Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group 48

Alessandro Marchei, Silvia Capozzoli, Tommaso Terzano

October 19, 2023

Contents

1 Introduction **1**

2 Insturction Set **3**
2.1 Registers and Instructions format . . . 3
2.2 Instruction Set . . . 4
2.3 Pipeline . . . 6

3 Datapath Overview **7**
3.1 Almond-32 Datapath Implementation . . . 7
3.1.1 Instruction Fetch Stage . . . 8
3.1.2 Instruction Decode Stage . . . 10
3.1.3 Execution Unit . . . 12
3.1.4 Memory Unit . . . 13
3.1.5 Write Back Unit . . . 14

4 Control Unit **15**
4.1 Hardwired Control Unit . . . 15
4.1.1 Managing the Pipeline . . . 16
4.1.2 ALU Opcode . . . 17
4.1.3 Reset of Control Words . . . 17

5 Register File **18**
5.1 Read and Write Operations . . . 19

6 ALU **20**
6.1 Logical Unit . . . 20
6.2 Shifter . . . 22
6.3 Adder . . . 23
6.3.1 Carry generator . . . 24
6.3.2 Sum generator . . . 25
6.4 Comparator . . . 25
6.5 Multiplier . . . 27
6.5.1 Booth's Algorithm . . . 27
6.5.2 Implementation . . . 27
6.5.3 Pipeline . . . 27

7 Zero Detection Unit **29**

8 Branch History Table **30**
8.1 Role and Design of BHTs in Modern Processors . . . 30
8.2 BHT implementation in Almond-32 processor. . . . 31
8.2.1 Latency improvements due to BHT . . . 32

9 Forwarding Detection Unit **34**
9.1 Data Hazards Significance in Operational Integrity . . . 34
9.2 Forwarding Unit Implementation in Almond-32 . . . 35
9.2.1 Forwarding for ALU Operations . . . 36
9.2.2 Forwarding for Zero Detection Unit . . . 37
9.2.3 Forwarding for Store Instructions . . . 38

10 Hazard Detection Unit **39**
10.1 Architecture of HDU . . . 39
10.2 RAW Hazard . . . 40
10.2.1 Hazard mitigation with forwarding . . . 40
10.2.2 Managing RAW Hazard . . . 40
10.3 Structural Hazard . . . 40

11 Conditional Write Back Unit **41**

12 DRAM **43**
12.1 Big-Endian Byte-Addressable Memory . . . 44
12.2 Byte Lane Control for Store Instructions . . . 44
12.3 LMD MUX for Load instructions . . . 44

13 IRAM **46**

14 Synthesis **47**
14.1 Results . . . 47

15 Placing and Routing **49**
15.1 Results . . . 50

A Simulations **51**

B System block diagram **53**

CHAPTER 1

Introduction

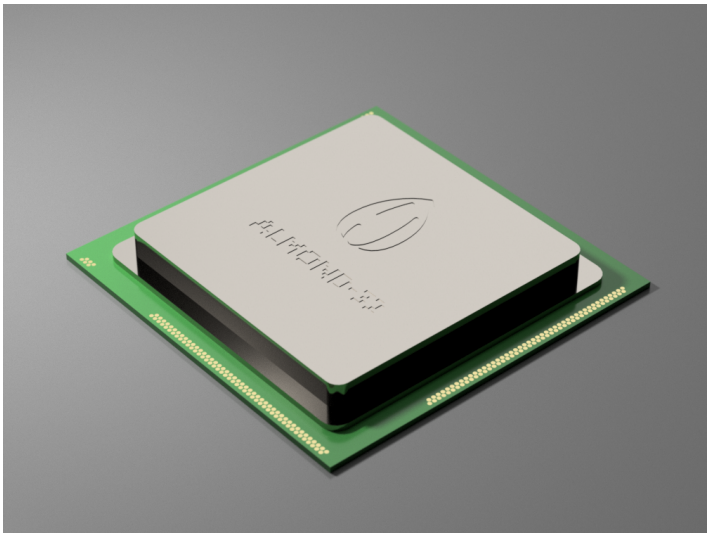

Figure 1.1: 3D render of Almond-32 microprocessor, a DLX based system

Almond-32 is a microprocessor inspired by the DLX (DELUXE) architecture, a RISC (Reduced Instruction Set Computer) processor, which itself shares design traits with established processors like AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation-1, Sun-4/110, Sun-4/260 [3].

At the heart of Almond-32 lies an ALU (Arithmetic Logic Unit) that boasts high-performance components, including an adder, a logical unit, a shifter, a comparator, and a multiplier. This assembly ensures that Almond-32 can handle a wide range of mathematical and logical operations swiftly and efficiently.

For a microprocessor to deliver peak performance, it must manage hazards seamlessly. Almond-32 excels in this aspect with a dedicated Hazard Management Unit. This unit encompasses a Hazard Detection Unit (HDU) and a Forwarding Detection Unit (FDU) and a smart Branch Prediction Logic to handle data, structural and control hazards, ensuring efficient operation.

Moreover, branch prediction is a crucial feature for enhancing processor performance. A Branch History Table (BHT, see chapter 8) is incorporated to optimize branching instructions, enabling the processor to work more effectively.

In the following sections, detailed description of the designed microprocessor Almond-32 is provided, along with its features.

In the first part, general characteristics and features that define the Almond-32 microprocessor are outlined. Subsequently, an examination of its block diagram and an in-depth analysis of the specific components is presented, while justifying design choices in terms of performance and optimization. Indeed, Almond-32 takes this seriously by implementing choices aimed at minimizing power consumption, which is a crucial factor in today's devices.

CHAPTER 2

Insturction Set

2.1 Registers and Instructions format

The DLX architecture employs 32 integer general-purpose registers, denoted as R_0, R_1, R_2, and so on, up to R_{31}. The register R_0 always holds the value 0, so it can be read, but it cannot be overwritten.

The microprocessor uses only two addressing modes:

- Immediate: the immediate is specified into the instruction itself
- Displacement: the value is added to the content of the register

All the instructions are represented on 32 bits with a 6-bit primary opcode and are grouped into three main types:

- R-type, which includes ALU operation
- I-Type, which includes load, store, operations with immediate, jump to a register and branch instructions
- J-Type, which includes jump and jump and link instructions

In the following, the format of instructions is explained.

For R-type instructions:

- 6-bit primary OPCODE (always 0x00 for R-type instructions).
- 5-bit RS1, indicating the address of the first source register.
- 5-bit RS2, indicating the address of the second source register.
- 5-bit RD, specifying the address of the destination register.
- 11-bit FUNC to configure the ALU.

For I-type instructions:

- 6-bit primary OPCODE.
- 5-bit RS, specifying the address of the source register.
- 5-bit RD, indicating the address of the destination register.

- 16-bit IMMEDIATE, which contains a constant value.

For J-type instructions:

- 6-bit primary OPCODE.
- 26-bit IMMEDIATE, representing the offset added to the Program Counter (PC).

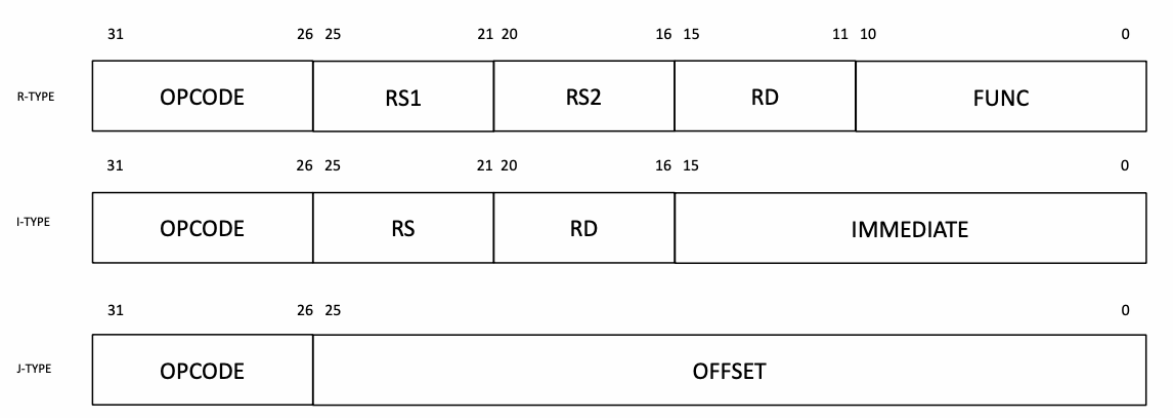

Figure 2.1: Instruction Format

2.2 Instruction Set

Table 2.1 lists all the instructions supported by Almond-32 in its basic version. Table **??** shows the subset of instructions supported by the processor in the PRO version, that is the one designed.

