors at lower cost and better performance with each generation. However, the computer architectures used on most prolific devices are based on x86 or ARM, two Instruct Set Architectures (ISA) that are proprietary and fully closed. Meaning only Intel, AMD (x86 ISA) and Arm (ARM ISA), the two main designer companies of processors and holders of the rights, are exclusively allowed to develop processors based on these ISAs, artificially increasing the cost microprocessors [1].

To confront this situation, the University of Berkley has been developing since 2010 the RISC-V open source ISA. The open source characteristic endorse anyone to design processors and commercialize them without acquiring a RISC-V license, leaving the rights of the product to the designer. The goal of this ISA is to increase the number of microprocessor products on the market and increase its performance by allowing different processors to be specialized on each computer field. The decentralization move on architectural design has been well received by many enterprises from the computer industry [2, 3]. The fast growth and popularization of RISC-V compatible products [4] creates a healthy ecosystem with a bright future on the computer industry.

1.1 Project overview

This project seeks to develop a core processor based on the RISC-V ISA. However, the RISC-V ISA is extensive, but in order to provide design space for customization, the various instruction sets it includes are modular in nature. This allows us to limit the focus of the project on designing a core compatible with the subsets M and F, supporting integer multiplication, division, and single-precision floating-point operations.

These core features are compatible with future RISC-V systems, since both of these subsets are ratified as shows Table 1.1. This table list all the instruction sets included on the unprivileged ISA and their status at the moment of writing.

Base	Version	Status	Extension	Version	Status
RVWMO	2.0	Ratified	Zifencei	2.0	Ratified
RV32I	2.1	Ratified	Zicsr	2.0	Ratified
RV64I	2.1	Ratified	M	2.0	Ratified
RV32E	1.9	Draft	A	2.0	Frozen
RV128I	1.7	Draft	F	2.2	Ratified
			D	2.2	Ratified
			Q	2.2	Ratified
			C	2.0	Ratified

Table 1.1: RISC-V ISA base and extensions partial extraction from the manual [5], page 1.

It should be noted the naming convention followed by the RISC-V ISA fixes that the core model name must include the letter of all supported sets in order from the table. The exception to the rule is of those that are dependent from another supported set, as is the case with Zicsr and F. For example, a base RV32I core implementing 32-bit integer arithmetic can be expanded with the extension set M to support integer multiplication and division instructions. This core would be addressed as RV32IM.

This thesis project uses as foundation a pipeline 5-stage RISC-V RV32I core. This has been built by adapting a RISC-V single-cycle 32-bit core from a previous Bachelor's Degree project, which project files can be accessed at its GitHub repository page [6]. Similarly, this project is also open source and all its contents, including technical documentation, are available at its GitHub repository page [7].

The project objectives achieved by this project are:

SO1. Study different algorithms to implement the hardware to support the operations from subsets M and F, including getting to know the capabilities of a Register-Transfer Level (RTL) designer and the limits of designing for a Field-Programmable Gate Array (FPGA) platform.

SO2. Learn to write in Verilog Hardware Description Language (HDL) and design the modules capable of implementing the required operations of the extension sets M and F.

SO3. Take to understand the base RV32I core developed by another engineer and modify it to expand its capabilities up to the RV32IM and RV32IMF core models.

SO4. Create a validation framework and validate the designed components through a simulation environment. These tests must confirm the developed circuits are RISC-V compliant.

SO5. Prototype the RV32IM and RV32IMF core models at the FPGA platform of the development board DE0 and recompile data about performance and resource usage using a synthesis software program.

Even though this project focus on hardware development by coding on Verilog HDL, this document presents the work done without showing any line of code. The intent is to allow readers unfamiliar with the language to be able to understand the contents of the dissertation with basic hardware architecture knowledge. Some useful references to learn to code are [8, 9].

The tools used in this project include the EP3C16D484C6 FPGA model from the Cyclone III family [10], central component of the **DE0 development board** [11] used for porting the project. Due to this FPGA model, the chosen synthesis tool used to port is **Altera Quartus II 13.1 Web Edition** [12] (free licence), being the latest version supporting the prototyping FPGA platform. During the design and validation of the hardware developed, the chosen simulation tool is **ModelSim PE Student Edition 10.4a** [13] (free license). These tools are independent from the RISC-V ISA, requiring more work from part of the designer to use them on this application. However, these are well established tools, meaning they are well documented.

In addition, by using the programming environment **Eclipse** [28] with the open source **xPack RISC-V compiler** [15] add-on, it has been possible to compile code-machine programs from C code. These has been used to validate even further the processor designs. An installation manual of this tool has been uploaded to project's GitHub repository page [7] for future reference.

The development process starts by understanding the operations to be supported on the implementation, fixing the requirements of each component. These are set by the RISC-V ISA manual [5], which it has been referenced at any time instructions or components from the standard are brought to the design.

Then, the various behavioural requirements are divided in a number of sub-modules, taking into account the state of the art. Both design and test code are written and used in an iterative design/validation process until the logic correctly implements the execution and is RISC-V compliant. Finally, the sub-module is adapted on a higher hardware level module and the last process is repeated until the core capable can be ported and tested on the FPGA platform.

This same process is presented on this dissertation, limiting the explanation to final designs and data representative of the work done, while leaving state of the art on simpler explanations to understand design decisions. Starting by the instruction set M sub-modules development on the next Chapter 2, set F on Chapter 3, core presentation and expansion on Chapter 4 and porting at the FPGA platform on Chapter 5.

2 | Extension Instruction Set M

The extension instruction set M of the RISC-V ISA provides 32-bit integer arithmetic support for multiplication and division operations. It includes 8 instructions classified into 2 categories: MUL, MULH, MULHU and MULHSU for multiplication and DIV, DIVU, REM and REMU for division operations.

On the first category, the instructions MUL and MULH output the low and high 32 bits respectively of the 64-bits product between two 32-bit signed operands. The instructions MULHU and MULHSU output the high 32 bits of the 64-bits product between two unsigned and between one signed and one unsigned 32-bit operands respectively. On the second category, the instructions DIV and DIVU output the 32-bit quotient of the division between two signed and between two unsigned 32-bit operands respectively. The instructions REM and REMU output the 32-bit remainder of the division operations from the DIV and DIVU instructions respectively.

The result of the operation is signed when any of the input operands are, meaning the instructions MUL, MULH, MULHSU, DIV and REM output signed operands and MULHU, DIVU and REMU output unsigned operands.

This chapter describes the implementation of a module capable of executing the M instruction set operations, the MULDIV block. It has been designed in a modular topology as Figure 2.1 presents, allowing it to be incorporated at a RISC-V core.

Considering the instructions from the set M support operations between unsigned and signed operands, both input and operation result pass through conversion blocks. These compute the unsigned conversion from the input, used to operate, and the signed conversion from the operation result when required, allowing the divider and multiplier to be shared by operating with unsigned values unless stated otherwise.

However, the implementation of this execution module on a core requires additional control logic to manage multi-cycle operations by asserting the *busy* flag. These modifications are presented on section 4.2.

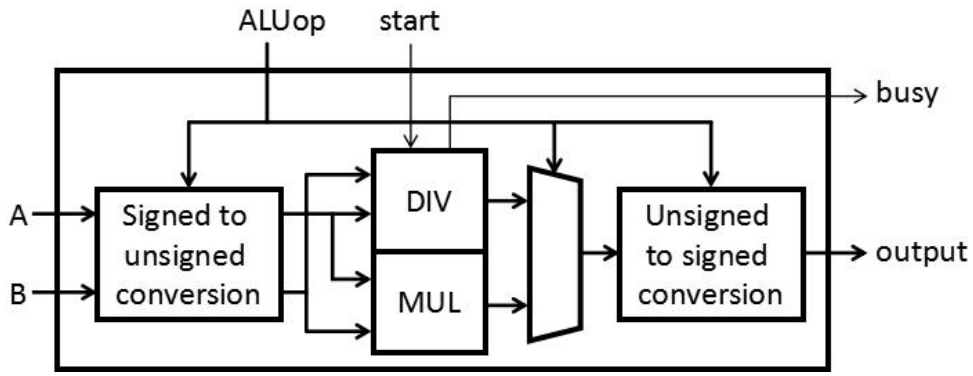

Figure 2.1: Block diagram of the MULDIV that executes the set M operations.

Due to the multiplication and division are intensive arithmetic operations, the circuits implementing these require a proper study and optimization on the area and speed performance in a trade-off exchange. The following section 2.1 introduces the figures used to measure both area and speed, and the general techniques applied to optimize the hardware being developed in this project.

The section 2.2 focuses on the development of the division module, by presenting state of the art algorithms, implementing optimizations and benchmark the various designed models. The section 2.3 focuses on the implementation of the multiplier circuit, by presenting the multiplication algorithms and performing an analysis of the intelligence property cores available and how are used to implement the multiplier.

The section 2.4 focuses on the development of the MULDIV, that is the module that encapsulates the hardware of previous sections to support the extensions set M operation meeting the RISC-V specifications.

The designed modules pass through several tests in a simulation environment to verify they meet requirements on section 2.5.

At last, the designed hardware is synthesized on our prototyping FPGA platform, obtaining the synthesis data used to analyze what module combination, among all the developed components in this chapter, performs the best on section 2.6.

2.1 Figures of performance and optimization methods

This project uses three main figures to measure and rate the performance of the developed hardware, resource usage, maximum frequency of operation and operation latency.

- The resource usage is measured in Logic Elements (LE), being the number of logic blocks that the Device Under Test (DUT) synthesizes on our particular FPGA prototyping platform. Even though the resource usage depends on the platform, it can be estimated for other FPGA platforms by multiplying it with a ratio. This is also the case for Application Specific Integrated Circuit (ASIC) implementations, which the analogue figure is implementation area in square micrometers.
- The Maximum Frequency of Operation (MFO) is measured in MegaHertz (MHz), being the maximum frequency the DUT allows without creating operation inconsistency. This happens when the internal signals are not stable long enough in time to be propagated through the logic elements correctly between register elements as cause of the high frequency of operation, resulting on wrong data being transmitted on the following clock cycles. The MFO is limited by the critical path of the DUT, that is the data path with the longest gate delay between register or flip-flop elements, making it to depend on the platform. However, the MFO on other platforms can be estimated through a ratio.
- The operation latency is measured in clock cycles, being the total number of clock cycles required by the DUT to complete the operation and output the correct result. The operation latency does not depend on the platform and is exclusive to the data path of the algorithm being executed.

However, it is important to note that, except on specific scenarios, the MFO is much important to optimize than latency. The cause is a lower MFO affects the performance of the core on all operations. By contrast, the latency is specific to the operation the DUT executes.

On the evaluation of advanced cores, it is common to present both implementation area and throughput (the number of bits being processed per unit of time) figures. However, **this project does not implement in any way features that allow parallel instruction processing or execution**, which would increase the throughput without modifying the MFO nor the latency. In turn, the throughput is directly proportional to the MFO and latency, allowing us to present the same data through these two figures over the throughput. This facilitates the data analysis, as the throughput is less clear on what figure, frequency or latency, is the DUT performing better.

Intensive arithmetic operations, such multiplication and divisions, require a trade-off analysis and optimization between resource usage, MFO and latency to achieve better performance on some of the figures. On the case of the division algorithms, they are thought to be implemented in a closed data path loop (fully rolled topology) for a number of iterations until the operation completes, fixing the latency of the operation.

A direct implementation of the algorithm usually offers the design with the lowest implementation area specific to the algorithm. The MFO and latency can be traded-off by segmenting and registering the data path in a **pipeline structure, exchanging a low latency for a higher MFO**. **The contrary exchange option is achieved by extending the closed data path, method named unrolling the roll**, to execute two or even more iterations on one clock cycle. However, the resource usage is increased while the MFO and latency are decreased at a rate equal to the number of iteration being processed per clock cycle, as unrolling the roll adds duplicates of the operation logic in series without additional registers.

Additionally, we are referring to **"dynamic" optimizations when, through operand identification, a number of iteration stages are skipped in the execution**, reducing latency for that specific operation. The number of clock cycles saved depends on the hardware added and the bit composition of the identified operand, meaning **dynamic optimizations do not save a fixed number of clock cycles for all operations**. Dynamic optimizations, thus, are especially useful on operations with high latencies because they allow to jump over a high number of iteration stages. However, they also add a high number of resources, which could also lower the MFO.

An important characteristic on the optimization of the MFO and latency figures is that **the latency of the DUT can be decreased though optimizations with a MFO limit of the core's MFO**. This is because the core implements a critical path fixing a global MFO, value that cannot be increased unless the part of the hardware that contains the critical path is modified. For this reason, unless the DUT implements a critical path more constrained on the MFO that the already present on the core, the **DUT can be further optimized for a lower latency without decreasing the general MFO**. However, the optimizations still increase the resource usage. A reference source for FPGA oriented optimizations is [16].

2.2 Integer divider

The integer divider executes the division operation, providing the result of the 32-bit unsigned quotient and 32-bit unsigned remainder between two 32-bit unsigned operands (dividend and divisor).

First an algorithm review is performed to find the adequate algorithm to base on the division circuit on section 2.2.1. Then, a division model based on that specific algorithm is developed on section 2.2.2, which is expanded with various optimizations on section 2.2.3 and benchmarked on section 2.2.4.

The common timing (the specific execution being performed at each clock cycle of the operation) that can be adjusted to any of the division algorithms presented by the following section 2.2.1 have to follow the Figure 2.2 structure.

First, one clock cycle is spent on to load the unsigned operands at (1). Then, a number of clock cycles equal to the iterations required by the DUT of the algorithm are spent to compute the operation at (2). At last, an additional clock cycle is spent to retrieve the unsigned result at (3). The control signals at the input (*start*) and output (*busy*) must follow the Figure 2.2 timings to avoid mismanagement of the execution, fixing the behavior to implement at the core and the division circuit respectively.

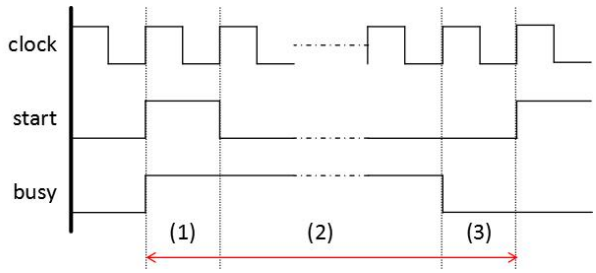

Figure 2.2: Timing rule diagram of the control signals for multi-cycle operations. The red line depicts the total latency of the operation in question.

Even though this timing rule increments the division latency with two additional clock cycles, this design leaves delay slack at the input and output paths. This is important, as will offer the MFO possible when it is incorporated at the MULDIV on section 2.4, by allowing the conversion logic to not set the critical path on the input or output.

2.2.1 Division algorithms

Through decades of computer advancements, several logic algorithms have been devised to be the base of complex logic processes, such as the division operation. These can be implemented by a closed looped on a fully rolled topology that takes multiple iterations to complete the operation and output the result, avoiding long critical paths to provide high MFO circuits with low implementation area.

The algorithm diversity offers different resource requirements and performance values that can be modified by applying the optimization techniques presented on the previous section. All algorithms can also be implemented on a fully unrolled topology with a latency of one single clock cycle. However, this is usually undesired, as the resource usage increases and the MFO decreases at a rate proportional to the number of iterations of the base algorithm.

The division algorithms studied to be the base of the division design can be classified into two categories: fast and slow division algorithms. The fast category includes the restoring, non-restoring and SRT algorithms, while the slow category includes the Goldschmidth and the Newton-Raphson algorithms. Note that the classification of “fast” and “slow” does not refer to the number of iterations of the algorithm, but to the MFO they implement. This is a result of the slow types requiring multiplier

circuits, which imply high gate delays, thus, lower MFO as trade-off for the low latencies. At continuation, all division algorithms are presented, although the following references explain them in higher detail [17-21].

- The **restoring division** algorithm follows the naive method learnt at schools to perform a division operation, by subtracting a multiple number of times the divisor from the dividend. However, at the binary system format, these subtractions are performed by **subtracting the divisor from the left-shifted dividend for a number of iterations** at distinct bit positions. If the subtraction result is positive, it is saved as the next dividend previous to the next shift. If the subtraction result is negative, it is discarded and the dividend previous to the subtraction is "restored" for the next iteration, hence the algorithm's name.
- The **non-restoring division** algorithm is based on the restoring algorithm, but it always saves the subtraction as the dividend for the next iteration. Thus, never restoring the dividend, hence the name. However, this generates negative operands, which the **non-restoring algorithm performs additions instead of subtraction**. This characteristic increases the logic of the circuit, while also requiring an additional clock cycle to rectify the possible negative result.
- The Sweeney Robertson Toacher or **SRT division** algorithm is based on the non-restoring algorithm but allows **multiple bit-shifting per iteration, reducing the latency of the operation compared to the previous algorithms**. This is achieved by using the most significant bits of the mid-operand to address a Look-Up Table (LUT) that contains pre-calculated quotient bits to generate. These LUT values are non-trivial to find. However, it is possible to decrease the number of iterations by half, a quarter, an eight or even a sixteenth by increasing the number of LUT bits quadratically. Because of this reason, there are studies that demonstrate designs with high radix (the number of bits used to address the LUT, equal to the ratio the latency is lowered) are area inefficient [22].
- The **Goldschmidt** algorithm computes the quotient through the **execution of a series of multiplication that find the divisor's reciprocal by limit approximation**. Again, an increase of performance in terms of latency is achieved at expenses of increasing the number of hardware resources. This is

reflected by the requirement of doubling the width of the buses for a fractional binary representation and the use of multipliers in parallel to decrease the latency. This algorithm is capable of performing a division with few clock cycles (low latency), but the bit-width of the multipliers imply a low MFO. Additionally, it is possible the use of precise initial reciprocals through LUT addressing, skipping initial iteration stages, thus, lowering the latency even further. However, the LUT values increases quadratically as happened on the SRT algorithm, but they are easier to generate.

- The **Newton-Raphson** algorithm is based on the Goldschmidt algorithm, but **the iterative execution is performed by two multiplications in a pipeline structure**, requiring one multiplier. This avoids to double bus width of the multiplication, thus, requiring lower resources and having higher MFO. However, in comparison with the previous Goldschmidt algorithm, the Newton-Raphson has a higher initial latency that can be also decreased with a LUT.

Table 2.1 summarizes the main feature of these four algorithms, used to assess the most adequate algorithm to base on the divisor. Starting by the restoring and non-restoring algorithms, both implement high MFO with high latency circuits, but the restoring requires lower resources.

The SRT algorithm maintains high MFO with the ability to exchange the low resource profile for lower latencies through a LUT. However, these LUT values are complex to find and manage, which untimely discards it.

N = 32 bits	Restoring	Non-Restoring	Goldschmidt	Newton-Raphson
Latency (cc)	32	33	12 down to 7	20 down to 5
Multiplexer	32-bit x2	32-bit x2	64-bit x2	67-bit x1
MUL module	-	-	64-bit x2	34-bit x1
ADD/SUB block	33-bit SUB	33-bit ADD/SUB	128-bit SUB	34-bit SUB
Register	32x2, 33x1	32x2, 33x1	64x2	32x2, 34x1
Possible LUT	No	No	Yes	Yes
Qualitative MFO	High	High	Low	Medium

Table 2.1: Resource and latency estimation for each division algorithm supposing stable input operands. SRT algorithm is omitted, as it is too complex to provide a safe estimate.

Between the Goldschmidt and Newton-Raphson algorithms, the higher MFO and lower number of multipliers makes the Newton-Raphson algorithm to be preferable over the Goldschmidt's. This leaves the both Newton-Raphson and restoring algorithms, which it has been decided **the restoring algorithm to be the base for the design of the division module**.

Two main factors have been for this decision. First, the Newton-Raphson algorithm inherits the fractional binary system from the Goldschmidt's, which complicates not only the design, but also the verification and debugging of the hardware. Secondly, the Newton-Raphson requires one multiplier, an expensive component in terms of area, which trade-off for lower latency can be matched by an optimized restoring design.

For that reason, the next section explains the design of the divider based on the restoring algorithm, named the Theoretical model, followed by a number of models that achieve to lower the high latency by lowering the MFO and increasing the resource usage as trade-off through applying various optimizations techniques.

2.2.2 Theoretical division model

The Theoretical (T) division model executes the procedure presented by the restoring algorithm, as simple as possible, in order to minimize the resource usage and present a high MFO in a fully rolled topology.

The recurring execution logic is implemented by two 32-bit registers used to store mid-operation operands, two 32-bit multiplexers and one 33-bit subtracter as shown in Figure 2.3. The diagram only shows the data path of one iteration of the algorithm, requiring a control block to keep executing iterations until the operation is finished, following the Figure 2.4 (a) flux diagram.

The control block implements a state machine represented by the Figure 2.4 (b). First, the circuit is at the idle state A, where an internal counter is set to zero and the *busy* control flag, used to enable the operation registers and signal the operation is not finished yet, is set to '0'. Once a *start* is asserted from outside of the divider, the control block changes to the state B of load operands, setting the *busy* flag to '1' and loading the dividend into the quotient register.