OPCODE	FUNC	INSTRUCTION	DESCRIPTION
0x00	0x04	sll	$rd \leftarrow rs1 << rs2\_27..31$ (unsigned)
0x00	0x06	srl	$rd \leftarrow rs1 >> rs2\_27..31$ (unsigned)
0x00	0x20	add	$rd \leftarrow rs1 + rs2$ (signed integers)
0x00	0x22	sub	$rd \leftarrow rs1 - rs2$ (signed)
0x00	0x24	and	$rd \leftarrow rs1 \& rs2$ (unsigned)
0x00	0x25	or	$rd \leftarrow rs1 \mid rs2$ (unsigned)
0x00	0x26	xor	$rd \leftarrow rs1$ XOR $rs2$ (unsigned)
0x00	0x29	sne	$if(rs1! = rs2)\ rd \leftarrow 1$ else $rd \leftarrow 0$ (signed)
0x00	0x2c	sle	$if(rs1 <= rs2)\ rd \leftarrow 1$ else $rd \leftarrow 0$ (signed)
0x00	0x2d	sge	$if(rs1 >= rs2)\ rd \leftarrow 1$ else $rd \leftarrow 0$ (signed)
0x02		j	$PC \leftarrow PC + imm26$ (signed)
0x03		jal	$R31 \leftarrow PC + 4;\ PC \leftarrow PC + imm26$ (signed)
0x04		beqz	$if(rs == 0)\ PC \leftarrow PC + imm16$
0x05		bnez	$if(rs! = 0)\ PC \leftarrow PC + imm16$
0x08		addi	$rd \leftarrow rs + imm16$ (signed)
0x0a		subi	$rd \leftarrow rs - imm16$ (signed)
0x0c		andi	$rd \leftarrow rs \& uimm16$ (unsigned)
0x0d		ori	$rd \leftarrow rs \mid uimm16$ (unsigned)
0x0e		xori	$rd \leftarrow rs$ XOR $uimm16$
0x14		slli	$rd \leftarrow rs << uimm16\_27..31$ (unsigned)
0x15		nop	idles one cycle
0x16		srli	$rd \leftarrow rs >> uimm16\_27..31$ (unsigned)
0x19		snei	$if(rs! = imm16)\ rd \leftarrow 1$ else $rd \leftarrow 0$ (signed)
0x1d		sgei	$if(rs == imm16)\ rd \leftarrow 1$ else $rd \leftarrow 0$ (signed)
0x23		lw	$rd \leftarrow M[imm16 + rs]$
0x2b		sw	$M[imm16 + rs] \leftarrow rd$

Table 2.1: Basic instruction set

OPCODE	FUNC	INSTRUCTION	DESCRIPTION
0x00	0x07	sra	$rd \leftarrow (rs1\_0)\ \hat{}\ rs2\#\#(rs1 >> rs2)\_rs2..31$ (signed)
0x00	0x21	addu	$rd \leftarrow rs1 + rs2$ (unsigned)
0x00	0x27	adc	$rd \leftarrow rs1 + rs2 + C$ (signed)
0x00	0x28	seq	$if(rs1 == rs2)rd \leftarrow 1 elserd \leftarrow 0$ (signed)
0x00	0x2a	slt	$if(rs1 < rs2)\ rd \leftarrow 1$ else $rd \leftarrow 0$ (signed)
0x00	0x2b	sgt	$if(rs1 > rs2)rd \leftarrow 1 elserd \leftarrow 0$ (signed)
0x00	0x3a	sltu	$if(rs1 < rs2)rd \leftarrow 1 elserd \leftarrow 0$ (unsigned)
0x00	0x3b	sgtu	$if(rs1 > rs2)rd \leftarrow 1 elserd \leftarrow 0$ (unsigned)
0x00	0x3c	sleu	$if(rs1 <= rs2)rd \leftarrow 1 elserd \leftarrow 0$ (unsigned)
0x00	0x3d	sgeu	$if(rs1 >= rs2)rd \leftarrow 1 elserd \leftarrow 0$ (unsigned)
0x01	0x03	mult	$rd \leftarrow rs1 * rs2$ (signed)
0x09		addui	$rd \leftarrow rs + imm16$ (unsigned)
0x12		jr	$PC \leftarrow rs$(unsigned)
0x13		jalr	$R31 \leftarrow PC + 4; PC \leftarrow rs$
0x17		srai	$rd \leftarrow (rs\_0)\ \hat{}\ uimm16\#\#(rs >> uimm16)\_uimm16..31$ (signed)
0x18		seqi	$if(rs1 == imm16)rd \leftarrow 1 elserd \leftarrow 0$ (signed)
0x20		lb	$rd \leftarrow (signextended)M[imm16 + rs]$
0x24		lbu	$rd \leftarrow 0\hat{}\ 24\ \#\#M[imm16 + rs]$
0x25		lhu	$rd \leftarrow 0\hat{}16\ M[imm16 + rs]$
0x28		sb	$M[imm16 + rs] \leftarrow rd\_24..31$ (signed)
0x0b		subui	$rd \leftarrow rs - uimm16$ (unsigned)
0x0f		lhi	$rd \leftarrow imm16\#\#\ 0\hat{}16$
0x1a		slti	$if(rs < imm16)\ rd \leftarrow 1$ else $rd \leftarrow 0$ (signed)
0x1b		sgti	$if(rs > imm16)rd \leftarrow 1 elserd \leftarrow 0$ (signed)
0x1c		slei	$if(rs <= imm16)rd \leftarrow 1 elserd \leftarrow 0$ (signed)
0x21		lh	$rd \leftarrow$ (sign extended) $M[imm16 + rs]$
0x29		sh	$M[imm16 + rs] \leftarrow 16rd\_...31$
0x3a		sltui	$if(rs1 < imm16)\ rd \leftarrow 1$ else $rd \leftarrow 0$ (unsigned)
0x3b		sgtui	$if(rs1 > imm16)\ rd \leftarrow 1$else $rd \leftarrow 0$ (unsigned)
0x3c		sleui	$if(rs <= imm16)\ rd \leftarrow 1$else $rd \leftarrow 0$ (unsigned)
0x3d		sgeui	$if(rs == imm16)\ rd \leftarrow 1$ else $rd \leftarrow 0$ (unsigned)

Table 2.2: Pro Instruction Set

2.3 Pipeline

This pipelining strategy is a key feature of the Almond-32 architecture, allowing it to achieve higher performance, and enhance the processor's throughput and efficiency by minimizing idle time and optimizing the utilization of its resources. Almond-32 processor exhibits a five-stage pipeline.

Pipelining is based on the concurrent execution of different phases, such as instruction fetch, decode, execution, memory and write back. It assumes that these phases are independent across various operations and can be overlapped. However, in cases where this independence condition is not met, the processor may need to stall or exploit forwarding to manage dependencies (see Chapter 9 and 10). As a result, multiple operations can be processed simultaneously, with each operation being at a different phase of its execution.

CHAPTER 3

Datapath Overview

3.1 Almond-32 Datapath Implementation

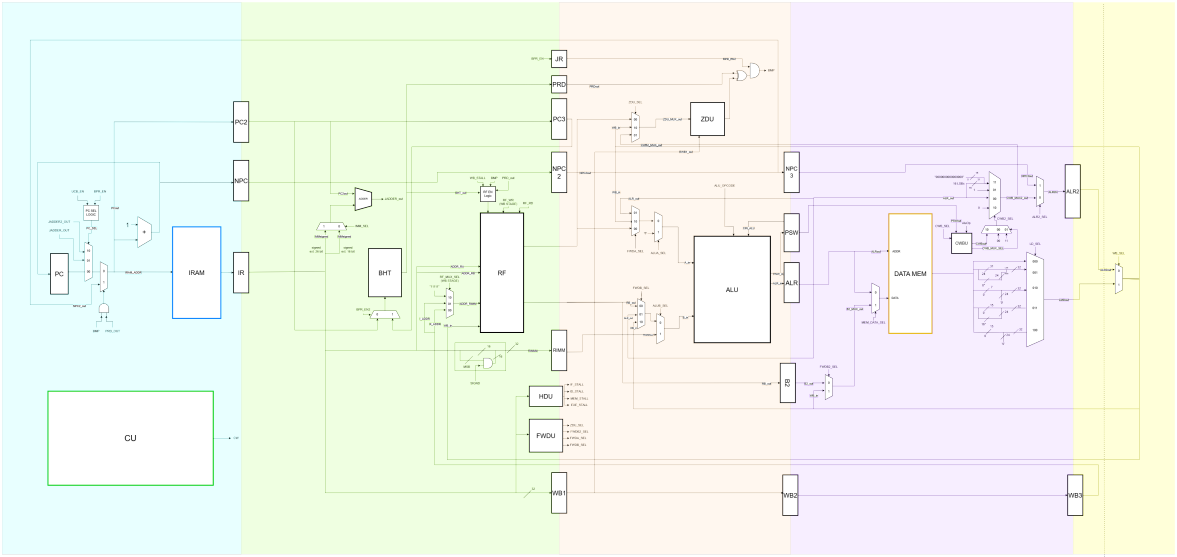

Figure 3.1: Block scheme of Almond-32 Datapath, highlight on the stages. For more readability refer to appendix.A

The funding principles on which Almond-32 is designed are derived from David A. Patterson and John L. Henessey work, the Deluxe processor [3].

The pipelining strategy adopted is a key feature of the Almond-32 architecture, allowing it to achieve higher performance, and enhance the processor's throughput and efficiency by minimizing idle time and optimizing the utilization of its resources. Pipelining is based on the concurrent execution of different phases, such as instruction fetch, decode, execution, memory and write back. It assumes that these phases are independent across various operations and can be overlapped. However, in cases where this independence condition is not met, the processor may need to stall or exploit forwarding to manage dependencies (see Chapter 9 and 10). As a result, multiple operations can be processed

simultaneously, with each operation being at a different phase of its execution. The five stages that compose Almond-32 datapath are:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execution (EXE)
- Memory (MEM)
- Write back (WB)

Between each pipeline stage, registers are placed to maintain instruction integrity. Their enable signals are controlled by the Control Unit (CU, see chapter 4) and can be overridden by the Hazard Detection Unit (HDU, see chapter 10) to implement stalls in case of hazards.

3.1.1 Instruction Fetch Stage

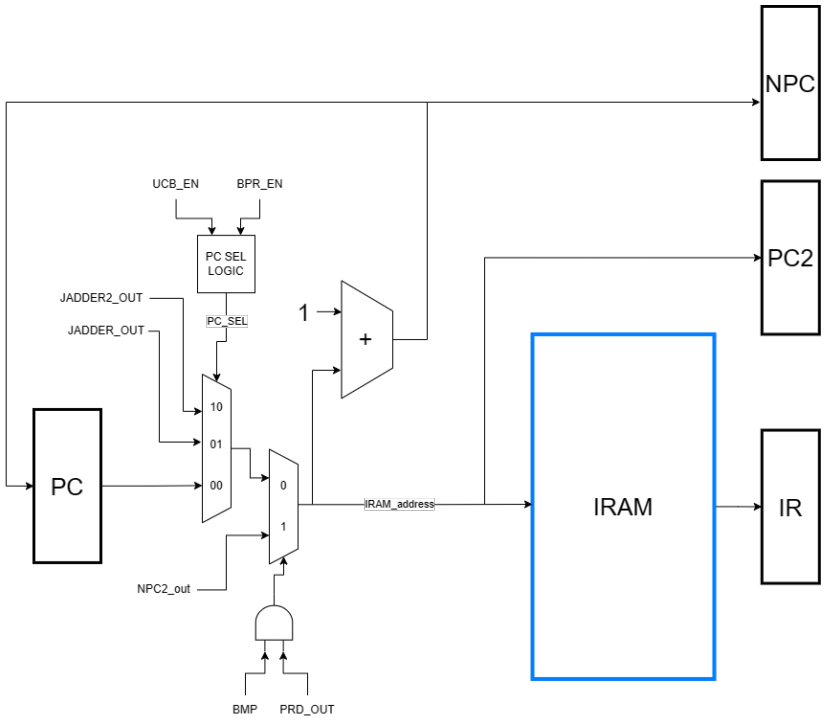

Figure 3.2: Block scheme of Almond-32 Instruction Fetch Unit. The Instruction RAM, here highlighted in blue, is not a part of the datapath

The purpose of the Instruction Fetch unit is to fetch the correct instruction from the Instruction RAM, described in detail in chapter 13. In order to do so it makes use of a Program Counter register (PC) which during regular instruction flow is incremented every cycle. In the case of a disruptive

instruction, such as conditional and unconditional branches, two multiplexers are used to divert the PC and to ensure the correct fetching.

- PC_MUX: driven by a logical combination of UCB_EN and BPR_EN, it enables the selection of the computed jump address during unconditional and predicted taken conditional branches.
- IRAM_MUX: driven by PRD_OUT *and* BMP (Branch MisPrediction bit, section 3.1.3 for additional information), it plays a crucial role in branch misprediction management. In the case of a branch misprediction, this MUX uses the NPC of the jump instruction in order to restore the correct program flow.

An important aspect to point out is that in a traditional DLX architecture the PC is incremented by 4, due to the IRAM being byte addressable. Almond-32 design is focused on the processor features and optimizations, thus it uses a word addressable high-level designed RAM and needs to increment the PC by one every cycle. With respect to the ALU P4-inspired adder, the PC adder needs to perform a sum with a fixed operand. For this reason, the system uses a much higher-level-described adder in order to leave more freedom to synthesis tools, thus improving the optimization of the component.

3.1.2 Instruction Decode Stage

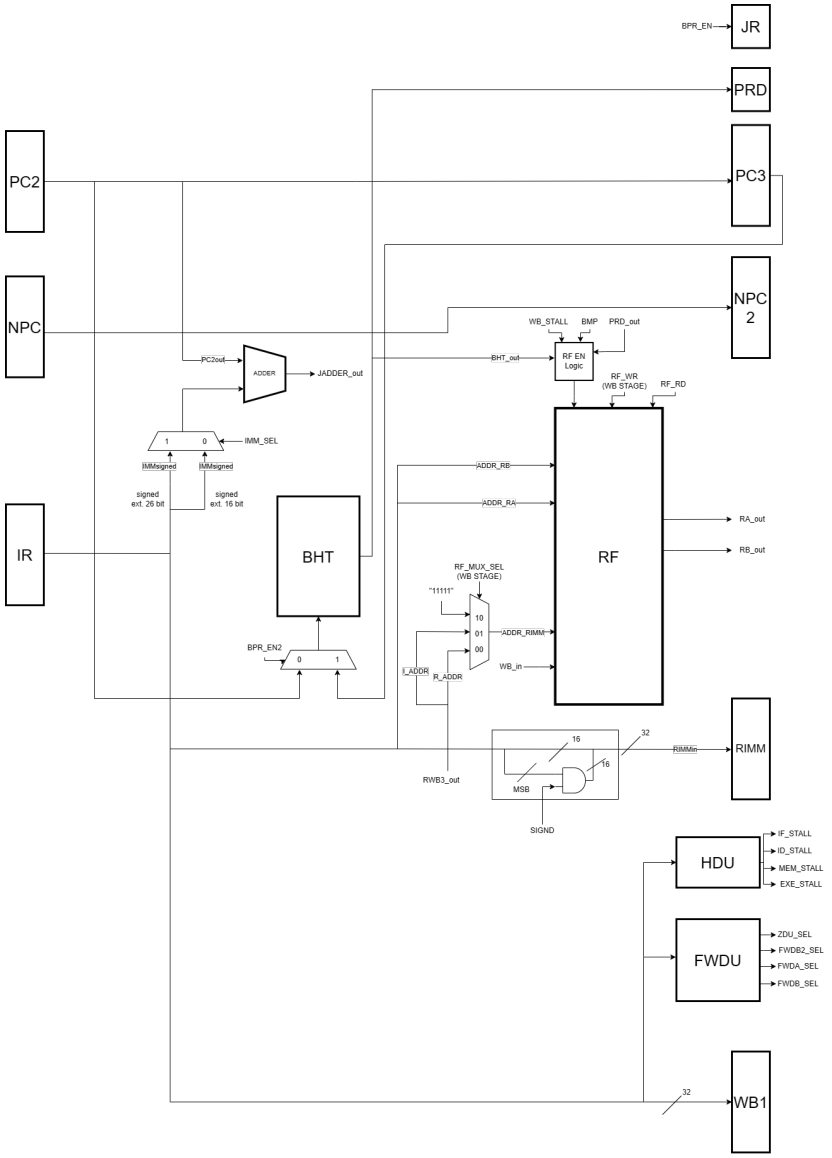

Figure 3.3: Block scheme of Almond-32 Instruction Decode Unit.

The Instruction Decode unit is responsible for several crucial features of Almond-32. Its main purpose is to obtain the operand data for the current instruction, using sections of IR_out as indexes to read from the register file (chapter 5).

Register File management

The register file is usually enabled but during specific situations, like mispredictions or stalls, a specific logic overrides `RW_EN` and disable the RF. The writing operation is enabled by the signal `RF_WR`, set by the CU during the WB stage. While the data to be written back in the RF derives from the WB stage output, the address is set by the multiplexer IMM_MUX.

Depending on the signal `RF_MUX_SEL`, this multiplexer can connect the address of the RF either to the destination derived from the current instruction or to '11111'. This last option is used in the Jump and Link instruction to save the current PC in the 31th register.