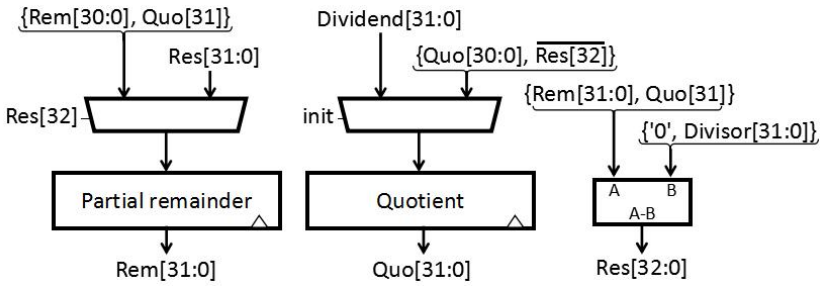

Figure 2.3: Block diagram of the Theoretical division model.

After one clock cycle, the control block changes to the state C, where the counter increases with each clock cycle, maintaining at what iteration number the execution of the operation is at. This stage C is where the 32 algorithm iterations are executed.

When the counter arrives to the iteration 31, it executes one iteration more and changes to the stage D, setting the busy flag to '0' to indicate the operation has finished. The quotient and remainder results are extracted from the "partial remainder" and "quotient" registers respectively. After another clock cycle, the control circuit returns to the idle stage A, waiting for a new division operation.

This timing behavior fixes **the total latency of the T model to be 34 clock cycles.** Note that it matches the timing rule set by the previous Figure 2.2.

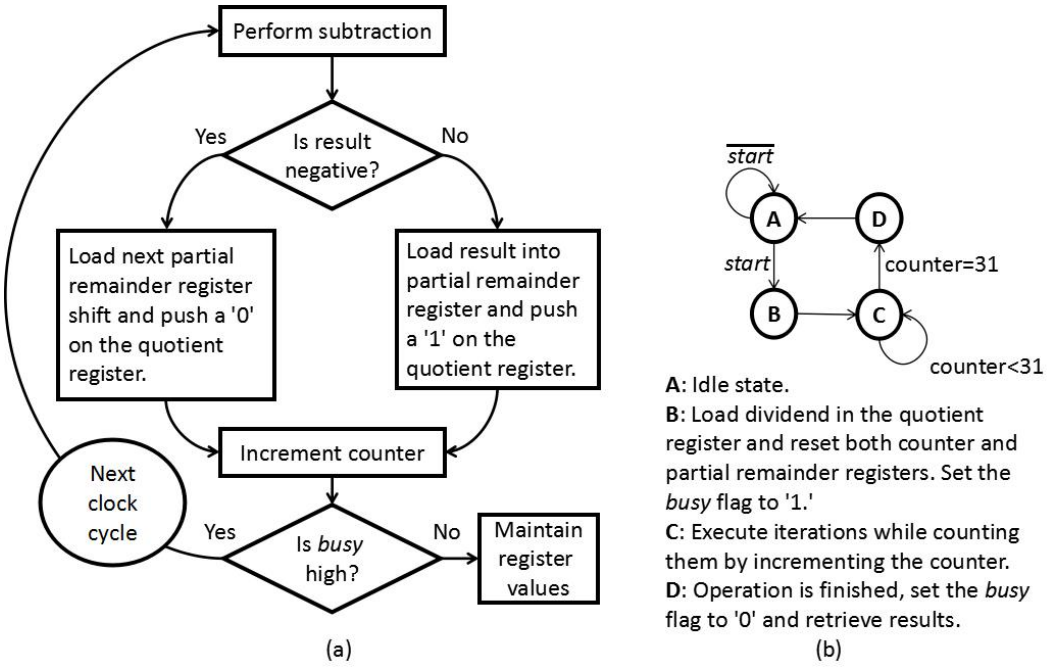

Figure 2.4: Flux diagram of the operation's iterative process on the Theoretical model (a) and state machine diagram (b) used to enumerate the operation stage and control the iterative logic.

The critical path of the T model can be predicted to be the subtracter element. Because the section 2.3 implements a multiplier, it can be deduced that the MFO of the T model is far higher than the multiplier. For this reason, several optimizations can be applied on the T model, generating model diversity that decreases the latency at expense of lowering this high MFO and increasing the low resource usage on various degrees, as shown at the following section.

2.2.3 Division optimizations

The previous Theoretical model uses the lowest number of resources and offers the highest MFO but presents a high clock latency of 34 clock cycles.

To decrease the latency of the T model, a number of models with various optimizations have been developed as Figure 2.5 introduces. These models are mapped on the graph between dynamic and unrolling the roll optimizations, which disposition on the figure indicate the type of optimization implemented or expanded during the evolution of each model.

For example, both Double and Quick Start models are based on the Basic model, which in turns is based on the T model. This means that both models inherit the characteristics from the Basic model while applying different optimizations.

Specifically, the Basic model skips the operation when the dividend is lower than the divisor, hence a dynamic optimization. Both Quick Start and Even Quicker Start models try to start the division operation at an advanced iteration stage, characteristic which the Skip Execution and its Double model counterparts also include. The Skip Execution model applies the same logic than the Even Quicker Start model also on the mid-operands to skip iteration stages during execution.

However, the Double model is the only optimization that unrolls the roll to execute two iterations stages per clock cycle. It could be extended to 3 or 4, but ultimately has been limited to 2 in order to avoid decreasing the MFO in excess.

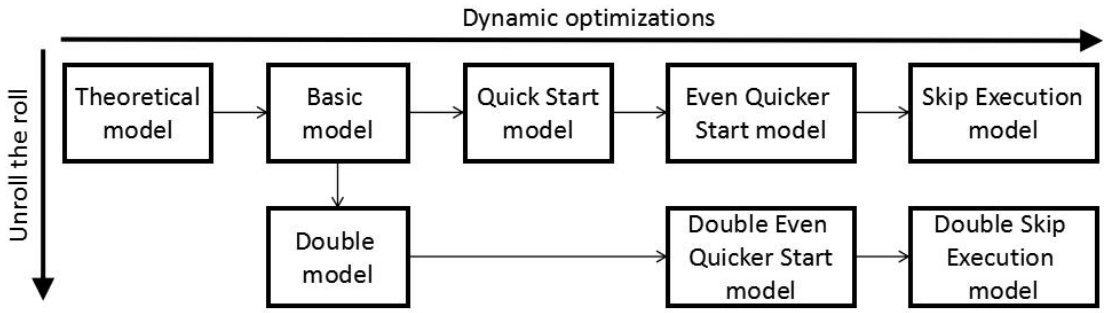

Figure 2.5: Model precedence between the division models.

The following list details every model optimization carried out and its general implementation:

- **Basic (B) model**: it exploits those situations where the divisor is higher than the dividend. In those cases, there is no need to incur on the execution of the operation because the quotient is always zero and the remainder is equal to the dividend.

  This is achieved by blocking the *start* input signal to the control circuit, gate set by **comparing both input operands, reducing the clock latency to one clock cycle on such cases.** Other operation cases are unaffected by this optimization, maintaining the original latency of 34 clock cycles.

- **Double (D) model**: it doubles the number of quotient bits generated per clock cycle. This effectively halves the latency of the iterative execution, lowering the **maximum latency to 18 clock cycles.** This optimization of the latency is independent from the inputs, except for those filtered by the Basic design that retain the single-cycle operation property.

  The latency improvement is achieved by **duplicating the subtraction logic in series and shifting the dividend twice per clock cycle**, taking the second subtraction result priority if positive as shown in Figure 2.6. Furthermore, the counter is adjusted to increment by two every clock cycle, in order to maintain the same iteration stage enumeration of the operation with other models. Because of this enhancement, **the Double model is limited to work exclusively on even stages of the operation**.

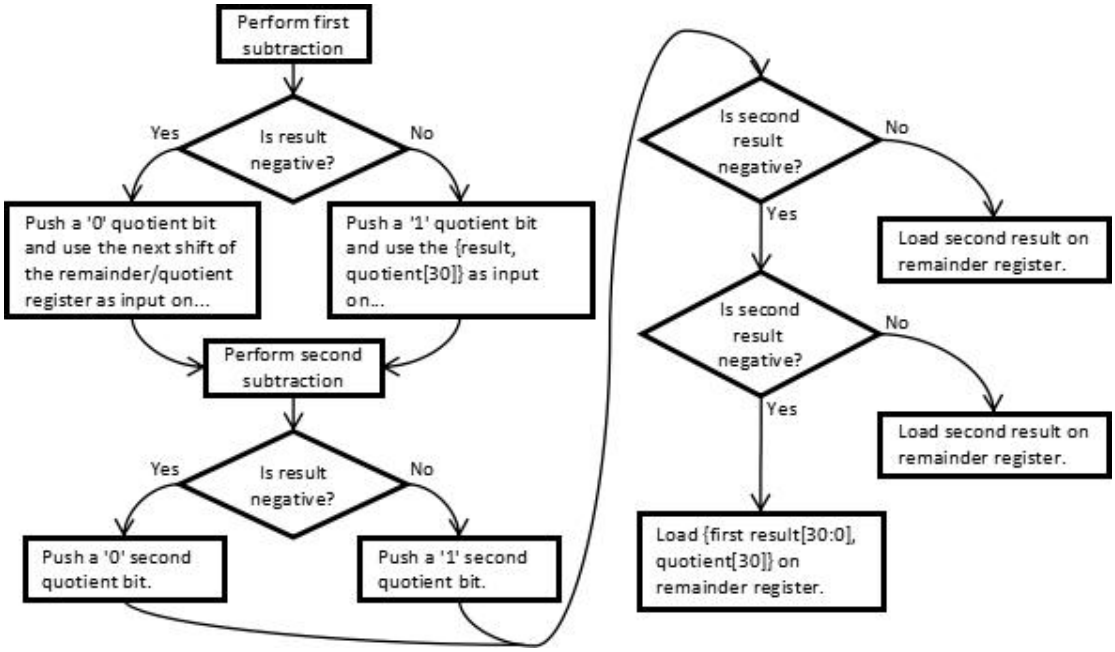

Figure 2.6: Flux diagram of the operation's iterative process on the Double model.

- **Quick Start (QS) model**: an identified trait of the restoring algorithm's data path is that for every '0' bit of the dividend operand by the left, a '0' quotient bit is generated. This is result of the positive input of the subtraction operation is always zero during these iterations, generating '0' quotient bit until the iterative shifting puts a value greater than the divisor on the partial remainder register.

  The Quick Start model takes advantage of this characteristic by **shifting the dividend to the MSB position on the quotient register during operand loading**. This operation is performed by first identifying the position of the most significant '1' bit of the dividend with the circuit shown in Figure 2.7. It is referred further in this document as the Most Significant High Bit (MSHB) block. The value generated by the MSHB circuit is used to initialize the counter on the same clock cycle as the loading of the shifted dividend.

  The number of iterations saved by this optimization is equal to the number of '0' bits at the left of the dividend, because the operation starts on an advanced stage compared to the previous presented models. Thus, this optimization is dynamic, that is the number of clock cycles depend over the input of the operands, the dividend specifically for this QS model.

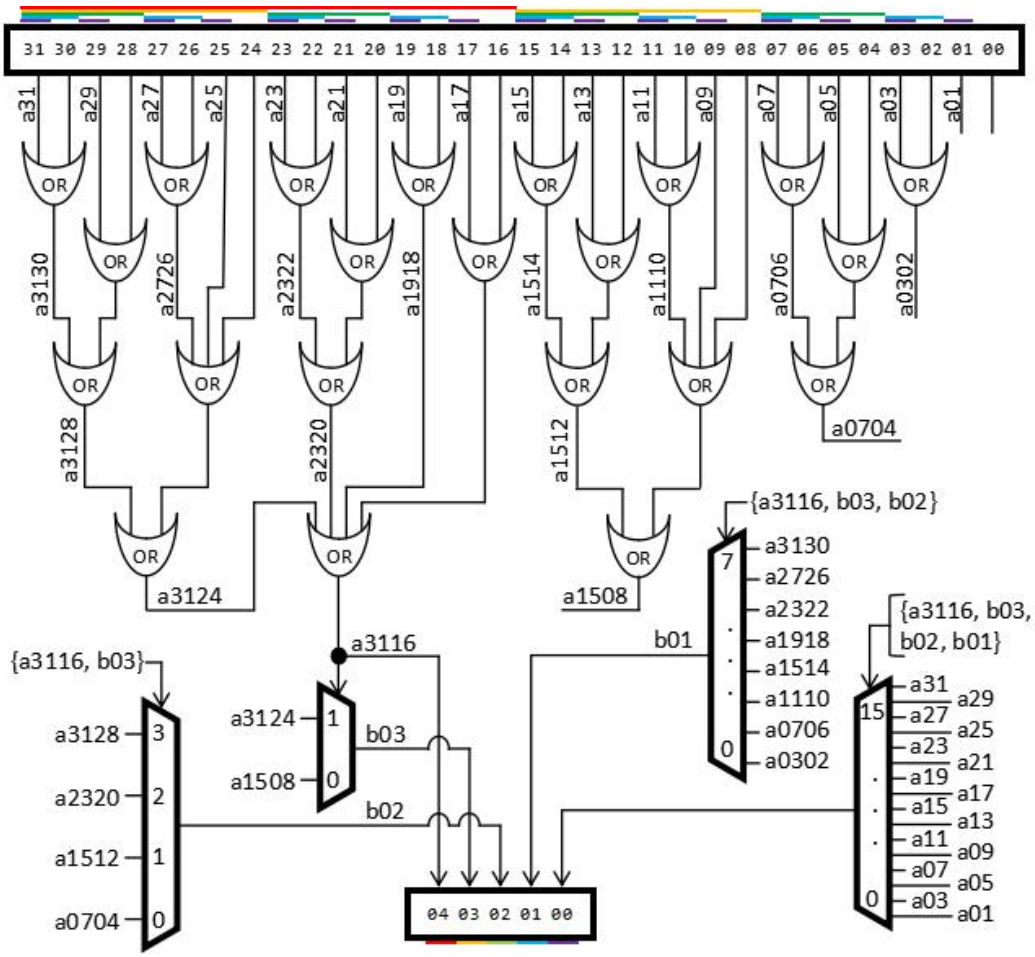

Figure 2.7: Block diagram of the Most Significant High Bit (MSHB) circuit.

- **Even Quicker Start (EQS) model**: it improves the QS model by guaranteeing that the first or second iteration generates a '1' quotient bit at least, starting the execution on an even more advanced stage of the operation, especially on those with high divisor values.

  The EQS **shifts the dividend when loading the operand into the remainder/quotient registers to match the '1' bit of the subtracting divisor.** The number of shifts to perform is equal to the position value of the most significant '1' bit of the divider plus the number of left-zeros of the dividend. That is the reason an additional MSHB circuit is required on the divisor input. This shift value is also used to initialize the counter on the same clock cycle as the loading of the shifted dividend as shown in Figure 2.8, being the number of clock cycles saved by the model. It might look that this value could be greater than 32, but this is not the case because of the input operand filtering inherited from the Basic model.

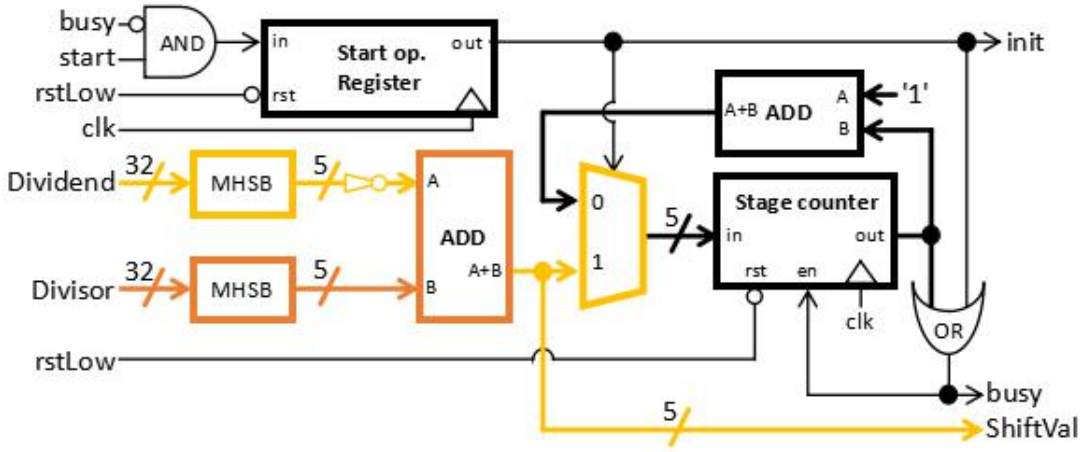

Figure 2.8: Block diagram of the starting stage calculation on the EQS model and the control block. The additional hardware by the QS model is highlighted in yellow and in orange for the EQS model. Components depicted in black are present on all models.

- **Double Even Quicker Start (DEQS) model**: it merges the EQS and Double models, inhering **the maximum latency of 18 clock cycles.** Because the EQS system can put the operation at an odd stage number, the control block commands a single subtraction when the operation is at the last odd stage of the execution (when counter value is 31), preventing that the number of iterations executed exceeds the 32 required by the restoring algorithm.

- **Skip Execution (SE) model**: by applying the EQS shifting system on both dividend load and mid-operation paths, the same **iteration's stage skip can also be achieved during the iterative execution**. The SE model guarantees to generate a '1' quotient bit at least every two iterations, fixing the latency of the operation to the number of '1' bits the quotient result has.

  This is performed by saving the most significant '1' bit position of the divisor on a 5-bit register during the loading of the shifted dividend. Then, both MSHB blocks inputs are redirected to be from the remainder and quotient registers, providing the leading '1' bit position of the mid-operation value. For each iteration, the SE model computes the number of shifts to perform on the remainder/quotient register in order to match the '1' leading bit to the divisor's, skipping an equal number of iterations to the number of left-shifts, as similarly done on the load of the dividend.

Even though the SE model does not implement additional logic on the value identification, since it re-uses the MSHB blocks from its predecessor the EQS model, it requires additional components on the control block to avoid exceeding the iteration counter over 32. This is calculated by computing, previous to the iteration execution, what is the value of the next iteration's counter, that is the previous counter value plus the shift value. On the case where it is over 32, the actual shift executed is the counter value minus 32 (the remaining iterations), otherwise it is the computed shift and then the counter is updated.

- **Double Skip Execution (DSE) model**: it is the model with the lowest latency among the developed in this project. It merges both SE and D models, inheriting the maximum latency of 18 clock cycles while also skipping iteration stages both at the load and during execution. To avoid executing over 32 iterations stages (16 clock cycles), it includes the same control block that the EQS, but with a counter limit of 31 to take into account the double stage execution per clock cycle.

The Table 2.2 summarizes the hardware required by each model. As it is shown, the optimizations implemented at each model increases the resources requirement, which translates into a higher resource usage and, most of the time, lower MFO as the synthesis data on the further section 2.6 demonstrates.

N = 32 bits	T	B	D	QS	EQS	DEQS	SE	DSE
MSHB blocks	0	0	0	1	2	1	2	2
32-bit multiplexers	2	2	3	2	2	3	2	3
32-bit shifters	0	0	0	1	2	2	2	2
33-bit SUB blocks	1	1	2	1	1	2	1	2
5-bit registers	1	1	1	1	2	2	2	2
32-bit registers	2	2	2	2	2	2	2	2
32-bit comparator	0	1	1	1	1	1	1	1
Max. latency (cc)	34	34	18	34	34	18	34	18
Qualitative MFO	Highest	High	Med.	Med.	Med.	Low	Low	Lowest

Table 2.2: Component requirements, maximum latency and qualitative MFO estimation for each division model.

Parts of these components are shown on the Figure 2.9 block diagram, indicating what iteration of the models evolution implements each component on the divider. The T model for example, implements the Figure 2.3 circuit on the "iterative execution logic" block, while also including the control block present on all division models. Then, the T model is expanded with the comparator logic, composing the B model, which is also shared on all the following division models.

Because the D model does not implement any operand's identification system, there is no additional hardware visible at the Figure 2.9 when compared to the B model. However, the D model expands the "iterative execution logic" block internally with the additional 33-bit subtracter listed by the previous Table 2.2.

From here, the model QS adds a MSHB block to shift the dividend to the MSB of the internal register and initialize the counter at a higher iteration stage. The model EQS follows by incorporating an additional MSHB block to shift the dividend to the leading '1' bit position of the divisor, while the DEQS model also shares the same changes maintaining the D model hardware. At last, both SE and DSE models add the multiplexers to allow the identification of the leading '1' bit on the mid-operation operands to jump iteration stages during execution.