Branch management

During the ID stage, in the case of a conditional branch the Branch History Table is activated to obtain a prediction the outcome. In order to eliminate the delay of the branch, a specific adder is deployed to compute the target address. This solution enables Almond-32 to fetch the branched instruction effortlessly, without loosing a single cycle. More information about this feature is provided in chapter 8.

A multiplexer is used to connect the adder either to a signed extension of the whole 26bit IMM field of the instruction or to a smaller 16bit one. The first case is used with unconditional branches, while the second one is needed for conditional ones.

Hazard management: HDU and FWDU

One of the most impressive feature of Almond-32 is the ability to detect and fix every type of hazard. During the ID stage, the Hazard Detection Unit (see 10) and the Forwarding Unit (see 9) acquire the instruction in order to perform hazard detection and implement stalls or forwarding.

3.1.3 Execution Unit

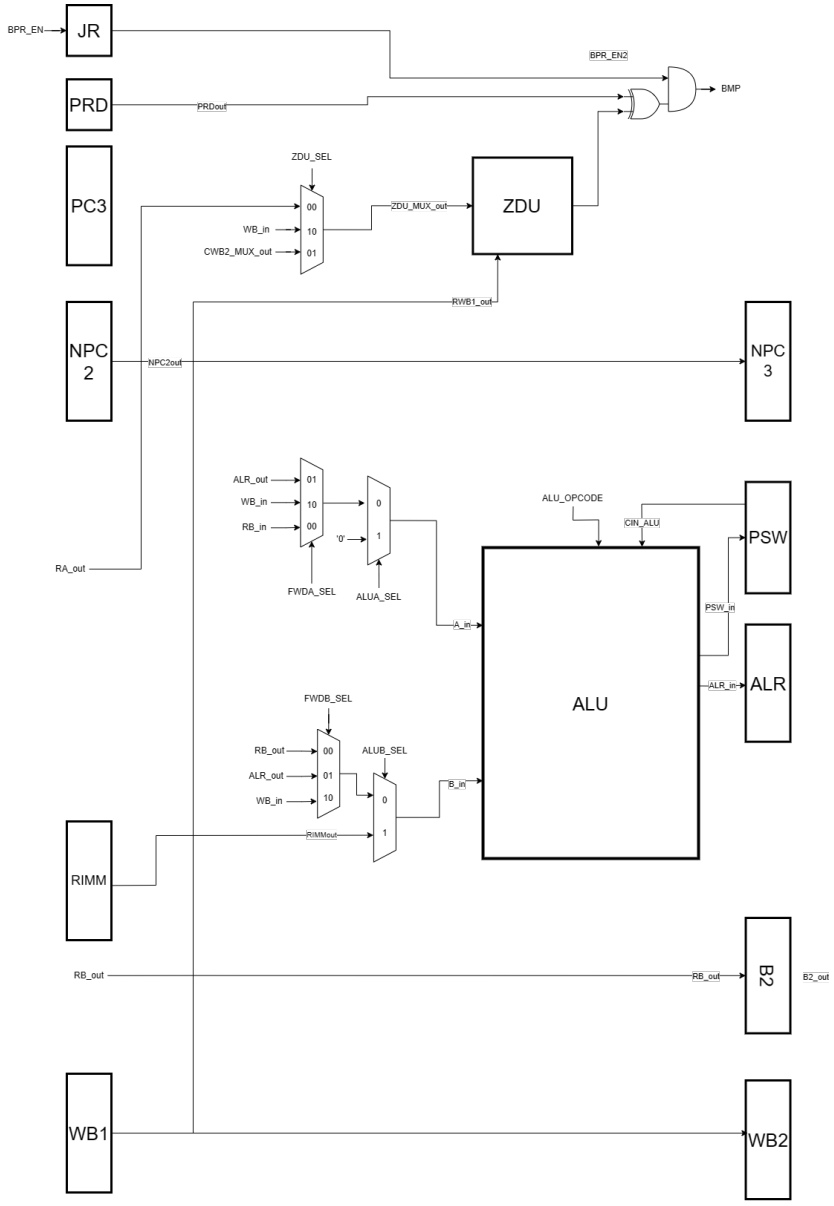

Figure 3.4: Block scheme of Almond-32 Execution Unit

The Execution unit has a central role in the architecture, since it includes every functional component and it's essentially the stage that enables any computational task.

Arithmetic Logic Unit

The focus of this unit is without doubt the ALU, described in detail in chapter 6. It's important to underline the role of the ALU inputs' multiplexers:

- FWDA_MUX and FWDN_MUX: they are driven by the FWDU to enable forwarding, as described in chapter 9.

- RegAMUX: used to connect the operand A of the ALU either to FWDA_MUX_out or to '0'. This last option is only used for LHI instruction, see chapter 2 for additional information.
- RegBMUX: used to connect the operand B of the ALU either to FWDB_MUX_out or to RIMM_out. This last option is used when I-type ALU instruction are executed.

Zero Detection Unit

To evaluate the actual outcome of a branch prediction, the Zero Detection Unit is employed. As the name implies, it compares the value of an operand to zero and produces an indication of the equivalence, depending on whether the instruction is a BNEZ or a BEQZ.

Branch Misprediction bit

Once that branch outcome has been evaluated by the ZDU, it needs to be compared to the actual prediction. A Branch Misprediction bit is thus set using a suitable logic to signal a misprediction. A mechanism is then activated using multiplexers to refetch the correct instruction and to ensure operational integrity (see 3.1.1).

3.1.4 Memory Unit

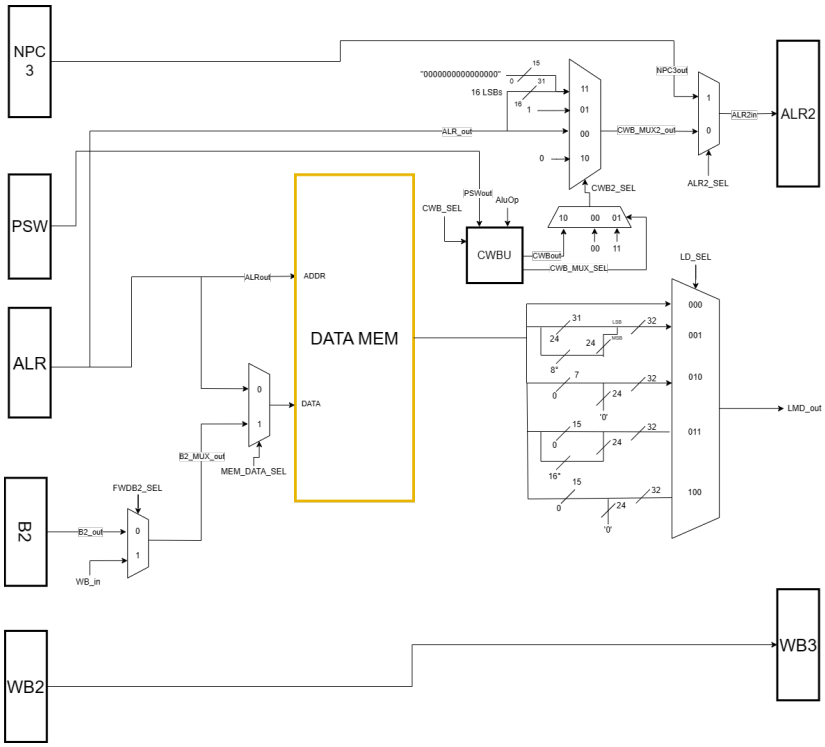

Figure 3.5: Block scheme of Almond-32 Execution Unit

The Memory unit is in charge of carry out store and load instructions correctly, interfacing with the Data RAM in an appropriate way.

Byte Addressable Memory Operations

As described in chapter 12, the DRAM of a Almond-32 is organized as a Big Endian, byte addressable memory. In order to manage byte, half word or word *store* instructions, a Byte Lane Controlled signal, driven by the CU, is used to communicate this information to the DRAM. To manage byte, half word and word *load* instructions, on the other hand, the system uses a 5-to-1 multiplexer, LMD_MUX. This component, driven by the CU signal, LD_SEL, connects the output of the memory stage to one of several hardwired configurations that rearrange correctly the data read by the DRAM.

Conditional Write Back Unit

This unit is in charge of driving the CWB_MUX2, the multiplexer used to carry out set instructions, e.g. SGE, SLT etc, by setting accordingly the MUX input to either 1 or 0. This MUX is also employed to perform the LHI instruction, described in chapter 2, i.e. it shifts the 16 LSBs of the input to the 16 MSBs of the output, setting the remaining bits to 0s.

Jump and Link Management

In the case of a JAL instruction, the next program counter of said instruction needs to be written back in register R31. In order to do so, ALR2 multiplexer is driven by the CPU to connect the register ALR2 to NPC3.

3.1.5 Write Back Unit

This unit is composed only of a single multiplexer, WBMUX, which is in charge to connect either ALR2 or LMDout to the RF input port and every other unit that requires feedback from this stage. During the write back stage, the register file is once again activated by setting the correct enable and write signals.