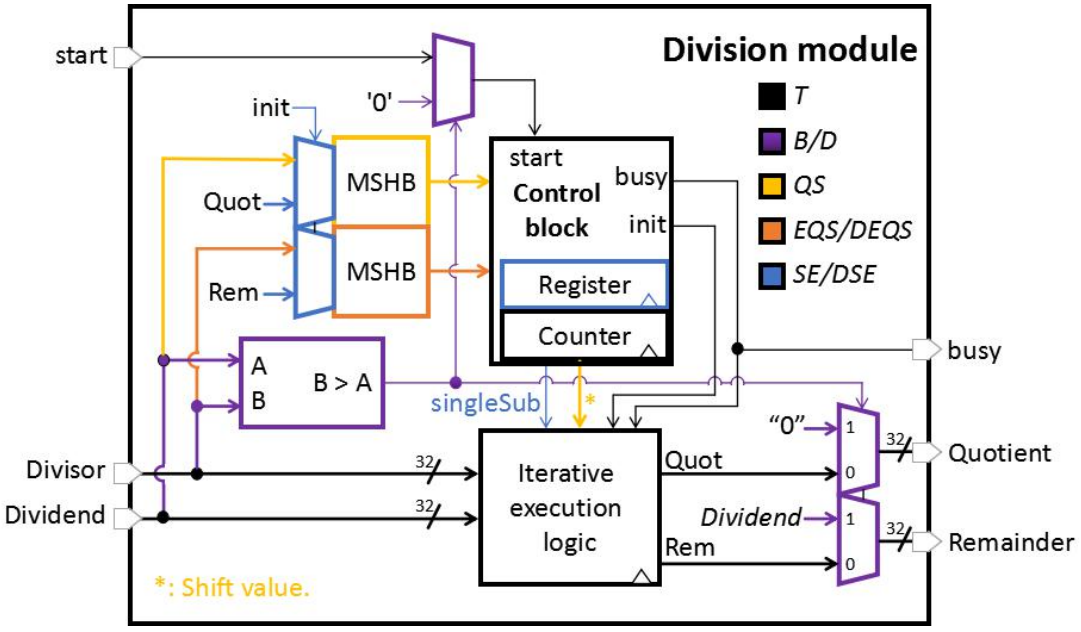

Figure 2.9: Block diagram of the division module. It includes colour coding depicting what model optimization added each part of the additional logic. Only representing external elements of the iterative execution logic block and control block, which also are internally modified by the optimizations of some models.

In order to understand better the capabilities of each division model, the following Table 2.3 presents the execution of a hand-picked division example, showing the differences between the optimizations on action. This task is facilitated because all division models share the same iteration stages, which are indicated by the value of the counter. Meaning, if the counter value is skipped from 2 to 4, the execution has gone from the iteration stage 2 to the 4 in one clock cycle for that particular model in this division operation.

Starting by the T model, it loads the dividend on the first clock cycle indicated by *, where the counter is 0. This clock cycle is not part of the iterative execution yet, since the control block is at the state B of the state machine from Figure 2.4 (b).

It is at the next clock cycle, where the control block changes to state C and the counter increases, that the recurrent operation starts. This is the reason the # number starts by 1 and not 2, since the # number does not indicate the clock cycle of the total execution, but only the clock cycle of the iterative process for that particular model.

The T model executes the restoring algorithm step by step with each clock cycle, until at the iteration stage 31 (with same counter value), that the last iteration is executed on the next clock cycle with a counter of 0, making a total of 32 clock cycles on the iterative process for the T model. Then, the control block changes to the C state for a clock cycle omitted at the table to deliver the results. Adding this clock cycle plus the one used for the load of the dividend and those spent on the iterative process, the T model accumulates a latency of 34 clock cycles in this operation.

The B model shares the same behaviour, as the division operation does not fall in the case where the divisor is higher than the dividend. The D model, however, jumps the odd counter values, since executes twice the iteration stages per clock cycle.

The QS model loads a shifted dividend to set the leading ‘1’ bit of the dividend on the MSB of the quotient register, saving 4 iteration cycle on the load cycle. The EQS does it similarly, but setting the leading ‘1’ bit to match the divisor’s, saving 19 iteration cycles on the load at this and further models. The DEQS model, being the implementation that merges both D and EQS models, applies both behaviours. However, the shifted dividend has set the operation to start at an odd iteration stage. That is the reason the last operation only executes a single shift operation to return to an even iteration stage, saving a total of 25 clock cycles.

Remainder	Quotient	Counter	T	B	D	QS	EQS	DEQS	SE	DSE
00000000	0C001801	0	*	*	*	-	-	-	-	-
00000000	18003002	1	#1	#1	-	-	-	-	-	-
00000000	30006004	2	#2	#2	#1	-	-	-	-	-
00000000	6000C008	3	#3	#3	-	-	-	-	-	-
00000000	C0018010	4	#4	#4	#2	*	-	-	-	-
00000001	80030020	5	#5	#5	-	#1	-	-	-	-
00000003	00060040	6	#6	#6	#3	#2	-	-	-	-
00000006	000C0080	7	#7	#7	-	#3	-	-	-	-
0000000C	00180100	8	#8	#8	#4	#4	-	-	-	-
00000018	00300200	9	#9	#9	-	#5	-	-	-	-
00000030	00600400	10	#10	#10	#5	#6	-	-	-	-
00000060	0C008000	11	#11	#11	-	#7	-	-	-	-
000000C0	01801000	12	#12	#12	#6	#8	-	-	-	-
00000180	03002000	13	#13	#13	-	#9	-	-	-	-
00000300	06004000	14	#14	#14	#7	#10	-	-	-	-
00000600	0C008000	15	#15	#15	-	#11	-	-	-	-
00000C00	18010000	16	#16	#16	#8	#12	-	-	-	-
00001800	30020000	17	#17	#17	-	#13	-	-	-	-
00003000	60040000	18	#18	#18	#9	#14	-	-	-	-
00006000	C0080000	19	#19	#19	-	#15	*	*	*	*
00004000	80100001	20	#20	#20	#10	#16	#1	-	#1	-
00000000	00200003	21	#21	#21	-	#17	#2	#1	-	-
00000000	00400006	22	#22	#22	#11	#18	#3	-	-	-
00000000	0080000C	23	#23	#23	-	#19	#4	#2	-	-
00000000	01000018	24	#24	#24	#12	#20	#5	-	-	-
00000000	02000030	25	#25	#25	-	#21	#6	#3	-	-
00000000	04000060	26	#26	#26	#13	#22	#7	-	-	-
00000000	080000C0	27	#27	#27	-	#23	#8	#4	-	-
00000000	10000180	28	#28	#28	#14	#24	#9	-	-	-
00000000	20000300	29	#29	#29	-	#25	#10	#5	-	-
00000000	40000600	30	#30	#30	#15	#26	#11	-	-	-
00000000	80000C00	31	#31	#31	-	#27	#12	#6	#2	#1
00000001	00001800	0	**	**	**	**	**	**	**	**

Table 2.3: Iterative execution of dividend 0x0C001801 and divisor 0x00008001 at each division model, including both register's hexadecimal next values and iteration stage counter. (#): Clock cycle of the iterative execution at that model. (-): Iteration stage skipped. (*): Operand load cycle. (**): Last iteration.

At last, the SE and DSE models achieve to save 29 and 30 clock cycles respectively by also shifting the mid-operation operand to set the leading '1' bit to the divisor's. These two models differ on that the DSE model has the ability of executing two iteration stages per clock cycle, where the the SE model only executes single iteration stage per clock cycle.

The Table 2.4 summarizes the clock cycles saved at the load, during the iterative execution and the total latency of the Table 2.3 example for each model. On one hand, because the T, B and D models does not implement any dividend shifting at the load, they cannot save any clock cycle at the load of the dividend. However, even though the QS does implement it, the DEQS, SE and DSE implement an expanded version that allows to save an even greater number of clock cycles during the load, which is also translated to a higher resource requirement as previously mentioned by the Table 2.2. On the other hand, only the D, DEQS, SE and DSE models allow to save clock cycles during the iterative execution of the operation. However, both D and DEQS models achieve this by computing two iteration stages per clock cycle (the DEQS saves lower clock cycles at the iterative as cause of the higher number saved on the load). The SE model by shifting the mid-operation value and the DSE by both ways.

	T	B	D	QS	EQS	DEQS	SE	DSE
Clock cycles saved at the dividend's load	-	-	-	4	19	19	19	19
Clock cycles saved at the iterative execution	-	-	16	-	-	6	10	11
Total operation latency	34	34	18	30	15	9	5	4

Table 2.4: Number of clock cycles saved and total latency of the operation at each division model at the division operation example from Table 2.3.

Note that these numbers are unique to this operation. Other division operations will vary depending on the input operands on, to be specific, the QS, EQS, DEQS, SE and DSE models, due to these implement dynamic optimizations, thus, not having a fixed latency dependent over the operands. That is the reason the following section performs a benchmark to measure the latency performance on a simulated sample.

2.2.4 Benchmark of the division models

Because most of the optimizations applied on the different division models are dynamic, making the latency to depend over the input operands, the only way to measure and compare the improvements on the latency is to perform an extensive benchmark. This benchmark is executed after the designs are final and have been verified to avoid erroneous data collection, following the verification techniques presented on section 2.5.

The benchmark is performed by injecting random operands to the target model or DUT, as shown in Figure 2.10. A counter keeps track of the number of clock cycles needed to perform a division by checking on the *busy* flag from the divider. The value of the counter is saved for each operation and finally, the average number of clock cycles per division is computed.

The benchmark performs 10.000 divisions with 32-bit randomly generated pairs of operands, one for the dividend and the other one for the divisor. The generated operands are reproducible by using a seed, which ensures that all model benchmarks use the same stimulus. It should be mentioned that the benchmark filters any division by zero, since this particular operation is not handled by the divider and will be filtered by the MULDIV module explained on section 2.5.

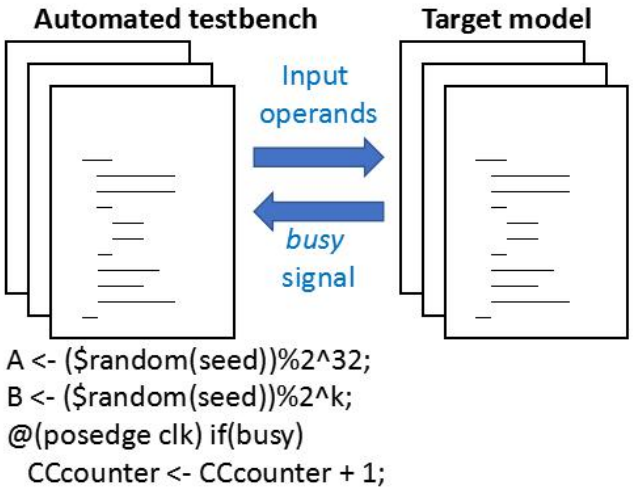

Figure 2.10: File hierarchy of the extensive sample benchmark simulation for the division models.

However, this sample provides deceptive data, since the random generation tends to input dividend and divisor operands of high value. It is considered as deceptive because the most advanced models maintain, on average, a latency much lower on divisor of high value than on lower. That is the reason the benchmark is further extended to include 10.000 division operations for each bit-width value for the divisor, limiting the randomly generated values to a maximum of 2^k-1 (being *k* the value's bit-width), while maintaining the dividend's random generation to 32 bits on all cases.

These averages are represented by the Figure 2.11 graph. It shows that the B and QS models perform similarly on latency average, being reasonable data as the QS model performs better on low value dividends. However, this is not benchmarked to not complicate the data any further, since the QS model is the only divider with this property. The D model performs almost half of the B model, which is reasonable since it executes the double of iteration stages per clock cycle than the B model.

Furthermore, the EQS model draws a straight line indicating that the latency average is proportional to the divisor's bit-width for this model. This makes sense, as the EQS model shifts only on the load cycle to match the divisor, thus, making the latency depend exclusively on the divisor value. The same happens for the DEQS model, but having a maximum latency limited to 18 instead of 34 on the case of the EQS's.

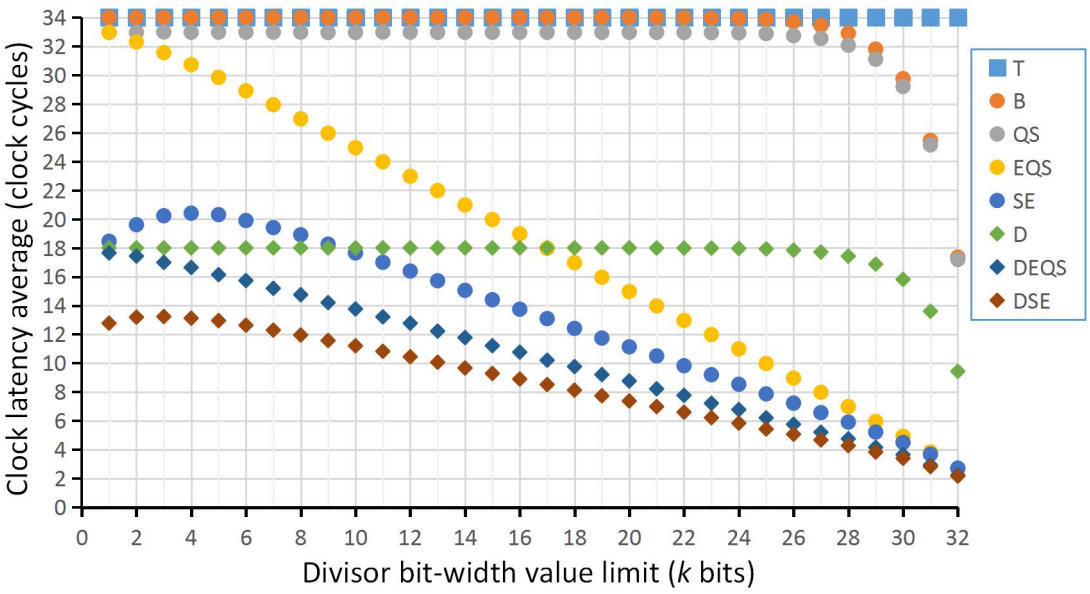

Figure 2.11: Benchmark results of the division models for each divisor bit-width value limit. The lower the clock latency is, the better the model performs. The dividend bit-width is fixed at 32 bits for all cases.

At last, both SE and DSE models are the most interesting, having a maximum latency average around 20 and 15 clock cycles at 4 and 3 divisor's value bit-width respectively. The reason behind this result is unknown, but it is probably related to the bit-width of the dividend, making an inflexion point which the dynamic optimization works worst.

The following Table 2.5 provides the global latency average of all the division models, obtained from performing the average of averages at equal parts. With these global latencies, it can be calculated the performance latency gain for each model respect the T model, since it is the one that implements the restoring algorithm step by step. This gain on latency performance is the final result from the analysis of this benchmark data, which it is further used to decide what model to implement on the MULDIV module on section 2.4.3, along with the other figures of MFO and resource usage.

	T	B	D	QS	EQS	DEQS	SE	DSE
Global LA (cc)	34	32.9	17.4	32.0	18.3	10.4	12.9	8.5
Lat. perf. ratio (%)	100	103.2	194.7	106.2	185.0	326.6	261.8	398.5

Table 2.5: Global Latency Average (LA) for each division module from the benchmark data, including also its latency performance gain in percentage respect Theoretical model.

2.3 Integer multiplier

The multiplier block MUL provides the 64-bit product of two 32-bits operands, being an essential component to the MULDIV module. It is based on the algorithms presented on the following section 2.3.1. Its implementation, contrary to the division's, it is not designed to operate on a closed loop for multiple iterations, but in one clock cycle. This characteristic is made feasible using Intellectual Property (IP) cores, explained on section 2.3.2.

2.3.1 Multiplication algorithms

There are two multiplication algorithms, named Wallace's and Dadda's trees. The implementation difference between these does not represent a noticeable difference when the target platform is a FPGA. However, introducing them provides the reasons to use IP cores.

Both algorithms start at a step similar to the multiplication method carried out by hand in decimal operations. First, a matrix is generated by AND functions of each bit between operands, resulting in 32^2 = 1,024 AND gates for a 32-bit multiplication.

This matrix must respect the bit position of each AND function by setting the result to the correct bit position, as shown at the stage 0 of Figure 2.12. These values, named partial products, are required to be added in order to provide the final result. The hardware placement to execute these additions is where the Wallace's and Dadda's algorithms diverge.

The Wallace's algorithm was first presented on 1964 [23] as a method **to increase the MFO** of the multiplication circuits by managing these additions optimally. The algorithm indicates the additions to perform on each stage are limited by the number of rows or height of the next stage, which the following expression $r_{i+1} = \lfloor 2r_i/3 \rfloor + r_i \ mod\ 3$ with a $r_0 = N + 1$ computes, N being the bit-width of the input operands. On the example of Figure 2.12 (a) these are $r_0 = 9$, $r_1 = 6$, $r_2 = 4$, $r_3 = 3$ and $r_4 = 2$, fixing 4 stages of additions.

The Dadda's algorithm presented on 1965 [24] **minimizes the number of hardware resources**, limiting the height of posterior stages to $d_{j+1} = \lfloor 1.5 d_j \rfloor$ with $d_1 = 2$, generating the series 2, 3, 4, 6, 9, 13... The next stage height is fixed by the value below N where it is situated on the series, which for $N = 8$ as the Figure 2.12 (b) example, the additions must reduce the height of the matrix to 6 for the next stage, then 4 and so on until the product result is obtained.

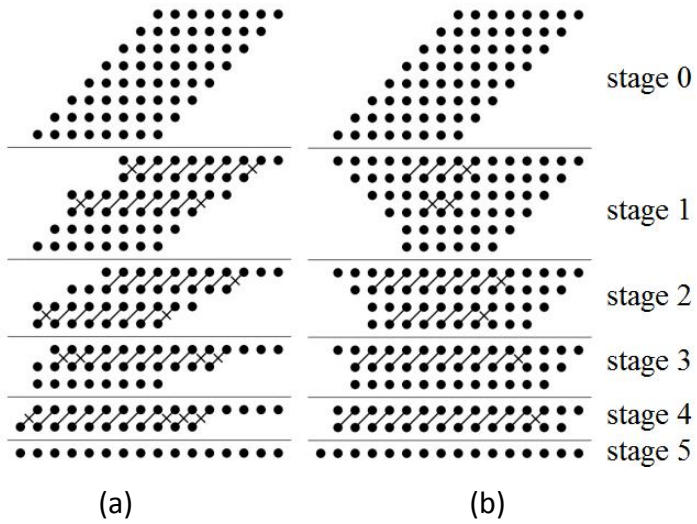

Figure 2.12: Dot diagram trees on Wallace's (a) and Dadda's (b) multiplication algorithms for a 8x8 bit-width multiplication example [25].

The additions are commonly implemented with half-adders (2,2) and full-adders (3,2), even though more complex components named accumulators and Booth encoders are used on the present-day, providing circuits with lower gate delay and area [26].

The implementation of these circuits using LEs on a FPGA produces multipliers with an extremely low MFO and consumes many resources, being indifferent what algorithm is being based on. To avoid this limitation, FPGA manufacturers include on their platforms dedicated logic cells to perform multiplications. These use Digital Signal Processing (DSP) units in conjunction to LUT for storing coefficients used on the operation. This composition is optimal, since they have been designed without the programmable gate-array structure internally, offering circuits with higher MFO than the same circuit implemented through LEs.

The use of embedded cells in HDL programming is achieved thanks to a synthesis program compatible with the FPGA platform in question. Additionally, synthesis tool manufacturers provide licences to use intellectual properties that achieve best performance on using these cells. The following section specifies the IP availability on multiplication IP cores to enable the use of these cells for a 32-bit multiplier.

2.3.2 IP core availability for an integer multiplier

To enable the use of embedded resources on a FPGA platform optimally, the use of IP cores is encouraged to generate modules that perform specific operations. The availability of these depends on the FPGA model, family, manufacturer and the IP licences provided.

In our case, Intel provides a number of IP cores with the entry-level Quartus II 13.1 Web Edition software under its Megafunction library. These license-free IP cores are allowed in the prototyping stage of hardware development on the supported platforms, which apply a time limit restriction on the programming of the FPGA. In order to use these IP modules on final commercial products, the purchase of a full production license is required, increasing the cost of production.

On the “Integer Arithmetic IP Cores User Guide” [27] from Altera (subsidiary of Intel) is listed all the pertinent IP cores related to integer multiplication operations compatible with our prototyping platform. From these, only the ALTMULT_ADD and LPM_MULT IP cores meet our requirements.

To generate a working module using these IP cores that adjusts to the requirements (bit-width inputs, clock latency, input and output registration, etc), the synthesis software includes the “MegaWizard Plug-In Manager” for this same purpose. Figures 2.13 and 2.14 show the options applied on the generation of both multiplication modules.

The tool provides a resource usage estimation indicated on the lower left corner in these figures, which both uses eight DSP 9-bit multiplication embedded cells of the FPGA prototyping platform. The pipelining option would increase the MFO by making it operate in multi-cycle mode (increase latency), but it has been decided that **the multiplication operation must take one clock cycle**. This design decision fixes the critical path of the instruction set M package to be the multiplier path, as these embedded cells carry a substantial gate delay. The performance impact of this is further analysed on section 2.6.

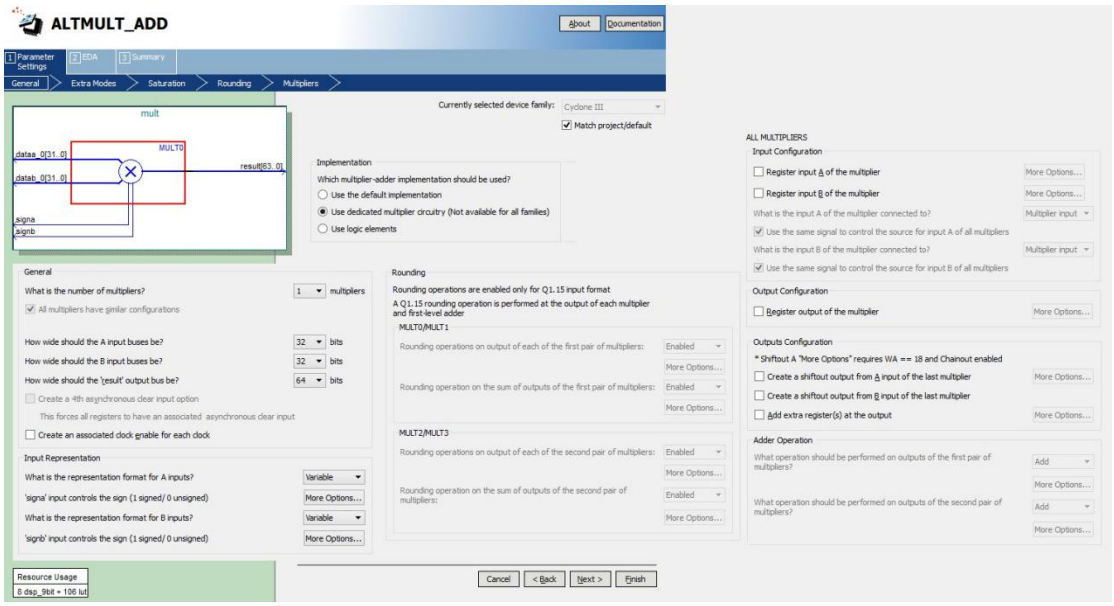

Figure 2.13: IP MegaWizard tool options for ALTMULT_ADD IP module.

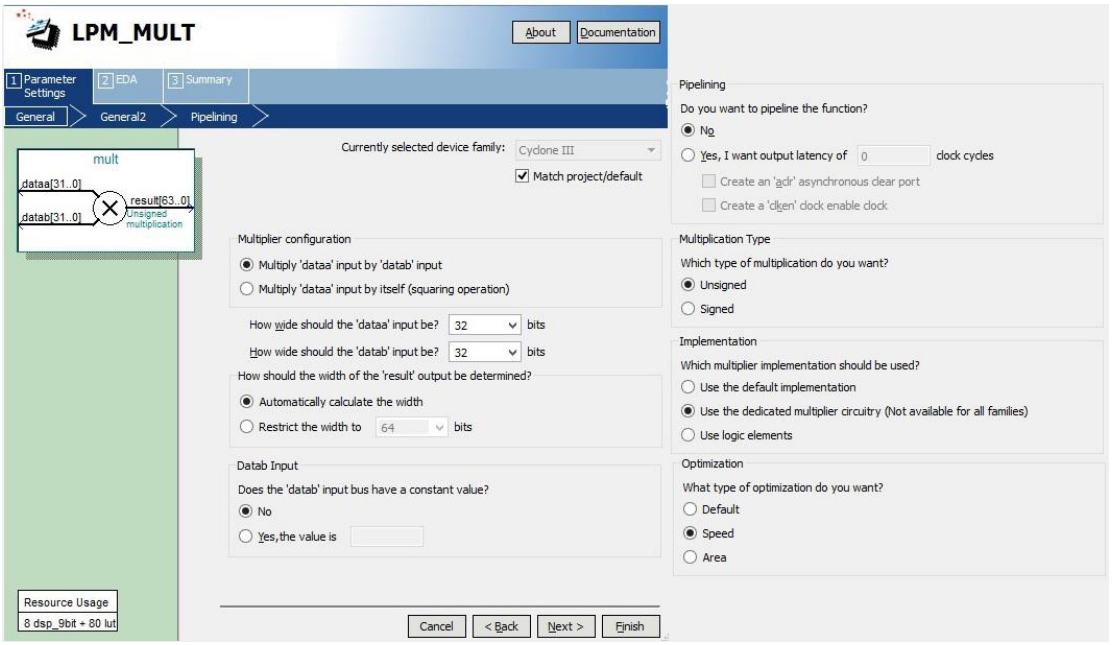

Figure 2.14: IP MegaWizard tool options for LPM_MULT IP module.

A distinction between both IP modules is that the ALTMULT core implements a multiplier capable of operating on both signed and unsigned operands, characteristic that allows us to design an alternative MULDIV configuration different from the Figure 2.1. This is further explored on the next section where the MULDIV is developed.

2.4 MULDIV module

The purpose of the MULDIV module is to encapsulate both division and multiplier modules in one package that fully execute the operations of the extension set M. This includes the ability of performing unsigned and signed division and signed, unsigned and signed x unsigned multiplication operations.

On one hand, the compatibility between all the division models facilitates the design of the MULDIV, since all the division models require the same signed/unsigned to unsigned/signed conversions at the input/output during signed operations. This is achieved by taking advantage from the 2's complement binary system, that allows to compute the singed/unsigned conversion by negating and incrementing the unsigned/signed operand.

On the other hand, the LPM and ALTMULT multiplier modules do not share this compatibility, being the ALTMULT module unique on its ability to execute all the required modes by itself. This is the reason to design two MULDIV models; **the MULDIV1 to adequate the implementation of the LPM multiplier** (following the design topology from Figure 2.1) and **the MULDIV2 to the ALTMULT's**.

Figure 2.15 shows the module design, where the LPM specific logic is exclusive on the MULDIV1 model and ALTMULT's is exclusive on the MULDIV2 model, differing on that the ALTMULT multiplier bypasses the conversion logic. This allows higher MFO on the MULDIV2 over the MULDIV1, since the LPM multiplier with the conversion logic is a path more constrained on gate delay than on the ALTMULT. This is further studied on the synthesis data from section 2.6.

Starting by the control logic, the input bus *ALUop* indicating what operation to execute is decoded in a number of control signals at the Operation decoder, which are used to signal information about the operation. This is further shown by the truth table from Table 2.6. For example, the *signedA* and *signedB* signals indicate those input operands are signed at '1,' enabling the required conversions to the divisor and multiplier on the MULDIV1 model or only the divisor on the MULDIV2 model.

Instruction	MUL	MULH	MULHSU	MULHU	DIV	REM	DIVU	REMU
ALUop	000	001	010	011	100	101	110	111
divisionOp	0	0	0	0	1	1	1	1
remSel	X	X	X	X	0	1	0	1
Hbits	0	1	1	1	X	X	X	X
SxU	0	0	1	0	X	X	X	X
signedA	1	1	1	0	1	1	0	0
signedB	1	1	0	0	1	1	0	0

Table 2.6: Control signals generation by the Operation Decoder depending on the ALUop value that indicates the operation instruction to execute.

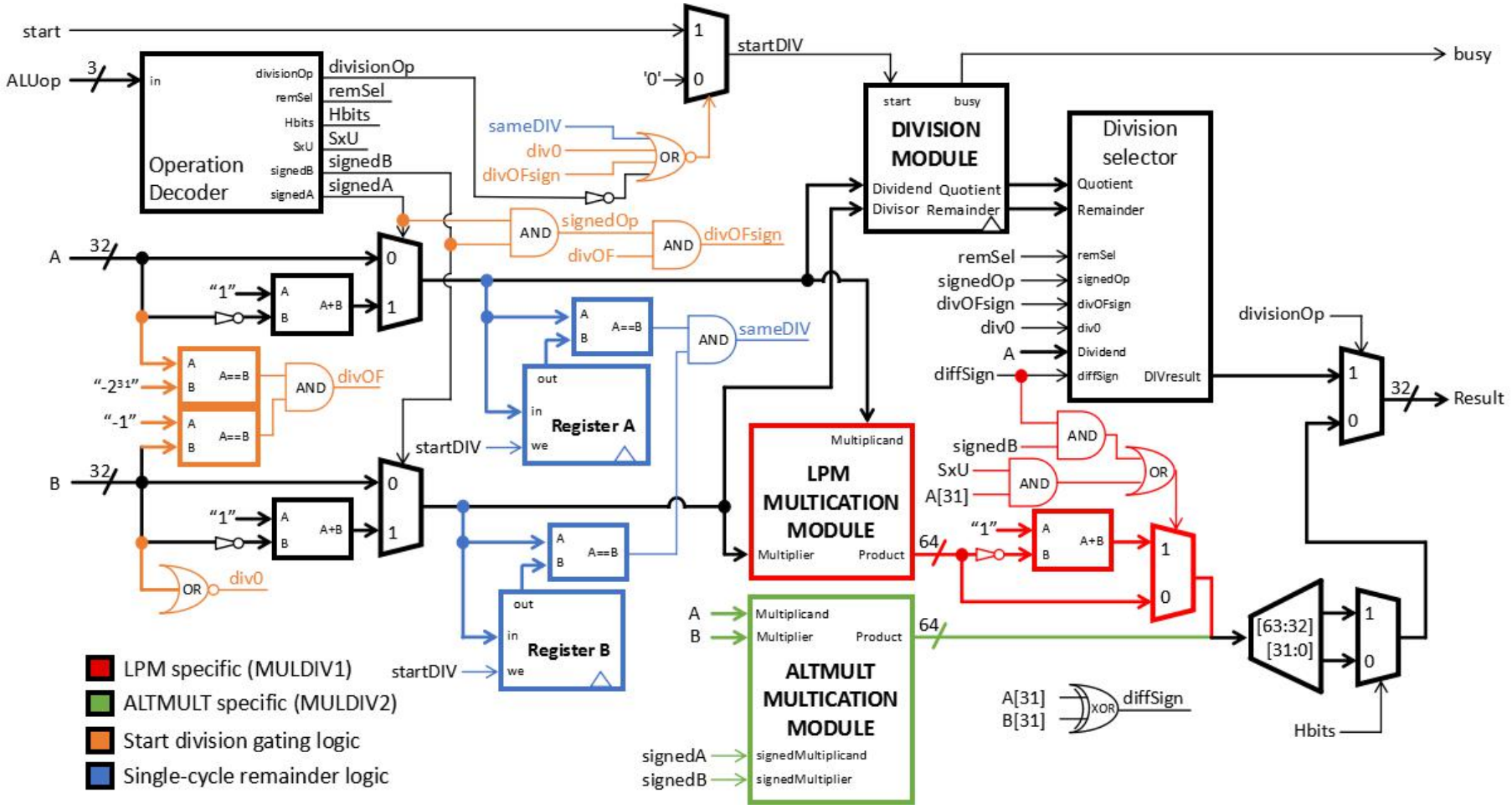

Figure 2.15: Module MULDIV schematic diagram. LPM and ALTMULT hardware are only specific to the MULDIV1 and MULDIV2 models respectively. On the other hand, start division gating and single-cycle remainder circuits are on both models.

The *SxU, Hbits* and *remSel* control signals indicate information about the output, being a signed x unsigned operation, output the 32 high bits from the multiplication and select the remainder output respectively. However, the *divisionOp* signal is used to gate the *start* input from the divider to avoid starting the execution of a division at a multiplication request or at a case where it is not required to operate.

These cases include same operands respect the previous operation, division by 0 and division overflow. The latter is only relevant at signed operations and occurs when the dividend -2^{31} (0x8000 0000) is divided by -1 (0xFFFF FFFF), since the quotient result 2^{31} cannot be represented by a 32-bit operand on the 2's complement format.

Furthermore, the MULDIV block manages the sign of the divisor's output, as the divisor module operates on unsigned operands. This is done at the "division selection" block by inputting the required control signals, providing the sign on 2's complement format of the Table 2.7.

Dividend	Divisor	DIV (quotient)	REM (remainder)
Positive	Positive	Positive	Positive
Positive	Negative	Negative	Positive
Negative	Positive	Negative	Negative
Negative	Negative	Positive	Negative

Table 2.7: Result sign selection for the division module. DIVU and REMU operation modes omitted since they take always the raw division module outputs.

The division selection block also output the proper result on division by zero or overflow cases shown by the Table 2.8 that are set by the RISC-V ISA. This is 0xFFFF FFFF as quotient and the dividend as remainder on division by zero and 0x8000 0000 as quotient and 0 as remainder on division overflow.

Operation case	Dividend	Divisor	DIVU	REMU	DIV	REM
Division by 0	X	0	$2^{32}-1$	X	-1	X
Overflow	-2^{31}	-1	-	-	-2^{31}	0

Table 2.8: Special division cases outputs. Fixed by the ISA manual [5], page 45.

The implementation of this Division selection block is quite simple through several multiplexers controlled by the input flags generated by the Operation decoder or the identification of the input operands that are also used to gate the *start* input. It is shown in Figure 2.16.

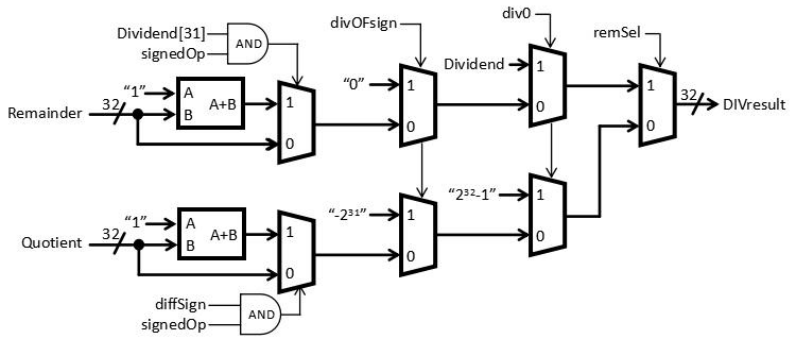

Figure 2.16: Block diagram of the division selector internals.

Similarly, the LPM multiplier on the MULDIV1 model also requires additional logic to compute when it is required to output the negative signed conversion of the result to follow the Table 2.9 sign rules. These are controlled by identifying when the operands differ on sign during the signed operations of MUL and MULH or the multiplicand "A" sign on MULHSU (signed x unsigned) operations.

However, this logic section is not required by the ALTMULT multiplier on the MULDIV2 model, since this multiplier already allows to set the format of the input operands through the *signedA* and *signedB* signals, saving resources.

Operation	Multiplicand	Multiplier	Result
MUL MULH	Positive	Positive	Positive
	Positive	Negative	Negative
	Negative	Positive	Negative
	Negative	Negative	Positive
MULHSU	Positive	Any	Positive
	Negative	Positive	Negative
	Negative	Negative	Negative

Table 2.9: Result sign selection for the LPM multiplication module. MULHU operation always takes the direct result from the operation module.

To ensure that every module meets the specifications set during design and there are not oversights that could result in erroneous execution, both MULDIV models and its components have been through a verification process explained on the next section.

2.5 MULDIV verification

The verification process allows to ensure that the designed hardware is working at the required specifications, that in this case are fixed by the ISA and external modules. This includes signal timings of the Figure 2.2, outputs and overall, every internal execution that may have an unintended effect during the operation.

The verification consists on injecting random operators to the DUT and to a reference module. The response of both blocks is compared. If they do not match, an error is generated. At the end of the test, the user checks if the test was successful. If not, it is necessary to find the bugs in the DUT. In that case it would be necessary to lot some waveforms, like the shown at Figure 2.17, to detect the bug on the data path of the design and rectify it on the Verilog HDL code of the DUT.

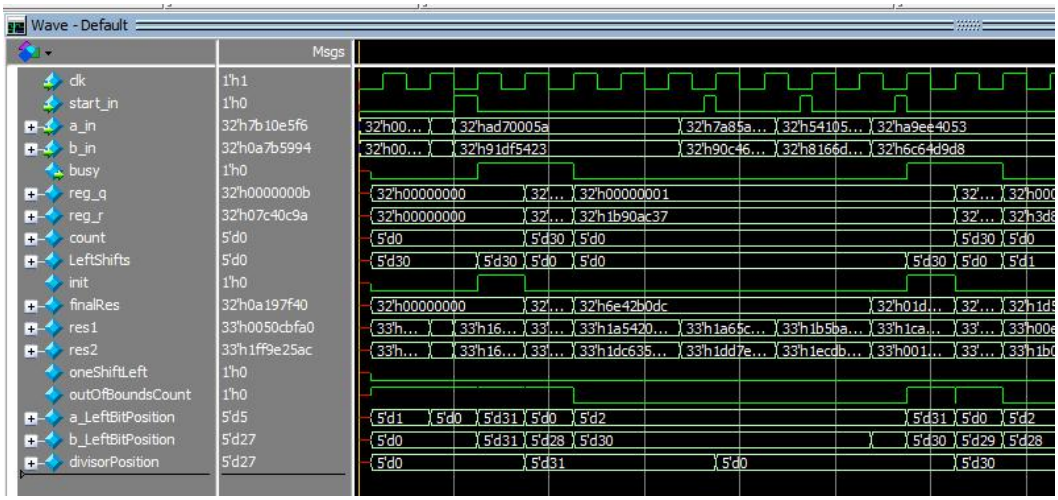

Figure 2.17: Waveform graph from a simulation on a DUT for manual verification.

Even though this method is enough for specific input cases, as for testing on special stimulus like a division by 0 output the proper result, it is inefficient in order to confirm a near complete correct function. This is why the test files run automated

routines injecting random and predetermined stimulus vector of interest, providing information about the robustness of the developed hardware.. The specific execution of these test is depicted on Figure 2.18.

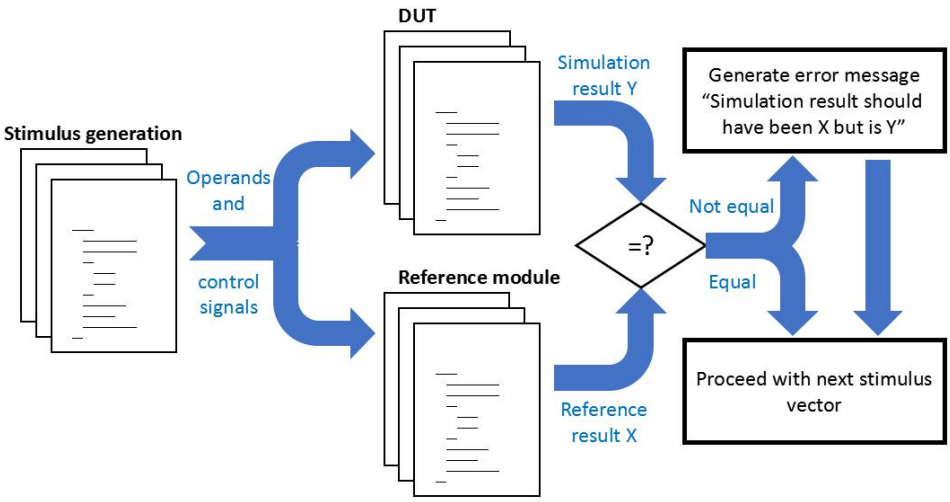

Figure 2.18: Scheme of the automated test execution for module verification.

The reference module is implemented by non-synthesizable code, easier to design and to prove correct functionality over the DUT synthesizable code.

The stimulus injection is not limited to operands. It also includes control signals used to change the operation request and any signal the DUT's input allows (clock, reset...). These must be also reflected on the reference module to carry out the operation being tested on the DUT.

These test files can also implement routines to count clock cycles used at the determining of the clock latency average on a particular operation, as done previously on the division benchmark. Other performance figures of implementation area or MFO cannot be obtained through this process, as the simulation does not imply synthesis of the hardware other than logic function. These must be estimated by a synthesis tool as it is done on the next section.

2.6 Synthesis on a FPGA of the extension set M modules

The hardware designed in this project is developed by writing Verilog HDL code. However, in order to obtain any circuit on bare-metal, be it on FPGA or ASIC, the code is required to be processed by a synthesis program. In our project, this synthesis program is Quartus 13.1, the last tool version that supports the FPGA model EP3C16F484C6N from the Cyclone III family where this project prototype on.

This software program first runs a compiler tool to detect any syntax problem at the DUT's code. Then, an assembler is executed to mount the hardware available at the FPGA model to create the same data paths that of the DUT's, efectively implementing the logic on physical hardware resources. Finally, a fitter tool manages to find the optimal disposition of the hardware to provide the routing with the lowest resource, lowest power consumption or highest MFO implementation possible. This is controlled by a number of optimization options available at the synthesis program used to prioritize for one during synthesis.

Since the synthesis process set the physical hardware that implements the DUT, the data regarding the number of resources (LEs) required to synthesize and the MOF it can reach is generated. However, this data is not only depending on the model of the target platform (because different families and models include unique resources), but also these optimization options at the synthesis program. These options have been left by default on this project to limit the number of variables affecting the synthesis data.

All the combinations of both MULDIV models with each divisor model has been synthesized, providing the synthesis data of the number of LEs used and the MFO it implements presented at Figure 2.19. The MFO value is generated for both 0 and 85 degrees Celsius (ºC) due to the capacitance of the device increases with the temperature, lowering the MFO in turn.

Analysing this data, a number of characteristics about the modules can be extracted. Starting by the MFO, the MULDIV2 module implements a higher MFO than its MULDIV1 counterpart. The reason being that the multiplier data path on the MULDIV2 bypasses the signed/unsigned conversion blocks. However, when using

both SE and DSE division models, the MFO decreases to a level comparable to the MULDIV1. This is only explained by how the complexity of these models increase the data path of the division operation to a point where it surpasses the multiplier, localizing a critical path more constrained and thus, lower MFO.