CHAPTER 4

Control Unit

The Control Unit (CU) is a vital component of a processor responsible for managing the system's status and generating the necessary commands to control the datapath's functions. It serves as the central controller, coordinating communication between various components, including memory and input/output blocks, to enable the system to execute its intended operations. The CU's general structure consists of two main conceptual elements:

- a **sequencer** that reads inputs, which can be information about the system's status or an instruction, and generates inputs for the command generator
- a **command generator** that produces the next commands the system is expected to execute

4.1 Hardwired Control Unit

Almond-32 utilizes a hardwired Control Unit, which employs a fixed, hardware-based approach. In this setup, the instruction register content serves as an address encoding to access a lookup table (LUT) containing control words associated with each instruction. As depicted in Figure 4.1, the 6-bit opcode is used for instruction decoding and LUT addressing.

The advantage of a hardwired CU lies in its speed and simplicity. It eliminates the need to fetch microcode from memory, enhancing processing speed. However, this approach lacks flexibility since any changes to control logic require hardware modifications, making it less adaptable to changes in the instruction set architecture.

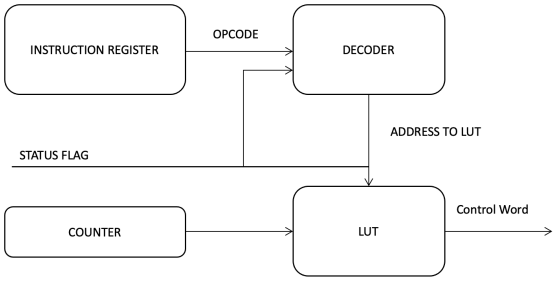

Figure 4.1: Hardwired Control Unit

4.1.1 Managing the Pipeline

To manage the pipeline and ensure that the control words are delivered to each stage at the correct clock cycle, a series of registers are used. This delay allows the control signals to be released in subsequent steps, while the CU can simultaneously generate a new control word without losing the previous one.

Below is a list of the output control signals to be released for each stage of the pipeline.

Decode:

- `ID_EN`: enables the registers in this pipeline stage.
- `RF_RD`: enables reading from the Register File.
- `SIGND`: distinguishes between operations with signed and unsigned operands (please refer to 3 to understand the related logic behaviour).
- `IMM_SEL`: serves as the selector for a MUX that extends the instruction's IMMEDIATE value to 16 or 26 bits (signed).
- `BPR_EN` (Branch Prediction Register Enable): used to turn on the BHT
- `UCB_EN` (Unconditional Branch Enable): active for jump instruction (please refer to chapter **??**)

Execute:

- `EX_EN`: enables the registers in the execute stage.
- `ALUA_SEL`: used as selector of 2-to-1 MUX, which selects the first operand for the ALU, either from the Register File (RA) or 0.
- `ALUB_SEL`: used as selector of 2-to-1 MUX, which selects the second operand for the ALU, either from the Register File or the immediate value from `RIMM`.

Memory:

- `MEM_EN`: enables the registers in the memory stage.
- `MEM_DATA_SEL`: used as selector of a MUX 2-to-1, which selects the input data to the DRAM from the ALU ('0') Register File (or forwarding process if necessary) ('1').
- `MEM_RD`: enables memory (DRAM) read.
- `MEM_WR`: enables memory (DRAM) write.
- `CS`: the chip select signal for memory.
- `MEM_BLC0` and `MEM_BLC1`: Configure store instructions in memory based on the Byte Lane Control (BLC).(see 12.2 for details).
- `LD_SEL0`, `LD_SEL1`, `LD_SEL2`: they are 3 bits of selector signal of MUX 5-to-1 (`LMD MUX`), which selects the configuration at the memory output for different types of load insturctions. (see section 12.3)
- `ALR2_SEL`: selector for a MUX to store the right value in register `ALR2`.
- `CWB_SEL0` and `CWB_SEL1`: 2 bits of the condition write back selector (see 11)

Write Back:

- WB_SEL:
- WB_SEL: selects the data to write back, either from memory ('1') or ALR2 ('0').
- RF_WR: enables writing to the Register File.
- RF_MUX_SEL0 and RF_MUX_SEL1: select the Register File address to write, depending on the instruction type. It serves to choose the correct destination register in case R-type or I-type instruction and to select register R_{31} in case of

4.1.2 ALU Opcode

The Control Unit also generates the output for the ALU_OPCODE port, which is sent directly to the ALU to configure it and enable it to perform various operations (e.g. AND, OR, ADD, SUB).

If the opcode of the instruction coming from the Instruction Register (IR) is 0x00, it indicates an R-type instruction, and the FUNC field is decoded to generate the correct ALU Opcode.

If the opcode of the instruction in the IR is equal to 0x01, the FUNC field is once again used to determine if it's a mult operation. In this case, the ALU Opcode is generated to configure the ALU for multiplication.

For all other cases, where FUNC is not used, a specific ALU Opcode is generated solely based on the instruction's opcode to ensure the correct configuration of the ALU.

Since the ALU Opcode is generated during the decode phase, but the ALU operates in the execute one, registers are used to implement the pipeline and ensure the synchronization of these operations.

4.1.3 Reset of Control Words

The CU has an asynchronous, active-high reset input denoted as RST. When the reset signal is '1', all control signals listed for each stage of the pipeline are reset, effectively setting the control words in each pipeline register to zero. The ALU Opcode is also set to that associated with the NOP instruction.

Reset Due to Branch Misprediction

One of the CU's input ports is BMP (Branch Misprediction). Branch instructions are managed by the processor, as explained in later chapters. For now, it's important to know that when BMP is '1', it indicates a misprediction has occurred. In this case, the control word in the execute stage needs to be reset before the pipeline can proceed. The BMP bit is forced to '0' after the CW reset is completed.

Control Word Management in Case of Stall

Another input port of the CU is STALL. If a RAW hazard (Read-After-Write hazard) is detected, STALL is set to "01", as explained in 10 In this scenario, only the control words in the execute and memory stages are shifted to the next pipeline stage.

CHAPTER 5

Register File

A Register File (RF) is a fundamental component within a microprocessor that serves as a bank of registers. Registers are high-speed memory elements that store temporary data and are essential for executing instructions.

RF in Almond-32 stores a set of 32 general-purpose integer registers (R_0, R_1, ..., R_{31}), which holds a data value consisting of 32 bits. It includes a special register, R_0, that always holds a value of zero and cannot be overwritten. Consequently, the 5-bit field in instructions dedicated to the destination register can never be a string of zeroes, unless it's an NOP (No-Operation) instruction. This behavior is common in processors and facilitates arithmetic and initialization operations.

RF is a sequential component with 10 input ports and 2 outputs.

Inputs:

- `CLK` receives the clock signal.
- `RESET` is connected to a synchronous and active-high reset signal.
- `ENABLE` is the port that receives the signal that enables/disables the RF operation. The choice made is to hardwire `ENABLE` to '1' to keep the register file always active. This simplifies the control logic of the component: there is no need to manage the enable/disable dynamics, reducing the circuit's complexity. Additionally, in a typical processor architecture, the registers of the Register File must be ready to respond to instructions very quickly. Constant enablement ensures instant data access, avoiding delays due to signal activation. As explained below, there are specific ports that allow disabling individual read or write operations based on the needs, i.e. the instruction to be executed at that stage of the pipeline.
- `RD1` is the port that receives the enable signal to enable reading at PORT1. It is a synchronous active-high signal.
- `RD2` is the port that receives the enable signal to enable reading at PORT2. It is a synchronous active-high signal.
- `WR` is the port that receives the enable signal to enable synchronous writing (active high).
- `ADD_WR` receives the 5-bit address of the register to write to ($5 = \log_2(32)$).
- `ADD_RD1` receives the 5-bit address of the register to read from PORT1 ($5 = \log_2(32)$).
- `ADD_RD2` receives the 5-bit address of the register to read from PORT2 ($5 = \log_2(32)$).

- `DATAIN` receives the 32-bit data to write to the indicated register when the write-enable signal is active.

Outputs:

- `OUT1` is the output port that allows synchronous reading at PORT1.
- `OUT2` is the output port that allows synchronous reading at PORT2.

Indeed, since RF is connected to the ALU, which usually has two input operands and one output, the typical choice is to design a register file with two access ports in reading and one access port in writing.

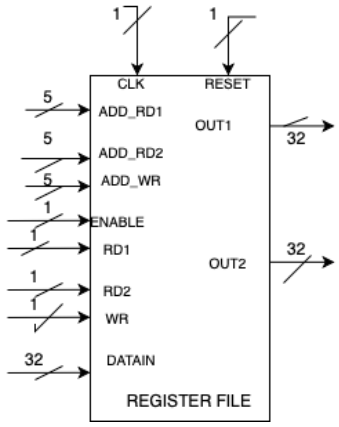

Figure 5.1: Register File

5.1 Read and Write Operations

It is possible to read and write at the same clock cycle: the Register File allows both writing of data to a specific register and reading from two distinct registers to occur simultaneously in the same clock cycle.

The choice of implementing simultaneous operation enables efficient pipelining in the processor, with one instruction in the decode stage (A) and another in the write-back stage (B) being able to access the Register File at the same time.

When enable slignals (`ENABLE`, `WR`, `RD1`, `RD2`) that determines whether write and read operations can be executed are active, reads from the registers occur on the rising edge of the clock signal (`CLK`), while writes operations occur on the falling edge of the clock signal.

CHAPTER 6

ALU

The Arithmetic Logic Unit (ALU) is a fundamental component of a processor that performs arithmetic and logic operations on binary data.

ALU in Almond-32 includes the following units:

- Logical Unit
- Shifter
- Adder
- Comparator
- Multiplier

The ALU Control Unit generates control signals within the ALU depending on the ALU Opcode generated by the CU. In particular, there is an enable signal for each of the components listed above, allowing only the required block for the operation to be activated, thus avoiding power consumption.

Furthermore, the selector signal for the 4-to-1 MUX at the ALU's output is generated. Depending on the operation being executed, the output of the corresponding unit is selected.

Conversely, the output of the comparator and the carry-out generated by the adder arrive at the ALU's `COND` and `Cout` ports, respectively, to then be stored in the PSW, as indicated in Chapter 11.

The figure Fig.6.1 depicts the architecture of the ALU present in the processor.

6.1 Logical Unit

The Logical Unit is inspired by the one present in the T2 ALU. The T2 implementation uses only NAND gates to implement the following logical functions: AND, NAND, OR, NOR, XOR, XNOR.

Figure 6.2 shows the schematic.

Let A and B be the operands represented as 32 bits, and S0, S1, S2, S3 be the 4 bits of the selection signal. By combining the inputs of the NAND gates, as shown in Figure 6.2, the output is given by the equation 6.1.

$$out = \overline{A} \cdot \overline{B} \cdot S0 + \overline{A} \cdot B \cdot S1 + A \cdot \overline{B} \cdot S2 + A \cdot B \cdot S3 \tag{6.1}$$

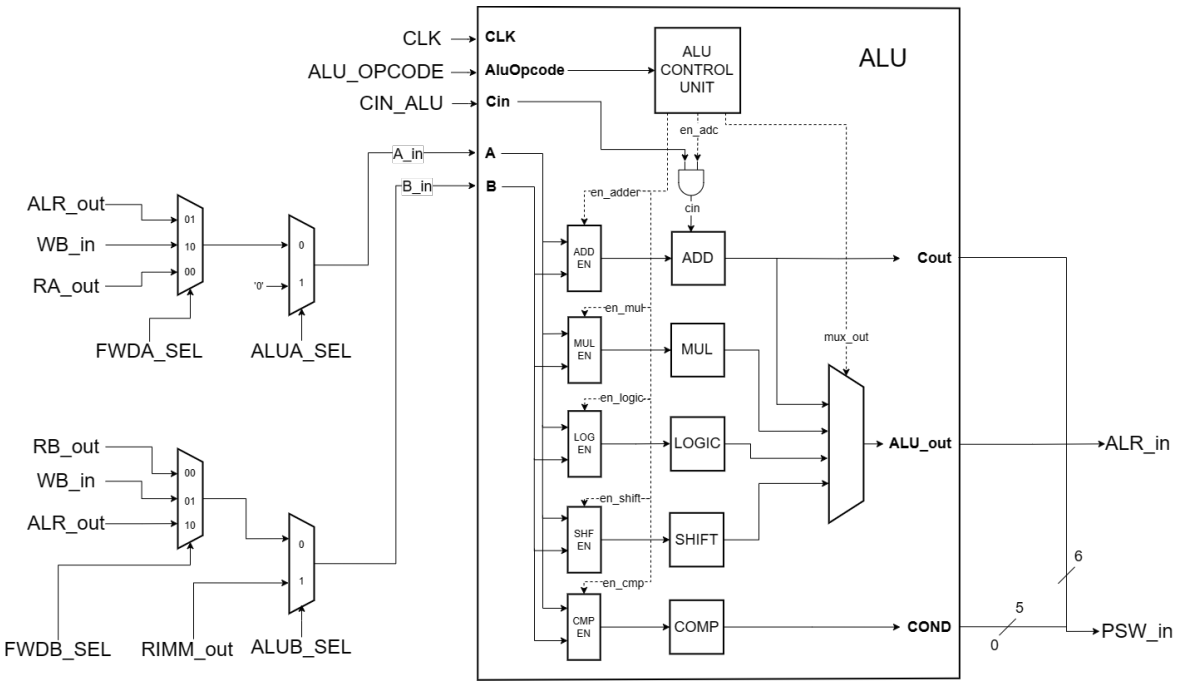

Figure 6.1: ALU Schematic

S3	S2	S1	S0	LOGIC
1	0	0	0	AND
0	1	1	1	NAND
1	1	1	0	OR
0	0	0	1	NOR
0	1	1	0	XOR
1	0	0	1	XNOR

Table 6.1: Logical Functions depending on S

Depending on the combination of bits S_i in the selection signal, the logical functions indicated in Table 6.1 are obtained.

It should be noted that thanks to this implementation, the logic transformation and selection phases are combined, and that only NAND gates and inverters to negate operands are required..

In contrast to the diagram in Figure 6.2, during synthesis, multiple NAND gates grouped together will be used. The goal is to create a balanced distribution of paths in terms of depth and delay to reduce glitches and power dissipation.

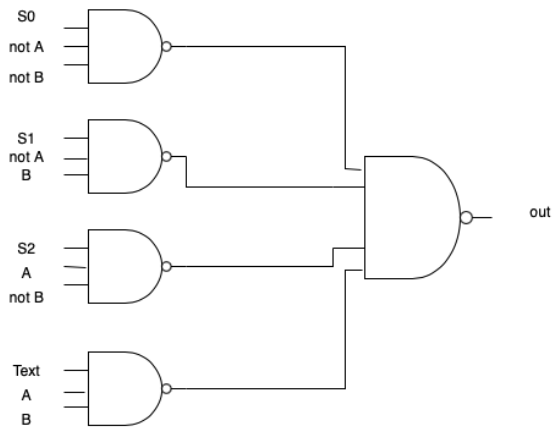

Figure 6.2: Logical Unit

6.2 Shifter

The shifter plays a pivotal role in facilitating efficient data manipulation and processing within a computer's ALU. It is a fundamental component designed to perform the shifting of binary data to the left or right by a specific number of position. It is essential for executing a wide range of mathematical and logical operations.

Shifting operations can be broadly categorized into two main types:

- **Logical Shifts**: bits are moved left or right, and the vacant bit positions are typically filled with zeros.
- **Arithmetic Shifts**: unlike logical shifts, arithmetic shifts preserve the sign of a number, and so vacant bit positions are filled with copies of the original sign bit.

In particular, the shifter in Almond-32 processor follows the T2 shifter architecture.

The instruction set includes the following operations: SLL (Shift Left Logical), SRL (Shift Right Logical), SLA (Shift Left Arithmetic), SRA (Shift Right Arithmetic).

Shifter in Almond-32 has the following ports:

- `DATAIN`, which receives input data to be shifted (the first operand)
- `R`, which receives the 6 least significant bits of the second operand, in order to define the shift amount
- `CONF`, which receives a 2-bit configuration signal to define if the shift should be right, left, arithmetical or logical
- `DATAOUT`, which is the unique output port with the shifted value represented in 32 bits

Step 1: Masks generation Depending on the configuration signal, 8 possible masks are generated on 40 bits, each already shifted of 0, 8, 16, ... 56.

Step 2: Coarse grain shift Depending on the three most significant bits of R, it chooses among the 8 masks the nearest to the shift to be operated. R[5,4,3] is the selector of the MUX which performs this selection process.

Step 3: Fine grain shift Depending on the three least significant bits of R, the correct 32 bits of the mask are selected to obtain the desired shifted output. This is accomplished by a MUX with R[1,2,0] as the selector.