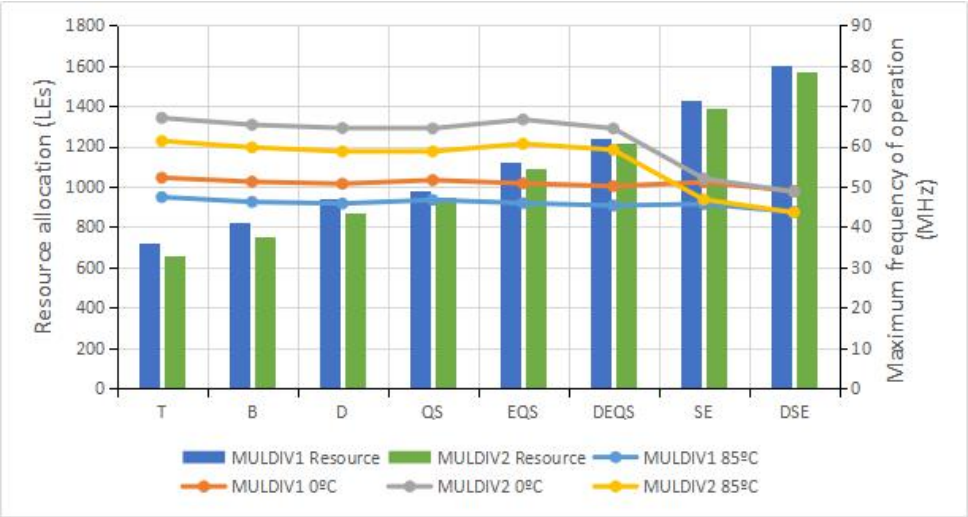

Figure 2.19: Resource usage in LEs and MFO in MHz of all MULDIV combinations. Higher MFO and lower resource usage is better.

This difference on how both MULDIV models implement the multiplier block is also reflected by the lower number of resources used on the MULDIV2 when compared to the MULDIV1.

In order to use this data to recommend a definitive module combination, the following Figure of Merit (FoM) has been created. This FoM combines both resource usage and MFO from the synthesis data with the latency performance on division operations previously presented on section 2.2.4. It has been given 3, 2 and 1 weight points of importance to the MFO, resource usage and latency performance, due to the frequency of operation affects the core performance for all operations, making it more important over the other two figures.

$$\text{FoM score} = \left(\frac{3\,x\,\text{MFO}\,\%}{100} + \frac{2\,x\,100}{\text{Resource}\,\%} + \frac{1\,x\,\text{Latency Performance}\,\%}{100} \right) \frac{100}{6}$$

The FoM score shown at Table 2.10 is computed by using the figures ratios respect the MULDIV1 with the T model divisor. By using ratios exclusively, only the FoM weights and the actual performance figures of the models over the reference affect the FoM score.

The table indicates that at this FoM rating, the MULDIV1 with the DSE model and the MULDIV2 with the DEQS model are the most optimal implementations, being the MULDIV2 over the MULDIV1 for higher MFO. However, the decision between both is limited by the IP license availability for the multiplier circuit.

Although different considerations from the applied FoM also varies the decision factor, if for example, the resource usage is more important than the MFO. This can be modified by applying the corresponding weights on the FoM formula without requiring any new data.

	Perf. figure	T	B	D	QS	EQS	DEQS	SE	DSE
MULDIV1	MFO	100	97.8	96.9	98.7	97.1	95.8	97.2	93.0
	Resource	100	113.2	129.8	134.9	154.6	171.4	197.5	220.6
	LP div. op.	100	103.2	194.7	106.2	185.0	326.6	261.8	398.5
	FoM score	100	95.5	106.6	91.8	100.9	121.8	109.1	**128.0**
MULDIV2	MFO	119.7	117.0	115.7	116.5	117.9	115.0	103.5	98.0
	Resource	91.0	104.3	119.7	130.9	150.2	167.3	192.0	216.1
	LP div. op.	100	103.2	194.7	106.2	185.0	326.6	261.8	398.5
	FoM score	113.1	107.7	118.1	101.4	112.0	**131.8**	112.8	130.8

Table 2.10: MFO, resource usage and latency performance on division operations ratios (%) of all MULDIV module combinations respect MULDIV1 with T model implementation. FoM score included with 3, 2, 1 weights for MFO, resource usage and latency performance respectively.

3 | Extension Instruction Set F

This third chapter focus on the hardware design of the Floating-Point Unit (FPU) module and its sub-modules, being these the circuits that provide support to floating-point arithmetic operations. Due to time constrains, the implementation only allows single-precision addition, subtraction, load and store operations. However, this project sets the framework to expand the functionalities to fully support the set F.

Other essential components for the extension set F support external from the FPU, which includes a dedicated file register for floating-point operands, a specific Control and Status Register (FCSR) and the additional core infrastructure to decode these instructions and to execute the load and store on this dedicated file register are explained on section 4.3.

The instruction set F hardware is also used as a base and it is a requirement for the implementation of the extension sets D and Q for double and quad floating-point format operations respectively. However, this project only implements support for the instruction set F, that is to support operations between single-precision operands.

The Figure 3.1 shows a simplified view of the FPU module. The three 32-bit input operands A, B and C pass through an input check block to ensure that they are floating-point values and do not fall under special cases. Then, the operation is executed on the specific sub-module, the data-path continues to a rounding step and finally, an output check similar to the input one.

The FPU control unit decodes the *ALUop,* checks if there are special input cases, redirects the *start* input signal to the proper sub-module, sets the correct rounding mode (if required by the operation) decoding the *frm*, sets to high both the *busy* output flag to indicate multi-cycle operation is ongoing and the exception flags encoded on the output bus *fflags*. These last can be triggered by several situations, which are explained on section 3.3.

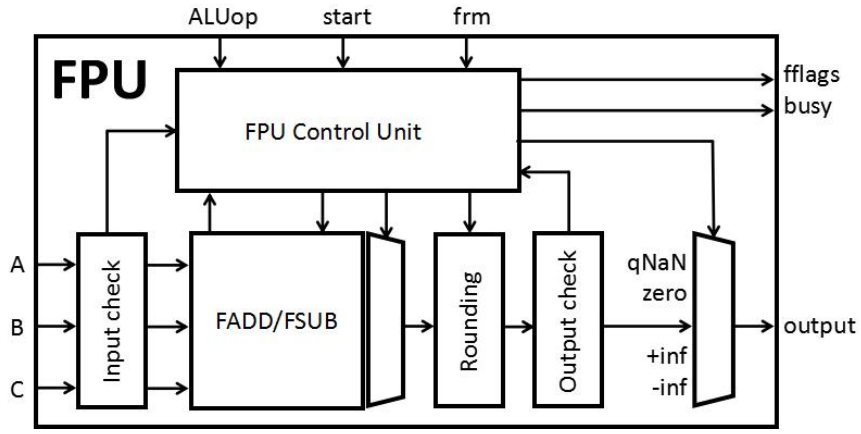

Figure 3.1: Block diagram of the Floating-Point Unit module.

The single-precision floating-point format is complex and has its own characteristics that must be considered during design. The format is described in section 3.1. Sections 3.2 and 3.3 describe the design of the addition and subtraction sub-module and the FPU that wraps it respectively. Finally, sections 3.4 and 3.5 present the limitations on the verification stage and the synthesis of this hardware.

3.1 Single-precision floating-point format introduction

The single-precision format supported by the RISC-V ISA is specified by the IEEE 754-2008 standard [28]. Every single-precision operand contains 3 fields: a sign bit, 8 exponent bits and 23 mantissa bits, composing a 32-bit word as shown in Figure 3.2. These operands can be also converted into a decimal representation and to the other way around, calculation used on the verification process of the designed hardware.

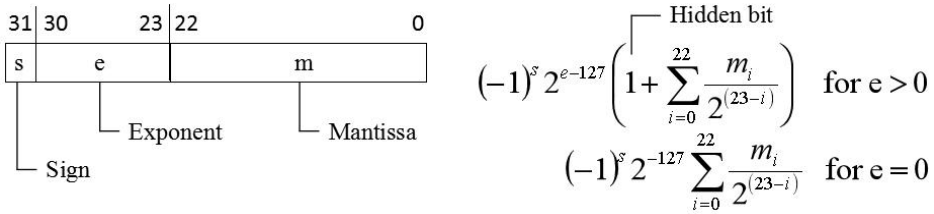

Figure 3.2: Single-precision floating-point format fields and its decimal value conversion for regular (e>0) floating-point and subnormal (e=0) value types.

The single-precision format has a limited maximum value of $3.4 \cdot 10^{38}$ and a minimum value of $1.4 \cdot 10^{-45}$ at normalized decimal conversion, excluding both infinite and zero representations. Of course, the range and precision can be increased by changing to formats with higher number of bits, like the double or quad-precision formats which use 64 and 128-bit operands respectively.

The format standard also includes a number of specific encoding values, summarized in Table 3.1. The signalling "Not a Number" or sNaN type is often used on advanced features, which the RISC-V non-privileged ISA does not support at the moment. As for the quiet NaN or qNaN type, it is generated as result at most arithmetical floating-point operations when an invalid operand has been used as input or the operation is invalid. Both positive and infinite types are used on results from operation overflows.

Type of value	Binary encoding (sign_exponent_mantissa)
Signaling NaN (sNan)	`X_11111111_XXXXXXXXXXXXXXXXXXXXXX`
Quiet NaN (qNan)	`0_11111111_10000000000000000000000`
Infinite	`X_11111111_00000000000000000000000`
Zero	`X_00000000_00000000000000000000000`
Regular FP value	`X_XXXXXXXX_XXXXXXXXXXXXXXXXXXXXXX*`
Subnormal or tiny value	`X_00000000_XXXXXXXXXXXXXXXXXXXXXX`

Table 3.1: Type of values and its bit representation allowed by the single-precision floating-point format. *: The exponent bits must be greater than one at least and lower than full ones to differ against other type values.

A first property to consider from the format is that it uses a sign bit, so there is **no compatibility with two's complement binary system**. This feature increases the complexity of the circuit design, because it adds a logic requirement between sign bits operands on simpler operations prior to the execution. A '0' sign bit indicates a positive floating-point value, where a '1' sign bit indicates a negative floating-point value. There are two possible representations for the zero value, however, the positive zero is indistinct from the negative zero for most floating-point arithmetic operations.

A second property is the **discrimination between zero and non-zero exponent operands**. The non-zero exponent or "regular" values take as granted a '1' MSB mantissa bit that is not present on the mantissa field called "hidden bit" in order to save a bit position on the representation. This is not the case for a zero exponent or "tiny" values, also called subnormal values, where this hidden bit is computed as '0'. The coexistence between tiny and regular floating-point values in the format further increases the logic complexity of the hardware design.

A third property is that this format is analog to the scientific decimal notation, so it can be translated to calculate and verify the operation is correct. However, the single-precision format bit-width limits the precision that the operand is able to represent depending on the exponent value. For example, the maximum mantissa value possible at 0x7F exponent (0 in decimal) is 1.99999988079 in decimal, $1.19 \cdot 10^{-7}$ short from a value of 2. In comparison with the next exponent 0x80 (1 in decimal), the maximum mantissa value is 3.99999976158 in decimal, $2.38 \cdot 10^{-7}$ short from a value of 4. It can be concluded with this example that with each binary exponent increment, the precision of the operand can represent halves, as shown in Figure 3.3. Therefore **floating-point operations may incur in a loss of precision**.

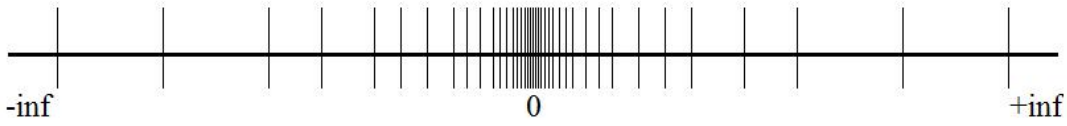

Figure 3.3: Floating-point precision representation depending on the value it stores. Any value out of the format representable limits falls under the "out of bounds" classification.

To address the cases where arithmetic operations incur on loss of precision, the RISC-V ISA fixes some rounding modes in conjunction with the exception flags. These are described in more detail in section 3.3. Despite of these do not fully solve the limitation of the format, they provide the means to control and manage them on software.

3.2 Addition and subtraction sub-module

The addition/subtraction sub-module executes the arithmetical addition and subtraction operations between two single-precision floating-point operands. This includes zero, infinite, tiny and regular values but also excludes the special operation of infinite minus infinite.

Contrary to regular binary encoding, the floating-point format is enough complex to require equally complex algorithms to execute, what would be otherwise, a simple operation like an addition. The algorithm applied can be summarized in 3 steps: exponent alignment between input operands, operation execution (addition or subtraction) and mantissa alignment.

Step 1. Exponent alignment: In order to be able to add or subtract both mantissa operands, first they have to be exponent aligned. This is computed by finding the difference between both operand exponents to apply an equal number of right-shifts to the mantissa of lower exponent.

The number of shifts takes into account that, if one of the operands is a tiny value, the actual number of shifts to performs changes to be the difference between exponents minus one. The reason being the exponent decrease from “1” to “0” does not imply a mantissa shift, only the loss of the ‘1’ hidden bit. If both operands contain tiny values, the computed shift value is zero, because their exponents are aligned.

Previous to the mantissa shift, a MSB mantissa bit position is added, being the hidden bit. This bit is ‘0’ when the exponent of the operand is zero and ‘1’ otherwise. Then, the shift is executed and a only the 27 MSB bits are passed to the next step, with the highest exponent value and both operand sign bits as the example of the Figure 3.4 shows and the RTL schematic of the Figure 3.5 implements.

These 27 mantissa bits are, from most significant to least, the hidden bit, 23 regular mantissa bits and 3 bits used at posterior rounding step, named guard, round and sticky respectively. The sticky bit is special because applies an OR function to all the bits lost from the shift. These rounding bits indicate loss of precision when any is ‘1.’

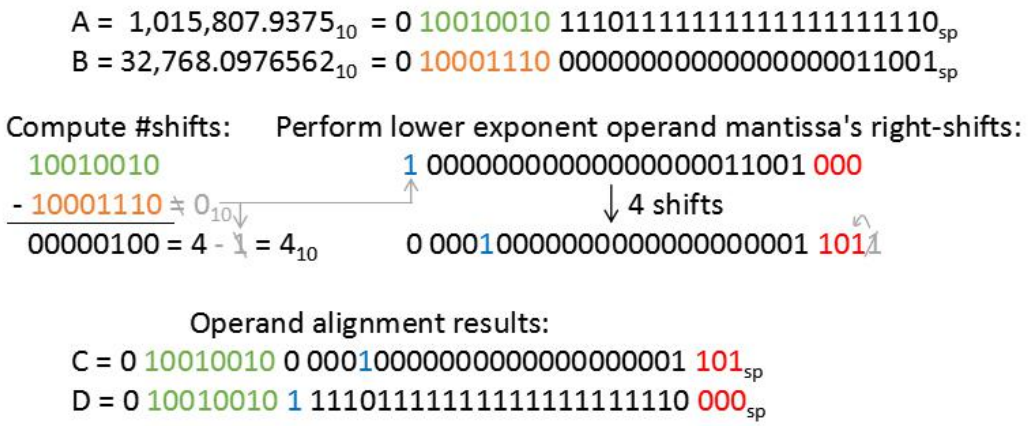

Figure 3.4: Exponent alignment example between two regular floating-point operands. Red indicates the additional three bit positions for rounding (guard, round and sticky). Blue indicates the hidden bit as result of the lower value operand containing a regular floating-point value (exponent greater than 0).

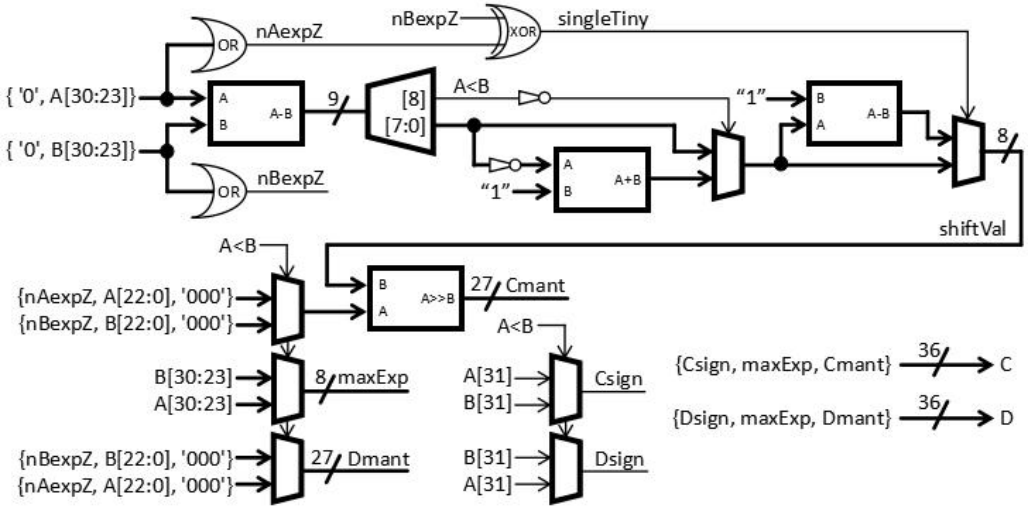

Figure 3.5: Operand exponent alignment schematic diagram.

Step 2. Mantissa operation: Because an addition instruction can incur on a subtraction operation if the input operands signs are different, this step computes both addition and subtraction between the exponent aligned mantissas. Note that the subtraction can only be performed on the operand with the greater or equal mantissa than the other one regardless of the sign, because the format is not compatible with two's complement.

To recognize if the actual operation to execute is an addition or a subtraction, an XOR function is performed to operands A, B sign bits and the *SUBflag* issued by the FPU control unit to generate the selector signal between the addition and subtraction results.

The mantissa result of the operation has an additional bit to contain the possible mantissa overflow from an addition. This is shown in Figure 3.6. This overflow bit is used on the following mantissa alignment step to update the exponent.

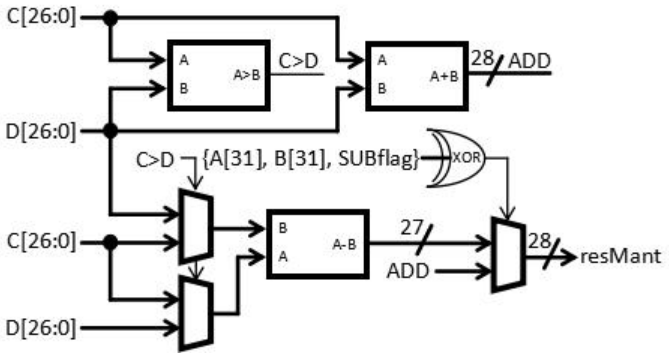

Figure 3.6: Mantissa addition/subtraction operation schematic diagram.

Step 3. Mantissa alignment: In order to return from the mantissa of the previous operation result to the mantissa field of the single-precision format, a mantissa alignment step by shift is executed.

This step searches to set the leading '1' bit from the mantissa to the 27th bit position (counting from least to most significant). This is performed by computing first where this leading '1' bit is using a MSHB sub-module, which has been presented on the previous chapter at the Figure 2.7. The bit-position minus 25 (because of how the MSHB reports the bit position) is the number of left-shift to set the mantissa on the required position.

Previous to the mantissa shift, the exponent is increased by one if the leading '1' bit is at the 28th bit position, stays without update if it is on the 27th or the number of left-shifts is subtracted from exponent if lower.

In the case where the number of left-shifts is greater or equal than the exponent, the number of left-shifts is updated to be equal to the exponent value minus 1 (as the exponent decrease from 0x01 does not represent a left-shift) and the exponent passes to be zero, since the format does not support the 2's complement. Then the left-shift of the mantissa is executed. This leaves the mantissa result to be from the 26th to the 1st bit position on most cases. However, on cases where the leading '1' bit is at the 28th bit position, the mantissa result is from the 27th to the 2nd bit position.

Due to the complexity of this sub-module is rather high, it can be anticipated that the MFO of the addition/subtraction sub-module will be limited. In order to increase this performance figure, the data path of this module is segmented in 2 pipeline stages by registering the end of the first.

The first stage includes both first and second steps, while the second stage includes exclusively the third step, since it is the most gate intensive from the sub-module. As trade-off, this pipeline topology increases the **latency of the addition/subtraction sub-module to 2 clock cycles**. The actual gain on the MFO is reviewed at section 3.3 with the FPU synthesis data analysis.

Lastly, to match input with the following rounding process, the mantissa output of this sub-module is modified by passing the last two bits through an OR function, that are the round and sticky bits to generate the new round bit. This is because the rounding step only requires two rounding bits, as the following section on the FPU explains.

3.3 Floating-Point Unit

The Floating-Point Unit (FPU) is the module that encapsulates the operation sub-modules, like the previous addition/subtraction sub-module, to execute the operations required by the extension set F. To achieve this, the FPU generates control signals for these sub-modules, adds a rounding step if necessary, manages input and output special cases and provides communication with external hardware, which includes exception signalling. Figure 3.7 shows a block diagram of the FPU.