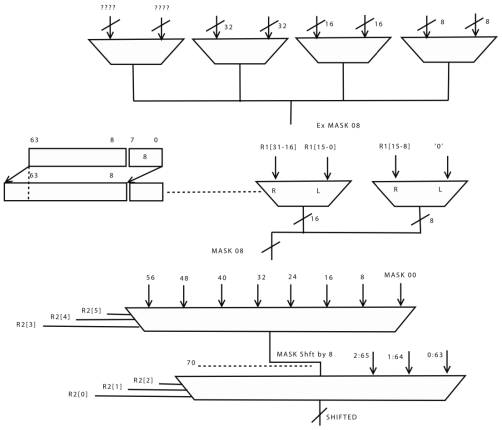

Figure 6.3: Shifter Schematic

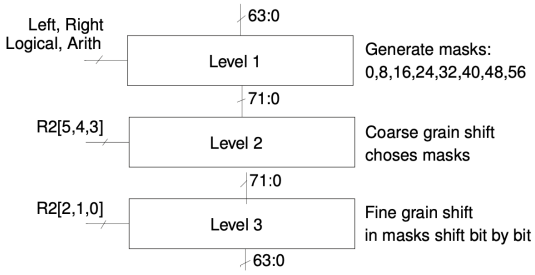

Figure 6.4: Shifter Implementation

6.3 Adder

Adders are commonly used digital components within computer processors. Essentially, an adder is a digital circuit designed to calculate the sum of two numbers, each typically represented using N bits.

In particular, the adder in our processor follows the Pentium4-Adder algorithm [4]. It has 4 inputs (`A`, `B`, `ADD_SUB`, `Cin`) and 2 outputs (`S`, `Cout`).

- `A` and `B` are the two input operands, represented using N = 32 bits.
- `ADD_SUB` is a 1-bit signal that selects whether to perform addition (`ADD_SUB` = 0) or subtraction (`ADD_SUB` = 1) between the two operands.
- `Cin` is the input carry (1 bit).
- `S` is the result of the operation (A + B or A - B) as a 32-bit value.
- `Cout` is the carry out (1 bit).

The adder consists of two major blocks:

- Carry generator
- Sum generator

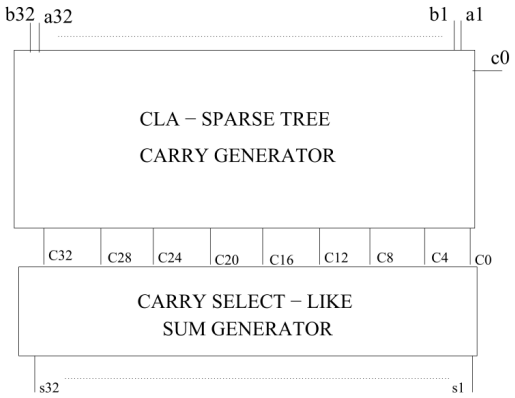

Figure 6.5: Carry generator and Sum generator

6.3.1 Carry generator

The carry generator is composed of blocks referred to as `PG, G, P BLOCKS`, connected as shown in Fig. 6.6. These blocks are based on propagate and generate terms.

In particular, given a_i and b_i as the i-th bits of operands A and B (32 bits), the following terms are defined:

- Propagate term: $p_i = a_i \oplus b_i$
- Generate term: $g_i = a_i \cdot b_i$

These terms are generated by the so-called PG network.
Now, one can define the so-called "carry operators" as follows:

- $G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$
- $P_{i:j} = P_{i:k} \cdot P_{k-1:j}$

where

- $i \geq k > j$
- $G_{x:x} = g_x$
- $P_{x:x} = p_x$
- $g_0 = c_{IN}$ and $p_0 = 0$

Each PG block receives two pairs of inputs ($P_{i:k}, G_{i:k}$ and $P_{k-1:j}, G_{k-1:j}$) and generates two outputs ($P_{i:j}, G_{i:j}$). These are intermediate results and not the final carries.

Each G block receives 3 inputs ($P_{i:k}, G_{i:k}$ and $G_{k-1:j}$) and generates $G_{i:j}$ as output, which is one of the final carries and will serve as an input to the sum generator.

The 32-bit adder unit contained in each section of the ALU employs a sparse radix-2 carry-merge tree, which generates every fourth carry in the adder. This architecture speeds up the critical carry path by moving a substantial portion of carry-merge logic from the main carry-tree to a noncritical side-path.

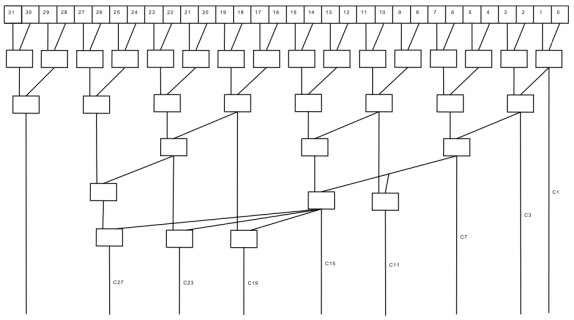

Figure 6.6: Sparse-Tree in P4 Adder

6.3.2 Sum generator

For an N-bit adder, the sum generator is composed of m blocks, each consisting of a carry selector. Each carry selector takes as input k bits ($k = \mathrm{N}/m$) from operands A and B, and one of the carry-outs from the Carry generator. It generates the corresponding k bits of the final sum.

Each carry selector consists of:

- Two Ripple-Carry Adders (RCA) of k bits each.
- A MUX 2-to-1.

The two RCAs calculate the sum of the input operands, considering one case with $Cin = 0$ (RCA0) and the other with $Cin = 1$ (RCA1). The two results serve as inputs to the MUX. The carry produced by the Carry generator, which corresponds to the selector of the MUX, determines whether the actual Cin is 0, resulting in the output being the sum calculated by RCA0, or 1, resulting in the output being the sum calculated by RCA1.

The advantage of using carry-select adders comes from dividing the N-bit adder into m groups of k bits each. This reduces the delay due to carry propagation.

6.4 Comparator

The comparator is a digital circuit that allows comparing two input numbers. Given operands A and B, the comparator provides outputs for the following operations:

- $A = B$
- $A \neq B$

- $A > B$
- $A \geq B$
- $A < B$
- $A \leq B$

The idea is to leverage the adder already present in the ALU.

The comparator takes as input the difference between A and B, produced by the adder. So, using the nomenclature from the previous section (6.3), it receives `S` and `Cout`. It has 6 1-bit outputs, one for each of the operations listed above. The output corresponding to each operation will be 1 only if the respective relationship between A and B is satisfied.

By calculating the difference between the two operands, three possible cases can be identified:

1. `Cout` $= 1$ and `S` $\neq 0$
2. `Cout` $= 1$ and `S` $= 0$
3. `Cout` $= 0$

It can be shown that the first case is satisfied only if $A > B$, the second case only if $A < B$, and the third case only if $A = B$. Therefore, by connecting logic gates as shown in the circuit diagram in Fig.6.7, the desired results are obtained.

The NOR gate that takes the bits of the sum `S` from the adder as input allows evaluating whether it is equal to 0 or not: the output of the gate is 1 only if all the bits of `S` are equal to 0. Since we can't have a 32-input NOR gate, the one shown in Fig.6.7 will be synthesized using multiple NOR gates.

Given how the comparator works, whenever the ALU receives an instruction that requires the use of it, both the comparator and the adder's enable signals will be activated. Additionally, the `ADD_SUB` and `Cin` inputs of the adder will be set to '1' because it needs to compute the difference between the two operands.

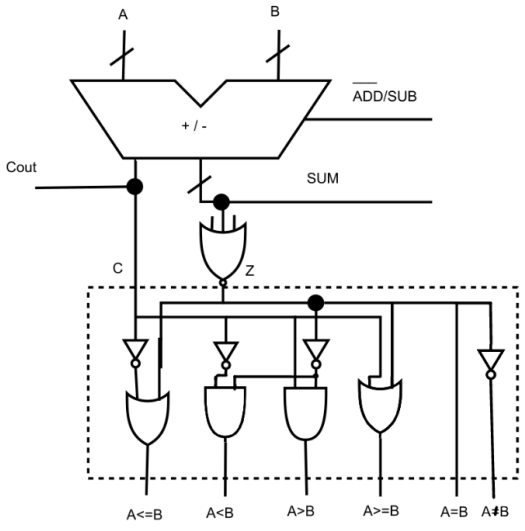

Figure 6.7: Comparator

6.5 Multiplier

The Almond-32 CPU offers also the possibility to perform signed multiplications between two 16-bit integers, and storing the 32-bit result into a register. The ALU selects the least significant halfword from both the input registers automatically, in order to produce results that don't fit into the register file.

The multiplier exploits the possibility of pipelining to reduce the critical path.

6.5.1 Booth's Algorithm

This unit implements the Booth's Algorithm [1] in order to increase the performance and reducing the latency (with respect to a more simple array multiplier). Radix-4 has been chosen due to simplicity reasons and to limit the area and power consumption, so the unit tackles 2 bits of the factors at a time.