The *ALUop* input bus indicating what operation must be performed at the FPU is decoded into the *FPop* bus. This bus is used to select between the different sub-modules results and to redirect the counter output signal to the sub-module that computes the requested operation. The selection circuitry is left mostly unused, as there is only a single sub-module implemented at the moment (the addition/subtraction sub-module).

The general counter uses the *start* input signal to load an initial value from the ROM table selected by the *FPop* bus. This initial value depends on the latency of the

operation to execute, being one less than the latency of the operation. The counter decrements with each clock cycle. It outputs '1' while the operation is being executed and '0' when it finishes. This behaviour makes it a suitable internal signal to indicate that an multi-cycle operation is ongoing with the *busy* output flag to external hardware, following the exact timing rule set by the Figure 2.2.

On cases where the result is obtained in one clock cycle, the *busy* flag is not asserted by gating the *start* input signal with the control signal *avoidStart*. This situation is not exclusive to single-cycle operations, but also when special input cases happen, which the output is selected from a number of fixed values (qNaN, both signed infinite and zero) with the *outSel* bus depending on the special case.

The input and post-rounding operands are passed through checks to encode four distinct buses, one for each operand bus. These indicate what type of value hold each operand bus, allowing to identify when the operation falls under a special case. The actual special cases depend on the operation being executed and are specified by the IEEE 754-2008 standard [28], which also fixes the value that must be output and what exception flags to signal.

For example, the addition of different sign infinite operands or subtraction of same sign infinite operands is a special input case that outputs a qNaN and signals a not valid (NV) exception flag through the *fflags* bus. These exception *fflags* are a listed at the Table 3.2 with the general trigger of each.

Mnemonic	Meaning	Triggered by...
OV	Overflow	Rounding exponent output greater or equal to 0xFF.
UF	Underflow	Rounding output is tiny value or zero and inexact.
DZ	Divide by zero	Input operand B is zero at a division operation.
NV	Invalid	Invalid input operand or rounding mode *frm*.
NX	Inexact	The operation resulted contains precision loss.

Table 3.2: Floating-point exceptions fflags and its implemented general trigger.

This operation only uses A and B operand buses. The C operand bus is specific to several fused multiply-add operations. These buses are registered externally to provide a stable input.

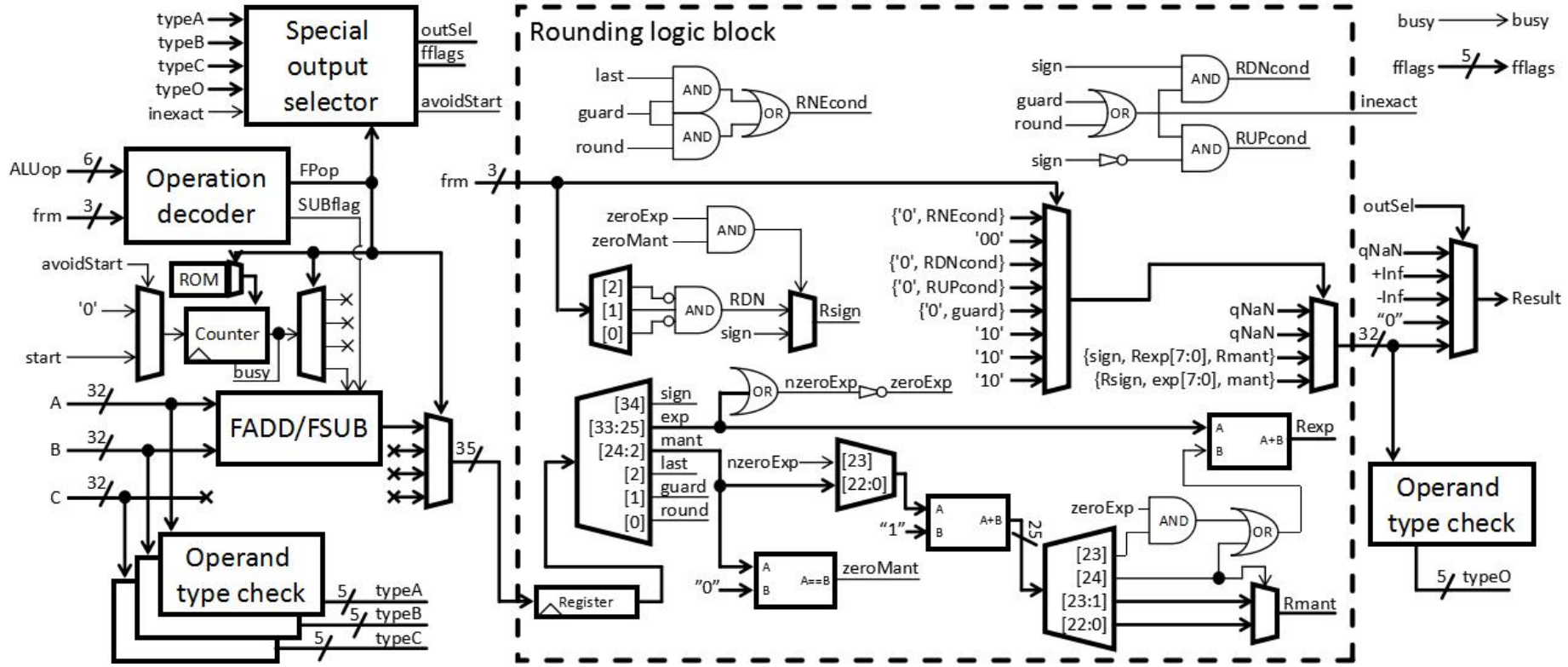

Figure 3.7: FPU module schematic diagram. Includes an input and post-round operand inspection section, an operation decoder to select the proper sub-module output, counter that decrements to indicate a multi-cycle operation ongoing and a rounding logic block.

The *busy* signal enables the write of the pipeline register from the addition/subtraction sub-module to allow internal execution. This design decision minimizes power consumption by avoiding unnecessary parallel computing on other operation sub-modules on the FPU.

Due to the addition/subtraction sub-module has a latency of 2 clock cycles, the addition result is not output until the second clock cycle. Then it is registered on the input of the rounding block with a format of one sign bit, 9 exponent bits (to avoid exponent overflow on other operations) and 25 mantissa bits (23 mantissa and 2 rounding). This design choice gives gate delay slack to all logic paths from any of the sub-modules, which allows to maintain a high frequency by just incrementing the latency of the operation by one. Thus, **the total latency of a single-precision addition or subtraction operation is 3 clock cycles**.

The rounding block truncates the operation result to match the single-precision format, and computes this value plus 1 LSB mantissa. The only possible rounding output are these operation result and +1 LSB mantissa, since only the magnitude is encoded on the exponent/mantissa fields. The selection between both is set by the *frm* input bus and last mantissa, guard and round bits from the pre-round operand. The rounding modes and what result must be output is fixed by the RISC-V ISA. These are summarized in Tables 3.3 and 3.4.

frm encoding	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max. Magnitude
101 and 110	-	Invalid. Set NV flag high
111	DYN	Select FCSR rounding mode. Invalid if still is DYN, set NV flag high.

Table 3.3: Floating-point rounding modes. The rounding mode DYN is used by the instruction decoding module external to the FPU to set the mode registered at the FCSR, that is also external to the FPU.

Rounding mode	Conditions for +1 LSB mantissa output	
RNE	Last and guard bits high.	Guard and round bits high.
RTZ	-	
RDN	Sign and guard or sign and round high.	
RUP	Sign low and guard or round high.	
RMM	Guard high.	

Table 3.4: Rounding floating-point logic condition on the result mantissa to output the operation result +1 mantissa LSB. RNE has two possible conditions, where at least one must be fulfilled. RTZ cannot output +1 mantissa LSB result.

RISC-V supports 5 rounding modes, which are used by the software to round the results when required. However, these are only effective when the arithmetic operations require rounding. On those cases where rounding is not needed, the IEEE 754 standard indications are followed unless the RISC-V ISA specifies otherwise.

The last detail to note is that the inexact (NX) flag is raised when any of the rounding bits (round and guard) are '1,' indicating a loss of precision occurred. This flag is a requirement to trigger the underflow (UF) flag by IEEE 754 standard directive. Although the standard allows to check for underflow on any of pre and post-rounding operands, the RISC-V ISA fixes that it must be done on post-rounding.

3.4 FPU verification

The verification of the extension set F modules is similar to the verification of the extension set M modules detailed in section 2.5 but with some differences. Two simulation environments are set; one for the individual addition/subtraction sub-module and another for the FPU. Both simulation environments pass through several non-synthesizable Verilog HDL tests with reproducible stimulus input. Then the output result is compared to a "known to be correct" result from a reference model that is not synthesizable, but easy to program.

At this point, it has been found that Verilog HDL has a limitation. The language cannot perform single-precision floating-point operations. This characteristic makes difficult the verification process, as the only way to generate a reference model is by

using third-party software or C code execution by invoking it from Verilog HDL during the simulation. The latter is the preferable, performed by what is called a Programming Language Interface (PLI) application [29]. However, due to time constrains, the reference model has not been finished.

This does not mean that there has not been any verification process, but it has been limited to specific cases, which has been thought to tack on possible weak points of the design. These refers to stimulus that search to overflow or take specific data paths on the circuitry. Specifically, these include addition/subtraction between two highest regular single-precision values, a regular and a tiny value, two tiny values while searching to output every output type possible. This is performed by computing the results of the stimulus previous to the simulation. Then, these results are compared with the output of the design under test as it would a reference model.

3.5 Synthesis on a FPGA of the extension set F modules

Through the use of a synthesis program, resource and frequency data from the developed designs can be generated, providing a clear vision of the hardware performance at a specific FPGA platform. The Table 3.5 presents the MFO and resource allocation data of the developed FPU on previous sections, which includes all its sub-modules. Additionally, the table also includes this same data in the case where the latency of the addition/subtraction operation were to be of one clock cycle (no pipeline).

	FPU with pipeline	FPU without pipeline
Max. Freq. 85ºC	59.8 MHz	21.4 MHz
Max. Freq. 0ºC	66.8 MHz	23.7 MHz
Resource allocation	994 LE	1,067 LE

Table 3.5: Synthesis performance data of floating-point operation assets.

On one hand, the maximum frequencies of operation make sense, as the frequency is much higher on the pipeline version than the one without. It can be seen that the ratio is near 3. This indicates that the pipeline segmentation of the FPU is balanced, as the operation with higher latency has the same value, 3 clock cycles.

Pipeline balance is found when the segmentation with the registers leaves all data path stages with a similar gate delay, which in turns maximizes the frequency of operation to a ratio near the number of pipeline stages, or said on another words, the latency of the operation. To increase the frequency even further, the number of pipeline stages can be increased, increasing equally the latency. However, the result topology must maintain a gate delay similar on all its stages or the ratio increase on frequency will be lower than the number of stages. This is a characteristic searched on high performance designs.

On the other hand, the number of LEs is lower on the FPU with pipeline than the one without stages. This does not make rational sense, as the pipeline topology increases the resources to the hardware implementation, at least on paper. The only cause to think of is that the synthesis tool is performing a part of the data path execution on the registers, allowing to decrease the resource usage lightly. It is possible that using specific optimization options other than the default, even lower values on resource allocation could be found at the expense of lower frequency of operation and higher synthesis time.

Comparing the MFO to the previous modules from the extension set M, both have similar figures. This could indicate that the inclusion of the FPU on the core will not have an impact on the MFO. Reason to maintain the module combinations recommended on the previous chapter.

However, it has to be taken into account that these values do not represent the actual MFO of the core when they are incorporated into the core. Synthesis routing of the hardware and data path external to the developed modules could set a new critical path with higher gate delay, decreasing the global MFO.

That is the reason the next chapter focuses on the implementation of these modules and analyses the synthesis data of the whole core, which provides more accurate performance values on section 4.5.

4 | RISC-V 5-Stages Pipeline Core

This chapter introduces the core used and its modifications to implement the extensions sets M and F modules designed in Chapters 2 and 3. Then, further verification and core's performance figures are evaluated. Both instruction sets require infrastructure modifications on the base core to accommodate the modules expansions, but to understand the changes made, first it is important to understand an overview of the base core logic.

4.1 5-stages pipeline RV32I core

The core is a 5-stages pipeline derived from a single-cycle core RV32I [6]. The original single-cycle core capable of executing the base instruction set I performs all the logic in a single combinational block. This property results in a long logic path (high gate delay) from the instruction memory to any internal register or data memory, therefore, a low MFO. The pipeline modification splits the core logic in 5 stages, achieving a higher MFO at cost of increasing the latency of the operations. The instructions are loaded in the first stage and, with each clock cycle, it passes execution through the stages in series using the stage's registers on the pipeline as indicates the Figure 4.1, while also loading next instruction words every clock cycle and also propagating its execution. The stages are listed by the following points:

- **Instruction Fetch (IF)**: it computes the address of the next instruction to be executed from the memory and fetches it. This stage contains the Program Counter (PC), the register used as a pointer on the instruction memory. Due to the instruction memory takes one clock cycle to answer with the instruction of the address called, there is a mismatch of one clock cycle between the PC and the received instruction word. To deal with this mismatch for next stages, the PC is registered while the instruction word is not.

  During an address jump in the instruction memory, an instruction word that should not be executed is called because of the clock cycle mismatch. This is solved by gating the instruction. During the jump, the registered control signal that indicates that a jump is executed or not takes control over the MUX for one clock cycle to send a no operation "NOOP" instruction.

- **Instruction Decode (ID)**: it includes a Control Unit (CU) that receives the instruction from the IF stage and decodes it into a series of control signals or immediate operands used through the core.

This stage also contains a Branching Unit (BU) that decides if it is necessary to perform a jump in the instruction address. Unlike other ISAs, the RISC-V ISA compute branchs over operands stored on the Regfile, a 32-bit register bank of 32 addresses. A Control and Status Registers (CSR) module may also be included, but this point is discussed further in this document. Following execution through the core is propagated to the next stage by registering all the signals of the stage, but specific execution to perform on the same stage between modules or at previous stages is conducted directly without registration on this stage.

- **Execution (EXE)**: it computes arithmetic operations between two immediate operands from the instruction or/and the Regfile. The module in charge to include this logic is the Arithmetic-Logical Unit (ALU), while the operand selection is performed by multiplexers and registered control signals from the previous ID stage. The ALU's output produces results to be stored on the Regfile at posterior clock cycles, but it is also used to compute the address pointer to the data memory at the next stage when performing load or store operations. In the latter case, the operand to be stored is from the Regfile, as the RV32I only allows load and storing operations between the Regfile and the data memory.

- **Data Memory Access (MEM)**: it focuses on load/store operations on the data memory. The MEM stage is designed to support read/write operations of several clock cycles. A handshake protocol between the MEM and the data memory is used. The pipeline is stalled until the MEM stage has granted the access to the memory.

- **Write-back (WB)**: due to the data memory takes one clock cycle to deliver operands during a load operation, this stage exists to match the timing. The output of this stage returns to the Regfile directly without registration in order to record the operand on the proper address, whether it is an EXE stage result or a loaded operand from the data memory. These addresses signals have been registered at each stage termination, but never used until now to preserve the pipeline data-path.

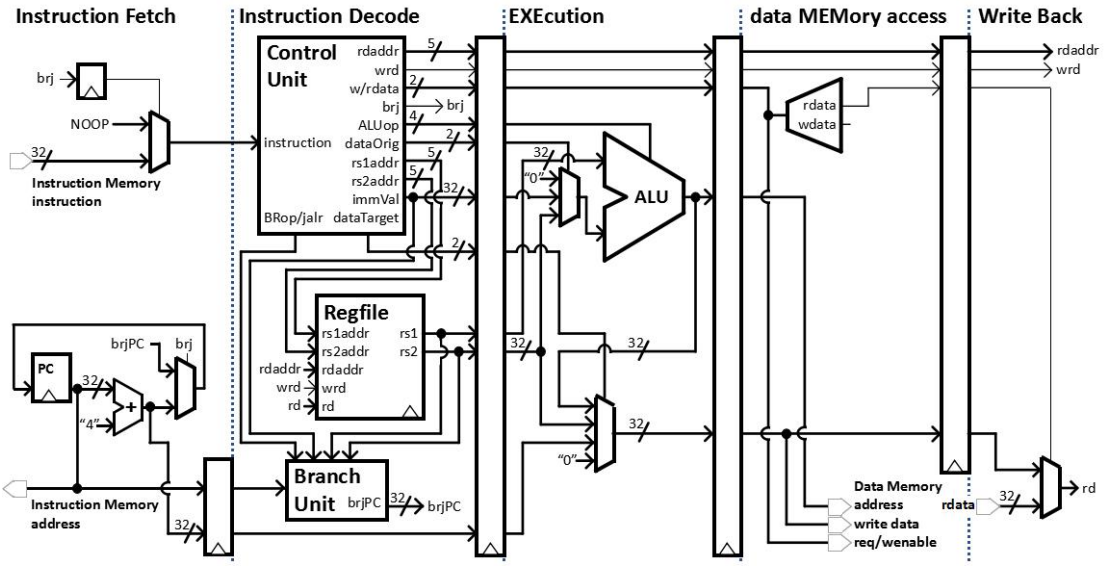

Figure 4.1: Block diagram of the RV32I 5-stage pipeline core. Operand forwarding system, CSR and minor logic is not represented for a simplified figure view.

An structural observation of the pipeline core is that both stages that maintain communication with external memories contain far simpler logic charge when compared to other stages. This allows to have a larger gate delay margin in the communication path with these external memories, opening the possibility of having them allocated on physical external memory modules, like FLASH or SDRAM on-board chips. Although this would enable higher instruction and data memories capacities, they require additional logic to adapt communication between the core and the chip memories (clock frequency, timing, data protocol in general), goal that this project does not focus on and leaves for future projects.

Even when every instruction is passed through each stage of the pipeline, is debatable that in a perspective of the instruction purpose, the total latency of each instruction depends on the operation that it executes. For example, at an unconditional jump operation the instruction is fetched, decoded and the jump is performed, accounting a total latency of 3 clock cycles. However, the instruction is still being propagated without any effective purpose through posterior stages. This is consequence of the pipeline design. **The consensus is to consider that the fully execution on the core of any instruction takes a number of clock cycles at least equal to the number of stages the pipeline has**, 5 clock cycles in our pipeline core.

This is related to the infrastructure design of the pipeline core when compared with the single-cycle core. The higher MFO it allows to execute a higher instruction throughput as shows the Figure 4.2, where it is only holds true when the pipeline core is not stalled. This is the goal of the pipeline design, to **increase the execution throughput by increasing the frequency of operation while avoiding pipeline stalls** as much as possible.

On the limit case where the program contains an infinite number of instruction executions without any pipeline stall, **the effective instruction frequency of execution equals to the frequency of operation**, being the best-case scenario. This premise allows us to compare the single-cycle with the pipeline core performances by just dividing both operating frequencies when considering best-case scenario, or that value by the number of stages of the pipeline for the worst-case scenario.

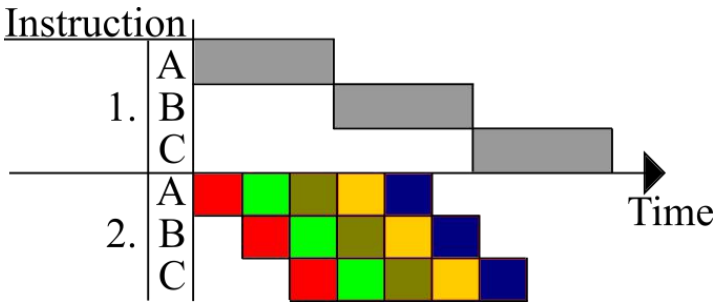

Figure 4.2: General case comparison of A, B and C instructions execution back to back between the single-cycle core (1) and the 5-stages pipeline core (2). Each color of the pipeline core depicts at what stage is the execution of that instruction word (IF, ID, EXE, MEM and WB stages).

Addressing the matter of avoiding pipeline stalls, the Figure 4.1 core has a limitation on its pipeline with the data path of the operands *rs1* and *rs2*. Because a result from the EXE stage takes 3 clock cycles to be recorded on the destiny register *rd* at the Regfile, following instructions may use this result as an input operand even before is on the Regfile. This is a data hazard, and there are two possible solutions; stall part of the pipeline 3 clock cycles to allow the result to be stored on the Regfile, or set up an operand forwarding system that takes the correct operand from further stages on the pipeline. Stalling the pipeline penalizes the core performance and hence, it was decided to implement the operand forwarding logic that is located just after the Regfile. This solution still has an impact on the logic charge of the ID stage and on

the resource allocation, but ensures the best performance, as the bottleneck on the MFO is expected on the EXE stage.

The forwarding system is completely made by combinational logic and follows the behaviour diagram of Figure 4.3. An important detail of the operand forwarding logic is that it prioritizes the nearest stages output operands over the furthest, which makes sense as this returns the most recent results from the pipeline data path.

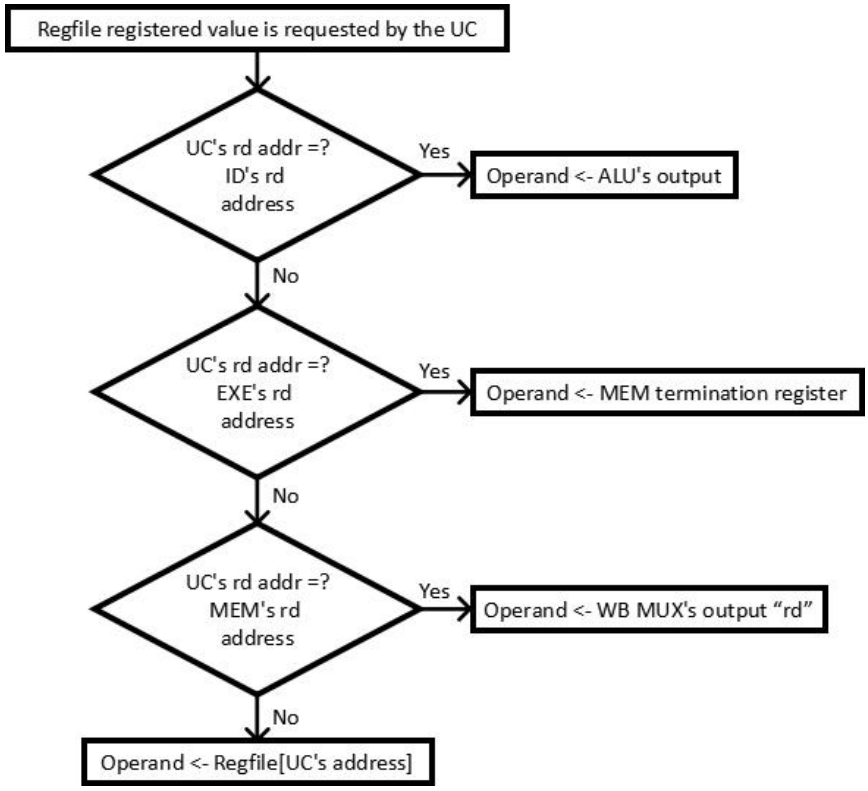

Figure 4.3: Flux diagram of the operand forwarding system at selecting the operand value to output.

The inputs that the forwarding logic manages are the destination addresses of the Regfile at each posterior registered stage's termination, used to compare it to the operand called by the CU. If there is a match and the write enable of the Regfile at that stage is asserted, the forwarding system redirects the operand of that stage to the ID termination registers and other modules including the CSR and the BU. With this method, the effective result is called from any posterior stage when applied to both *rs1* and *rs2* input operands, delivering the requested operand without stalling.

Even though this forwarding circuit solves this data hazard for most instructions, there is a specific case where stalling the IF and ID stages is still required. This case is when a decoded instruction calls for a operand that is not yet on the core, because it is a operand to be loaded from the data memory on the following clock cycles. On these situations, the IF and ID stages stall for 1 or 2 clock cycles until the forwarding system can provide the requested operand. However, this stall can be avoided at software by not writing instructions with data dependence over operands to be loaded by the previous instruction.

Previously, it has been introduced an optional CSR module on the ID stage of the pipeline. This component is not part from the base set I of the RISC-V ISA but the subset Zicsr.

The extension set Zicsr defines a separate address space of 4,096 32-bit registers, which are accessible with 6 instructions specific to the subset. However, apart from some counters, timers and a floating-point status register specific on cores supporting the instruction set F, they are optional to implement even on the Zicsr subset. These registers are used to allow designers to implement custom functionalities on its processors. Due to the inclusion of the extension set F on this project, the Zicsr subset is supported since this RV32I implementation of the core.

In order to provide a single processor package capable of executing a program, it is incorporated alongside of the instruction and data memories as shown at Figure 4.4, being this module named the core's region. The communication between the core with these memories is not direct, it is a master-slave implementation allowing to control the core execution by an external entity.

This design sets the foundation to allow re-writing the program loaded into the instruction memory and access to the data memory without recompiling the whole project. However, further work is required to enable this functionality, that includes implementing an Advanced eXtensible Interface (AXI) bus and setting the communication protocols with other components connected to the bus, tasks left for a future project.

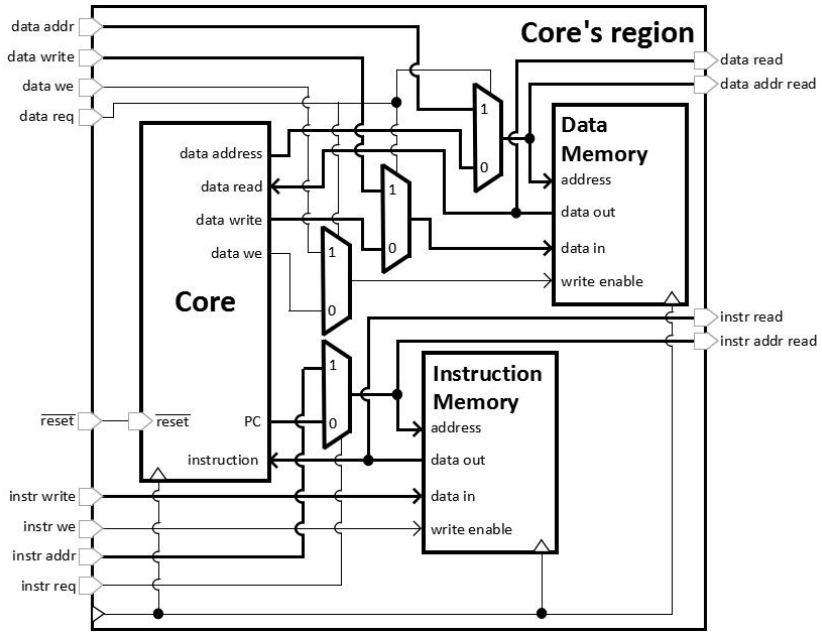

Figure 4.4: Block diagram of the core region, including instruction and data memories, core and master-slave logic to access the memories.

4.2 Core adaptation for the instruction set M support

The instruction set M adds integer multiplication and division operations support. The Figure 4.5 shows the structural impact on the core required to expand the RV32I core to RV32IM core version.

The main addition made to the core is the MULDIV module and a “start multi-cycle” system that asserts an *start* control signal to the respective arithmetic module when the operation may take more than one clock cycle to be finished. To avoid an execution loop of the same operation, a close-loop feedback with the *busy* signal generated by the MULDIV module is made. This feedback logic leaves a clock cycle between finishing a multi-cycle operation and the signalling start, so the next instruction propagation may happen at the whole pipeline and the result of the operation may pass through to the next stage. This logic follows the previously presented timing rule at Figure 2.2.

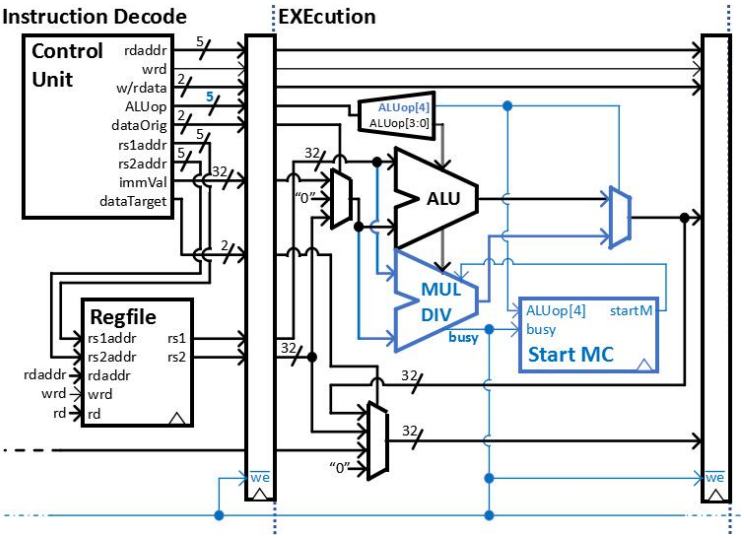

Figure 4.5: Core RV32IM with modifications marked in blue to implement extension set M support from the core RV32I. Operand forwarding circuit, other stages and minor logic omitted for figure clarity.

When a multi-cycle operation is on-going, a pipeline stall on the core is required to leave room for the operation and obtain the proper result. Because of this, the *busy* signal is used as general pipeline stall, which is distributed along the core to stop all the stages registers from propagating data. This stall limits the core throughput, being the reason of why it has been put so much care into offering a variety of sub-module options with lower latencies on the division operation, so the stall is minimized.

The length of the bus used to specify the operation, to be executed in the EXE stage, *ALUop,* has been increased one bit in order to support the new operations. This added bit is used to select the arithmetic output between the integer ALU and the MULDIV, while the other bits of the bus are used to indicate the operation to be performed by those modules.

This expansion is also needed at the CU of the ID stage by implementing the decoding logic for the instruction subset M. Figure 4.6 shows that the decoding is done by taking segments of the instructions, avoiding the need of a LUT. This design decision may result in a lower impact on resource allocation and higher MFO in case of implementing the core in an ASIC. But if ported in an FPGA, this optimization does not provide any better performance because of the use of standard LEs.

RV32M Standard Extension

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
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

ALUop = [funct7[0], funct7[5], funct3]

Figure 4.6: ALUop encoding for the extension set M instructions.

A side effect of the addition of the MULDIV module in the EXE stage is an increase of the power consumption and of course, also of area. This is the result of the fact that every operand from the ID is passed to the ALU and MULDIV modules as shown in Figure 4.5. These operands propagate through both modules although no computation is needed. A possible power consumption optimization would be to stop the propagation of the operands when no computing has to be performed. This can be done by gating the input operands of each module and only changing when a valid operation is signalled to the specific module avoiding the undesired internal execution. This design does not gate the input operands because this project does not focus on a low power consumption implementation.

4.3 Core adaptation for the instruction set F support

The instruction set F gives support for the management and operations of floating-point operands, including load and store of floating-point operands to the data memory. The module additions required in the core to implement this support include the incorporation of the FPU on the EXE stage, an FP Regfile (FPRF) and a Floating-point Control and Status Register (FCSR) on the ID stage, expanding the RV32IM core model to a partial (because the set is not fully supported) RV32IMF. Figure 4.7 shows a block diagram of the core with these modifications depicted in blue.

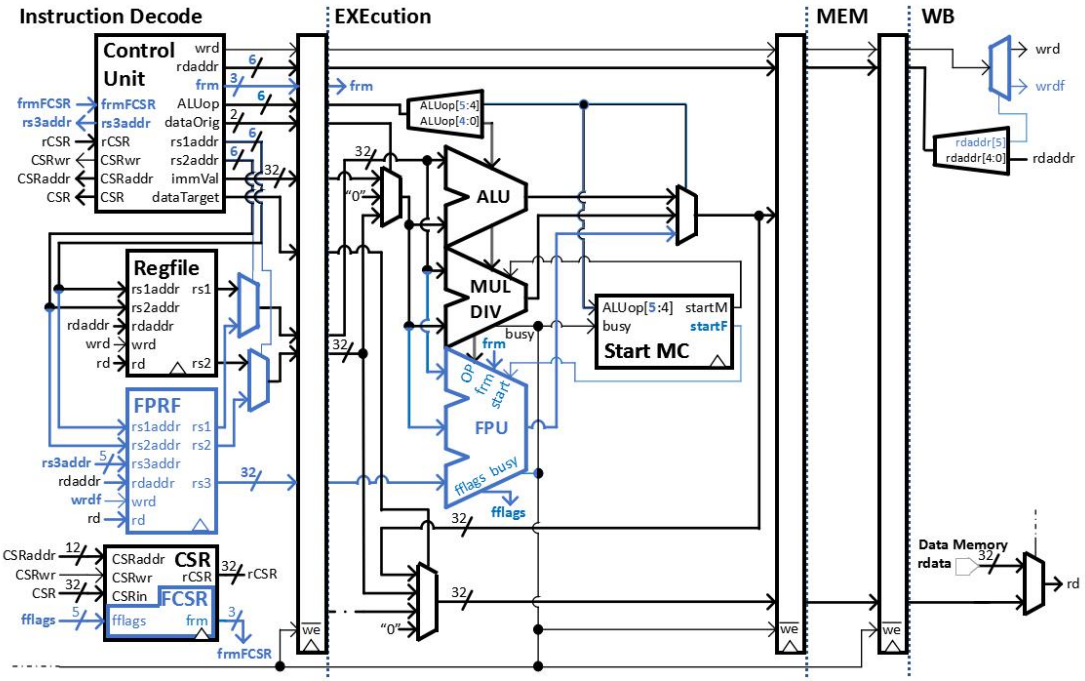

Figure 4.7: Core RV32IMF with modifications marked in blue to implement subset F support from the core RV32IM. Operand forwarding circuit, IF stage, data memory access and minor logic omitted for figure clarity.

The FPRF bank register is similar to the integer Regfile. Both are internal memory banks of 32 registers with a width of 32 bits. The only structural difference is that the FPRF has a third operand addressing, allowing 3 operand inputs for the fused multiply-add operations at the FPU. This third operand requires its own forwarding logic, which adds more resources on use but does not lower the MFO because this operand bus is specific of the FPRF, not requiring a multiplexer to select between integer and floating-point register banks unlike the other two input operands.

To address both register banks, the address buses have been increased to from 5 to 6 bits. The added bit is used to select the operand between the integer Regfile and the FPRF. This *rd*'s sixth addressing signal registered at the WB stage is used to redirect the write enable signal to the Regfile or to the FPRF, preventing any erroneous writing in the register bank. The encoding of this sixth addressing signals for the input and destiny operands at the register banks is hard-wired to be on the Regfile by default, to ease future extensions.

This method of increasing the address buses also helps the encoding of various instructions of the subset F, as some of the instructions use integers or write on the Regfile registers. Further details on each instruction are given on the ISA manual [5]. The write enable encoding takes into consideration that depending if the destination register *rd* is at the Regfile or at the FPRF, it does not write on the x0 position if is on the integer bank or does not check if is the floating-point bank. This is because writing on the position x0 of the Regfile is forbidden by the RISC-V ISA, where it is not the case on the FPRF.

The ability of performing the floating-point operations at the EXE stage is extended by adding the the FPU module described in Chapter 3 and increasing the *ALUop* bus from 5 to 6 bits. The management of the selection of the result and the multi-cycle operation *start* signal is controlled by the 2 MSB of *ALUop*. In this way, any incorrect general stall for multi-cycle operations is prevented from triggering, as only the correct module start is asserted.

The *ALUop* encoding at the CU to support this instruction subset is done as shown in Figure 4.8. Because of the number of instructions is high and there is no bit segment shared by all the instructions to be used as unique signal derivation encoding, the bit encoding is split between 3 groups of instructions: simple FP operations, fused multiply-add operations and load/store operands. The latter follows the same *ALUop* as an integer load/store operation, with the only difference in taking the operand to store from or loading the operand on the FPRF. This means that the data memory address is still being computed using integer values from the Regfile with an integer ADD operation through the ALU.

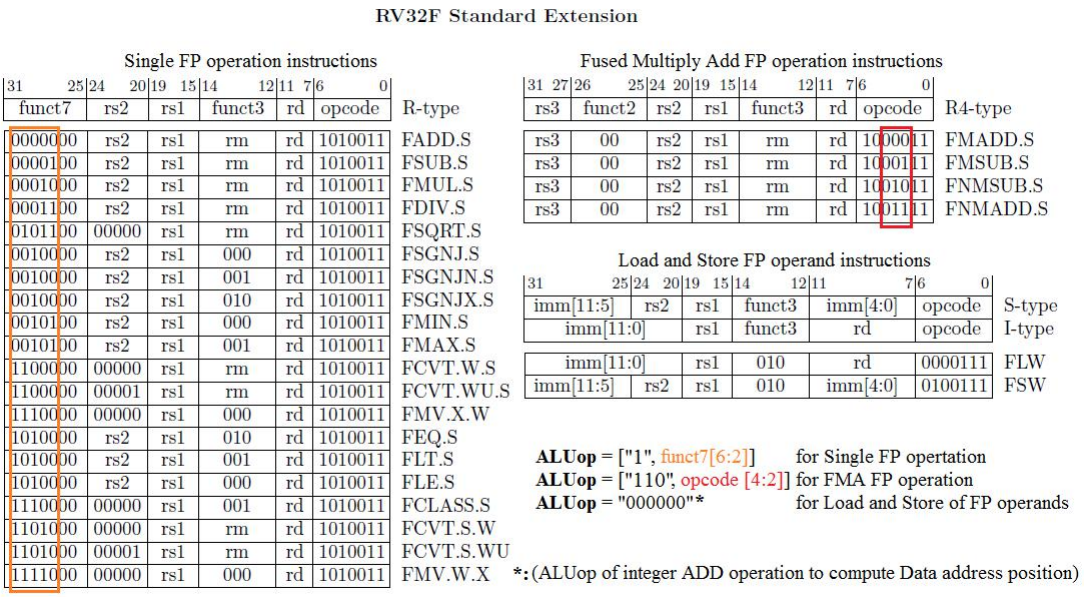

Figure 4.8: ALUop encoding for the extension set F instructions.

The Floating-point Control and Status Register is a 32-bit register located in the CSR module of the ID stage and it is managed by Zicsr instructions. The FCSR contains a 5-bit accrued exemptions field *fflags*, a 3-bit rounding mode field *frm* and 26 reserved bits field, as shown in Figure 4.9.

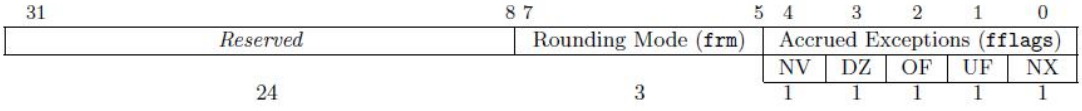

Figure 4.9: Floating-Point Status and Control Register.

Starting by the *fflags*, these bits are risen by any flag exception from many of the FPU operations as stated previously in Chapter 3. The FCSR registers these flags when asserted and hold them high unless a Zicsr instruction modifies them, hence the accrued property. Because of this requirement by the ISA, a specific 5-bit bus is connected from the FPU to the FCSR register. This field allows the software to evaluate the program execution by looking at these bit positions.

The *frm* field is used by the FPU when the instruction word has set its rounding mode field to dynamic. The multiple modes are reviewed in Chapter 3. The connection of these bits to the FPU pass through the CU to decide what rounding mode must be set between the instruction word and the FCSR. After this, the resulting rounding mode is registered at the termination of the ID stage, since they must be synchronized with the input operands and the other control signals at the EXE stage. This field allows software to set a default rounding mode for FP operations.

The reserved field is used by other standard sets and should not be accessed if they are not implemented.

With this, the core RV32IM has been expanded to support the instruction set F, resulting in a RV32IMF design. This is asserted because, even when the FPU does not implement all the operations from the instruction set F, the core has been modified to provide support to all the instructions, although further verification is required when the remaining are implemented.

4.4 Core verification

The multiple modifications performed to obtain the core versions RV32IM and RV32IMF may interfere in unexpected ways with internal processes of the base core. That is the reason a verification process through simulation has been performed to guarantee the instruction executions are carried out as expected and are RISC-V ISA compliant.

At this project stage, the simulation stimulus is not provided directly by a test file, but by simulating the execution of a software program recorded at the instruction memory during the start of the simulation. This instruction memory program is written by a series of pseudo-code instructions, that are machine encoded by an assembly file as shown in Figure 4.10.

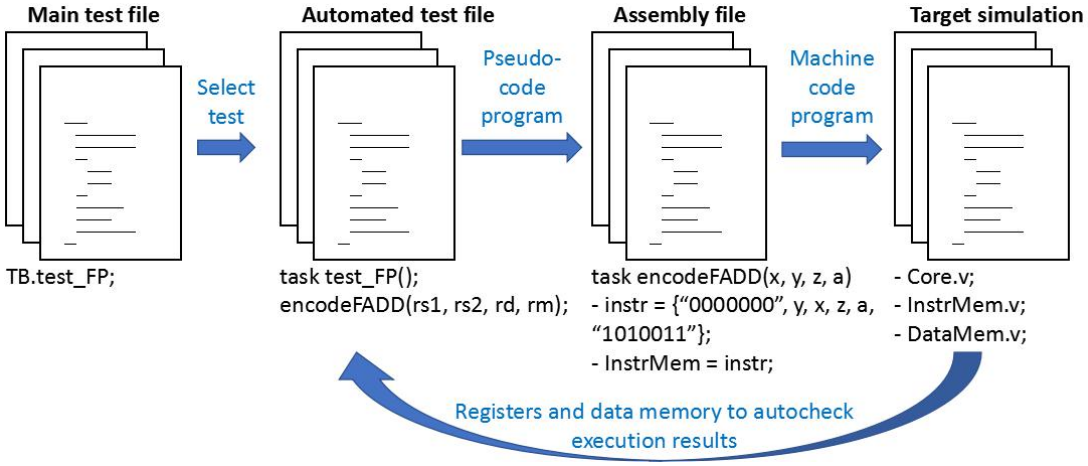

Figure 4.10: File hierarchy for simulating a software program execution.

The translation from pseudo-code to machine code performed by the assembler is achieved using *tasks*, a Verilog HDL command able to call a process with some inputs to perform a routine. The specific work of the assembler is to take the task inputs depending on the pseudo-instruction (*rs1*, *rs2* and *rd* 5-bit addresses, 3-bit rounding mode for a FADD instruction) and translate it to the RISC-V machine instruction the core is able to understand. All pseudo-code is translated to its own machine word and written on the instruction memory at the DUT core.

Once the program is written, the automated test file sends a clock signal to execute the software program on the DUT core. Using Verilog commands at the automated test file, an execution time is left to provide enough time to the core to execute the program. Then an automatic check is made on the Regfile, FPRF and data memory to compare the results of the program execution with the expected previously calculated, outputting an "Ok" or an error message.

Just as on previous verifications, the simulation tool allows to represent graphically the execution by tapping at the connections and registers of the simulation target, the core region and its components in this environment. This is used to debug the program execution, that at this stage becomes more complex by the quantity of input and output signals at every stage of the pipeline.

The pseudo-code programs are written to test the extension sets support and verifies if it is working as expected. Full operation of the modules added are not tested, as that task is already done at previous verification stages on Chapters 2 and 3.

The base core RV32I is verified with load and store, base arithmetic and branch tests. The expanded core RV32IM is verified with the 8 expanded operations, multiple expanded operations in a row and the one-cycle remainder feature tests. The expanded core RV32IMF is verified with loading, addition, subtraction and storing of FP operands tests.

Furthermore, by using the xPack RISC-V compiler on the software programming environment Eclipse, it has been able to create a program written on C++ and then translated to the RISC-V machine language. Then, this same program has been executed on the DUT cores, **confirming that higher level programming on these processors is possible**.

Because the RV32IMF supports set I and M, it also has been verified with the tests applied to those sets. The same can be applied to the RV32IM with the base set I.

4.5 Synthesis on a FPGA of the core models

With the knowledge of the designed cores are fully functional, they have been synthesized in our prototyping FPGA to produce performance data, including MFO and resource allocation in LEs. The Table 4.1 summarizes these data.

Because of these data depend on the module combinations used at the implementation of the subset M support for the RV32IM and RV32IMF cores, it has been fixed with the performance recommendations made in Chapter 2, being the MULDIV2 with the DEQS division model and the ALTMULT multiplier.

Reviewing these data, it can be said that on the best case-scenario, **the pipeline RV32I core allows 19 times the instruction throughput than its non pipeline variant** by the increase of the MFO. Even though this gain in performance may be limited by the possible pipeline stalls because of data hazards, there is still a gain of 3.8 times. However, **the number of resources is increased by a 43%** as trade-off on our prototyping platform.

Core model	No pipeline RV32I	RV32I	RV32IM*	RV32IMF*
MFO 85ºC	3.0 MHz	57.7 MHz	44.7 MHz	44.2 MHz
MFO 0ºC	3.3 MHz	63.1 MHz	49.8 MHz	49.5 MHz
Resource usage	2,465 LEs	3,520 LEs	5,125 LEs	8,913 LEs

Table 4.1: Maximum Frequency of Operation at 85 and 0ºC in MHz and resources used in Logic Elements at the synthesis of the different core models. *: Using the best performance recommendation from the subset M module combination.

Comparing the RV32I and the expanded RV32IM there is a decrease on MFO, indicating the implementation of the MULDIV2 module has set a more constrained critical path, limiting the MFO to 44MHz. Furthermore, the resource usage is increased a 45%, a value similar to the previous increase in resource usage respect the core predecessor.

Performing the same operation between the RV32IM and RV32IMF core models, it is seen that the MFO is barely affected, indicating that the segmentation of the floating-point addition/subtraction sub-module has been effective on holding a high MFO on the core. Nevertheless, the resource usage is again incremented, now a 69% over its core predecessor. However, the resource data regarding the RV32IMF will change on the future project that adds full support to the subset F, since only a part has been developed.

Figure 4.11 shows this resource data visually by segmenting the resource usage on each core model in such way to represent on what components the resource are being used on.

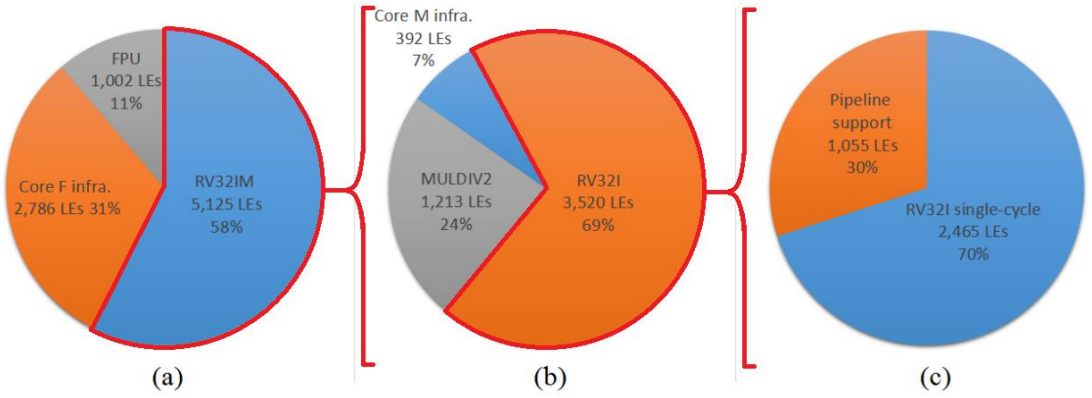

Figure 4.11: Resource allocation for cores (a) RV32IMF, (b) RV32IM and (c) RV32I pipeline indicating on what components the resources are being used on.

5 | Prototyping on a FPGA

On the previous chapter the core was synthesized on a EP3C16F484C6N FPGA from Intel in order to obtain data on performance figures at every stage of the development process. In this chapter, the core is ported into a DE0 development board to test it in real time. For such a purpose, it is necessary to add to the core a series of components to interact with the external world. Those are described in section 5.1. At last, the porting method is explained in section 5.2.

5.1 FPGA implementation and synthesis

The core, as previously mentioned, needs some additional components to interact with the external world. It was decided, due to available time, to do a quick prototyping on a DE0 development board from Terasic (Intel), so the core only managed some LEDs and 7 segments displays. One of the main disadvantages of this prototyping, is that no new program can be loaded into the instruction memory after port. To run a new program, the project needs to be recompiled. Future additions, as a master/slave interface in the core's region to communicate with an AXI bus, will provide the means to re-program the instruction memory without recompiling the project.

Figure 5.1 shows a block diagram of the core with the added components. A 10 bits GPIO memory area is assigned to handle the LEDs. These represent the 10 LSB of stored data at a specific data address memory, allowing to see what is being written. Two 7 segments display controllers are connected to the 8 LSB of the core's PC. Because of the limited number of 7 segments displays available in the DE0 board, it is not possible to display the whole word of the instruction being loaded into the core, so just the 8 MSB are displayed on the other two 7 segments.

The correct program's execution must be deduced by looking at these outputs with such limitations. Future tasks adding a debug ring will allow direct access with to these memory regions, avoiding these limitations.

The reset signal of the core is connected to one of the board's buttons, but not directly. A debounce circuit filters some noise pulses produced by the mechanical nature of the physical button. The clock signal is managed by using the first two on-board switches to select three clock modes; a 1MHz clock from a PLL, a 10Hz clock from a GPIO counter or from a button to allow step-by-step execution.

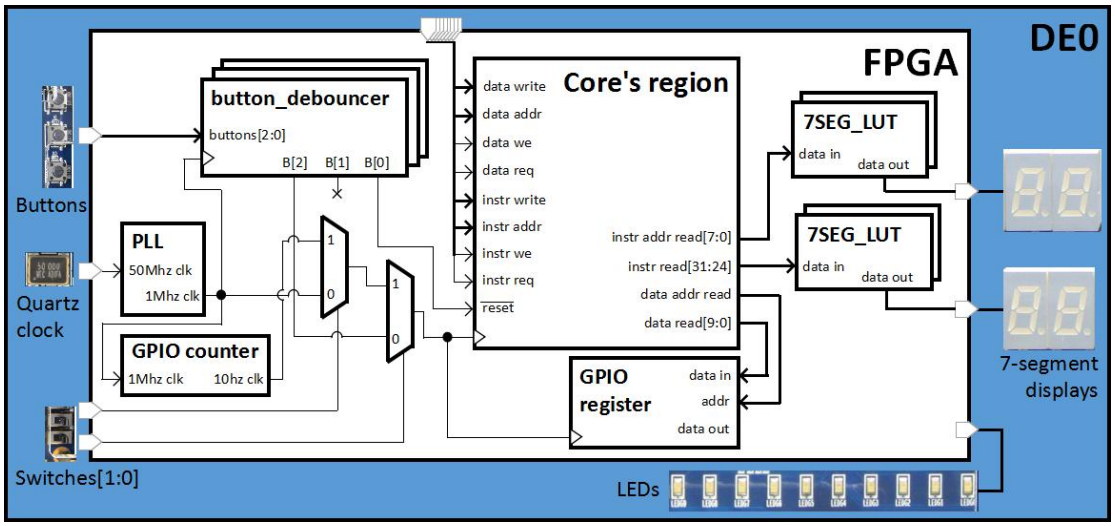

Figure 5.1: Interface module block diagram.

The use of the 50MHz DE0's board quartz clock signal might not be adequate for some of the designs, such as the RV32IM and RV32IMF cores which maximum frequency of operation is below this clock signal frequency. That is the reason to use an embedded PLL block, which allows to tweak the frequency of operation while providing an stable clock signal. It is possible to obtain lower multiple clock frequencies with this clock signal through the use of registers. However, the clock signals at an FPGA synthesis tool require high specifications only obtained through the use of an PLL to be able to assign a clock tree.

There are many options to generate a suitable PLL clock generator for any application, including on what type of PLL to use (selects between different PLL embedded blocks depending on the constrains), mode of operation (normal, source-synchronous, zero-delay buffer, no compensation and external feedback), between others that can be reviewed at much larger extension at its IP documentation page [30]. Some of the specification options include the input and output frequency, the frequency lock range, etc. While many of these characteristics are not a necessity at this project, the main

requirement is to lower the 50MHz clock input signal to an 1MHz clock output signal for testing purposes.

FPGA resource synthesis is limited by the number of LEs available by the FPGA model being ported on. In our case, the limit of the FPGA only allows to allocate a low number of memory blocks to synthesize the instruction and data memories, insufficient for a practical use. That is the reason to use the embedded M9K blocks.

These are specially designed to allocate a higher memory space than using LEs. This M9K embedded blocks are reviewed at the Cyclone III Handbook Volume 1 [10], Chapter 6, where indicates that M9K is the only memory type embedded block available at the Cyclone III family. Other platforms may have different resources which must be adapted when porting to another FPGA.

Contrary to the multiplier modules, the requirement of using a specific hardware IP is set aside, since Intel leaves the option to synthesize memory blocks by using specific HDL code templates. The synthesis tool tries to fit the written HDL code to a template and if the assimilation is achieved, the synthesis is performed by default using the M9K memory blocks. The Verilog HDL template and other synthesis options are specified at the "Design Recommendations" document from Intel's Quartus User Guide [31].

With the use of M9K blocks, it has been able to implement **32KB of instruction memory** (8,192 instructions) and **16KB of data memory** (4,096 addresses of 32-bit registers) using 48 of the 56 M9K blocks available at the FPGA target, without any LE allocation.

However, if this memory is not accessible by external signals, the synthesis tool optimizes them as a ROM. This has the effect of not synthesizing correctly the design, thus, the buses that allow to implement the AXI bus are connected to the exterior, even though they end being grounded during synthesis (the outcome of this step depends on the synthesis tool).

5.2 Porting on the DE0's FPGA

The final step to port the project onto the target FPGA is relatively easy if all previous steps have been taken with care and the synthesis has no errors. This is because the board DE0 has embedded an USB Blaster Circuit that makes the process of programming the FPGA easier, performing as interface between the programming tool from Quartus II and the target FPGA through JTAG UART configuration signals, as shows the Figure 5.2.

The specific steps are connected the computer to the USB-B port of the DE0 board, set the PROG/RUN switch to RUN at the board DE0 and use the "Program Device" option at Quartus II selecting the connected board. This and further information is specified at the DE0 User Manual [11].

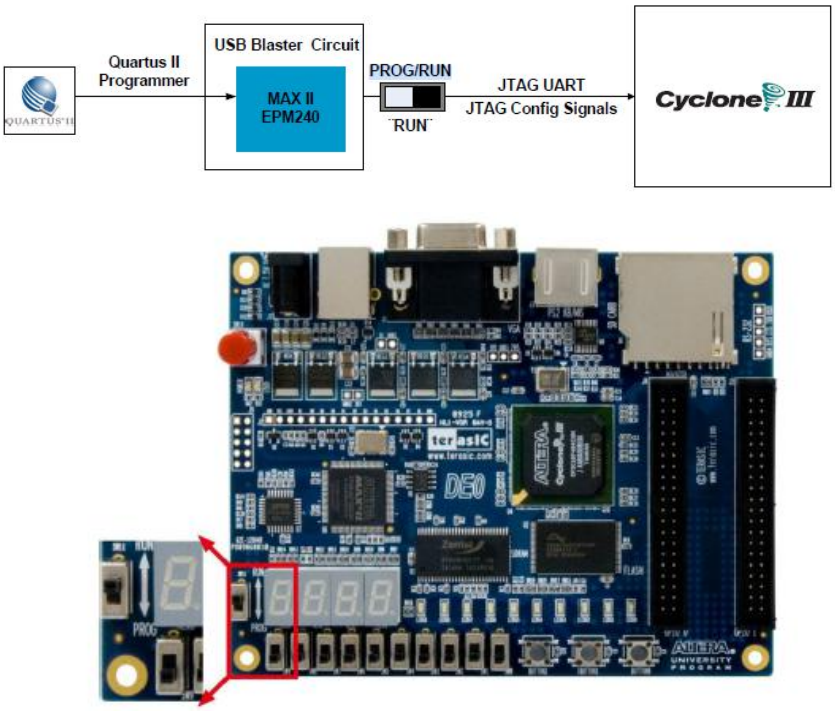

Figure 5.2: Block diagram of the communication system used to program the FPGA and DE0 development board. This figure has been extracted from the DE0 User Manual [11], Figure 4-1 and 4-2 at page 21.

Once the porting process is finished, it can be verified that the project synthesis onto the DE0 board has been a success. This verification is performed by comparing the program execution at the simulated space with the LEDs and 7 segment displays of the physical prototyping DE0 board.

6 | Conclusions and future work

This project has been a success on developing several processors on Verilog HDL with different degrees of performance and support on the instruction sets from the RISC-V ISA. Specifically, base set I, subset M and partial support for the subset F on the RV32I, RV32IM and RV32IMF core models respectively.

Not only has it been followed state of the art algorithms and data for the design, but particular optimizations on the divider module have been developed, achieving matching performance on average with the most complex algorithms, as our benchmark data has proven. The developed implementation options grant flexibility to adapt the core to the needs of the client, whether in a range for lower area usage or better performance.

All the developed cores allow a frequency of operation at 40MHz minimum on the ported FPGA platform, by using the embedded 9-bit DSP and M9K blocks to implement multipliers and memory space respectively.

Furthermore, all work presented has passed a verification process that ensures the developed components are RISC-V compliant. These could be expanded to provide an even more rigorous verification, especially on the subset F operations. Moreover, the execution of RISC-V programs compiled from C and ported to physical FPGA hardware opens the possibility of writing software for these cores on the future.

The designs take into consideration future work to be done, by leaving the proper framework to expand the core with a debug ring, an AXI bus and to implement the full support for the subset F.

Acknowledgements to Intel and Siemens for providing free license on its legacy software Quartus and ModelSim, the “Centre de Prototips i Solucions Hardware-Software” (CEPHIS) from MiSE Dpt. at UAB for providing the DE0 development board and special thanks to my tutors for their training and patience on the proofread of this master thesis.

Bibliography

[1] Krste Asanovic and David A. Patterson, “Instruction Sets Should Be Free: The Case For RISC-V.” Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, 2014.

[2] RISC-V Industry alliances. url:https://riscv.org/community/alliances/

[3] Seagate Technology plc, "Seagate Designs RISC-V Cores to Power Data Mobility and Trustworthiness," December 8, 2020. Fremont, CA. url:https://www.businesswire.com/news/home/20201208005224/en/Seagate-Designs-RISC-V-Cores-to-Power-Data-Mobility-and-Trustworthiness

[4] Kim McMahon, "RISC-V Introduction. Welcome to the Open era of computing!" url:https://riscv.org/about/

[5] Andrew Waterman, Krste Asanović, “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA,” SiFive Inc., CS Division, EECS Department, University of California, Berkley, December 2019.

[6] Pau Casacoverta Orta, Single-cycle RISC-V core "RV32I Verilog” GitHub repository, lastly updated by Februrary 2020. url:https://github.com/4a1c0/RV32i-Verilog

[7] Francisco Javier Fuentes Diaz, 5-stages pipeline RISC-V core and instruction sets M and F expansions GitHub repository. url:https://github.com/FFD-UAB/RISC-V-Instruction-sets-M-and-F

[8] “ECE 5745 Complex Digital ASIC Design: Verilog Usage Rules,” School of Electrical and Computer Engineering, Cornell University, February 2019.

[9] Doulos, "The Verilog Golden Reference Guide," 2003. ISBN:0953728048

[10] Terasic Altera (Intel), "Cyclone III Device Handbook," CIII5V1-4.2, August 2012. url:https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyc3/cyclone3_handbook.pdf

[11] Terasic Altera (Intel), "DE0 User manual: Development and Education Board," 2009. url:https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-5804152209-de0-user-manual.pdf

[12] Intel, "Quartus II Web Edition," realease 13.1 v21.1, November 2013. url:https://fpgasoftware.intel.com/13.1/

[13] Siemens, "ModelSim" behavioral, RTL and gate-level simulator. url:https://eda.sw.siemens.com/en-US/ic/modelsim/

[14] Eclipse IDE programming environment. url:https://www.eclipse.org/

[15] Liviu Ionescu, et al, "xPack dev tools" RISC-V compiler GitHub repository. url:https://github.com/xpack-dev-tools/riscv-none-embed-gcc-xpack

[16] Steve Kilts, "Advanced FPGA design: Architecture, Implementation, and Optimization," Spectrum Design Solutions, Minneapolis, Minnesota. Published by John Wiley & Sons, Inc, 2007. ISBN:978-0-470-05437-6.

[17] Yamin Li, "Computer Principles and Design in Verilog HDL," ch. 3.4, 2015, Hosei University, Japan, published by Tsinghua University Press, Wiley. ISBN:9781118841099.

[18] Ted E. Williams, Mark Horowitz, "SRT division diagrams and their usage in designing intergrated circuits for division," Stanford University, Computer Systems Laboratory. Technical report CSL-TR-87-326, November 1986.

[19] E. Matthews, A. Lu, Z. Fang and L. Shannon, "Rethinking Integer Divider Design for FPGA-Based Soft-Processors," 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019, pp. 289-297. doi: 10.1109/FCCM.2019.00046.

[20] T. Dutta Roy, "Implementation of Goldschmidt's algorithms with hardware reduction," July 2015. doi:10.13140/RG.2.1.2290.4163.

[21] James E. Stine, "Digital Computer Arithmetic Datapath Design Using Verilog HDL," ch. 7, published by Kluwer Academic Publishers, Boston, Dordrecht and London. ISBN: 978-1-4419-8931-4.

[22] Stuart F. Oberman and Michel J. Flynn, "Measuring the complexity of SRT tables," Departmens of Electrical Engineering and Computer Science, Standford University. Technical report CSL-TR-95-679, November 1995.

[23] C. S. Wallace, "A Suggestion for a Fast Multiplier," in IEEE Transactions on Electronic Computers, vol. EC-13, no. 1, pp. 14-17, February 1964, doi:10.1109/PGEC.1964.263830.

[24] L. Dadda, "Some Schemes for Parallel Multipliers", Alta Frequenza, vol. 34, pp. 349-356, 1965.

[25] Whitney J. Townsend, Earl E. Swartzlander, Jacob A. Abraham, "A comparison of Dadda and Wallace multiplier delays," December 2003, University of Texas at Austin, doi:10.1117/12.507012.

[26] Rathisha Shetty et al, "Design and Implementation of High Performance 4-bit Dadda Multiplier using Compressor," International Journal of Computer Science and Mobile Computing, Vol.6 Issue.7, July-2017, pg. 249-254.

[27] Terasic Altera (Intel), "Integer Arithmetic IP Cores User Guide," August 2014. url:https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altmult_add.pdf

[28] "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2008 , vol., no., pp.1-70, 29 Aug. 2008, doi:10.1109/IEEESTD.2008.4610935.

[29] Stuart Sutherland, "The Verilog Pli Handbook," published by Springer, March 1999. ISBN:079238489X.

[30] Terasic Altera (Intel), "Altera Phase-Locked Loop (Altera PLL) IP Core User Guide," June 2017. url:https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altpll.pdf

[31] Terasic Altera (Intel), "Quartus Primer Pro Edition User Guide," pp. 13-15, September 2020. url:https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-design-recommendations.pdf