6.5.2 Implementation

The hardware implementation is one of the most complex among the other execution units and the one with the highest latency. It consists in 3 parts :

- Addends generation unit : This unit generates all the possible addends that will feed the multiplexers. In fact, based on the output produced by the encoders, 5 possible addends could be chosen for each adder For every stage i (starting from i = 0), these are :

  1. 0
  2. $4 \cdot A \cdot (i+1)$
  3. $4 \cdot (-A) \cdot (i+1)$
  4. $4 \cdot A \cdot 2 \cdot (i+1)$
  5. $4 \cdot (-A) \cdot 2 \cdot (i+1)$

  To generate all these vectors, a consecutive left shift has been performed for both positive and negative operands. This is where the pre-computation occurs, offering advantages in terms of speed and performance.

- Encoder unit : This part is made of a series of encoders able to translate each triplet of bits of the 2nd operand into valid selectors for the multiplexers within the datapath section, in order to properly select the pre-computed addends of first operand. The logic behind the encoders is shown in figure where, through a simple Karnaugh map the truth table has been implemented.

- Datapath : This is the part where the actual computation occurs. Due to the nature of the algorithm, 7 (NBIT/2 - 1, with NBIT = 16) 32-bit adders are implemented in a cascade mode to store the partial results. Pentium 4 adders have been used to mantain omogeneity with the other unit. Moreover, as specified in the former section, 8 multiplexers are adopted to provide the right addends, with the respective selectors produced by the encoders. As described in the following section, pipeline registers are included in this unit.

6.5.3 Pipeline

Due to the complexity of this execution unit, the propagation delay of the entire module would be extremely high with respect to the other units and hence reducing the maximum achievable frequency by a significant amount. To tackle this problem, a pipelined architecture with 4 stages has been

chosen instead, by grouping the 7 adders in pairs and using 3 internal pipeline registers. The addends generated need also to be properly synchronized with the clock, so other registers are inserted in cascade-like manner at the output of the muxes.

This way, it is possible to improve the throughput to 4 and to increase the maximum frequency of the module, at a cost of having a latency equal to 4 clock cycles and some more area given by the registers. The behaviour of the pipelined multiplier is shown in Figure A.2 where the test bench generated 4 pairs of factors. It is clearly visible that the result of the first multiplication is ready after 4 clock cycles, while the other results are available right after.

It should be noted that due to this choice, the ALU block is fed with a clock signal that is only used by the multiplier.

The architecture of this module is shown in Figure 6.8. The rectangles in red are the pipeline register needed to synchronize every phase. As the deepness of the stages increases, more registers are needed at the output of the multiplexers, in order to make available the pre-computed (and already selected) addends in the right stage.

For simplicity reasons the addends generator and the encoder section are represented as blocks. There are 3 internal registers but the overall latency is 4 due to the presence of the ALR out register at the output of the ALU, which delays the result by one further clock cycle.

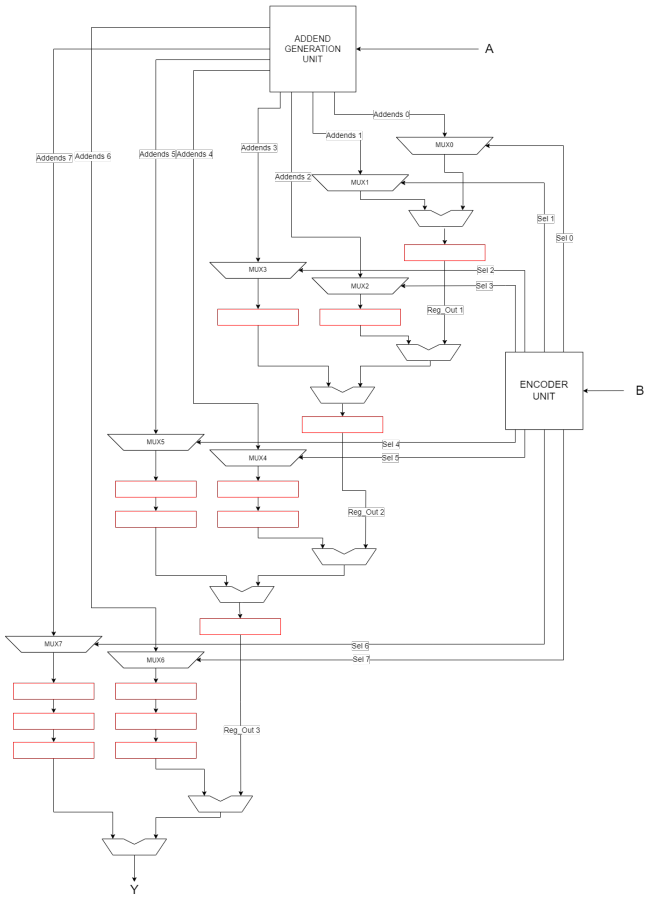

Figure 6.8: Architecture of the pipelined Multiplier

CHAPTER 7

Zero Detection Unit

The Zero Detecion Unit (ZDU) functions as a specialized logic module within the Arithmetic Logic Unit (ALU), specifically designed to detect whether a given binary data element, represented using N bits, holds a value of zero or not. In Almond-32, N is equal to 32 bits.

Efficiently detecting zero values is a crucial requirement in various computational and decision-making processes, particularly in the context of conditional branching.

Instructions that rely on the ZDU for their operation are BEQZ (Branch Equal Zero) and BNEQZ (Branch Not Equal Zero).

In our processor, the ZDU receives two inputs:

- the content of the register in RF (32-bits)
- the LSB of the opcode of the instruction currently in the execution stage.

The ZDU generates a 1-bit output signal, which transitions to '1' only when the specified condition is satisfied.

- If the instruction in the execution stage is of the BEQZ type, characterized by an opcode of 0x04, the LSB passed to the ZDU is 0. In this scenario, the ZDU produces a '1' at its output if the content of RF is equal to 0; otherwise, the output is '0'.
- Conversely, if the instruction in the execution stage is a BNEQZ instruction, identified by an opcode of 0x05, the LSB input to the ZDU is 1. In this case, the ZDU outputs '1' if the content of RA is not equal to 0; otherwise, it produces '0'.

It is important to highlight that the ZDU operates continuously, ensuring that the zero detection capability is always available for immediate use.

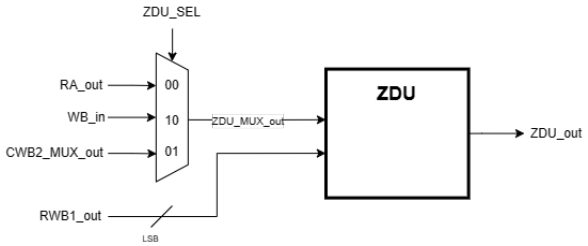

Figure 7.1: Zero Detection Unit

CHAPTER 8

Branch History Table

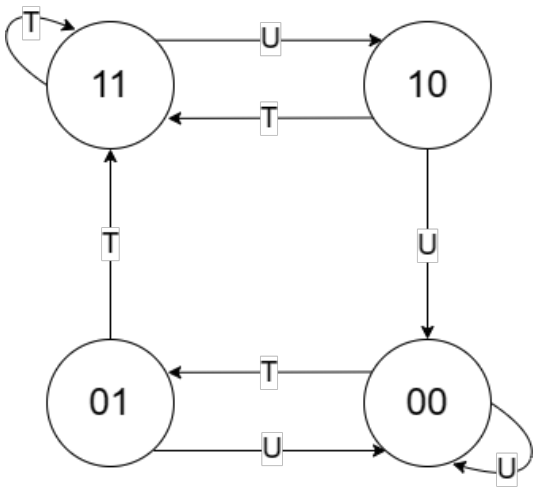

Figure 8.1: Scheme of the 2-bit BHT Finite State Machine. U stands for *Untaken* while T stands for *Taken*

8.1 Role and Design of BHTs in Modern Processors

A Branch History Table (BHT) is a key component in modern processors, used to predict the outcomes of branch instructions, particularly conditional branches. Its primary goal is to reduce the pipeline latency by reducing the impact of branch mispredictions.

The BHT is composed of a small memory, which is accessed via the last n-bits of the branch instruction's PC and contains an m-bit record of the past outcomes of the branch. Depending on the record's value, the branch is predicted taken or untaken.

The standard size of a BHT is 2 bits since it is demonstrated that a wider record has little advantages in terms of prediction capabilities. For this reason, the index's number of bits is the base 2 logarithm of the number of the BHT entries. Any lower-order bits or word offset bits not relevant for branch prediction are usually discarded.

The prediction mechanism is activated during the instruction decode stage and consists in a 4 state Finite State Machine, as depicted in figure **8.1**.

Each state of the BHT's entry is interpreted as a different prediction:

- **Strongly Taken** (3: "11"), which indicates a strong likelihood that the branch will be taken.
- **Weakly Taken** (2: "10"), which suggests a tendency for the branch to be taken but with less confidence.
- **Weakly Not Taken** (1: "01"), which suggests a tendency for the branch not to be taken but with less confidence.
- **Strongly Not Taken** (0: "00"), which indicates a strong likelihood that the branch will not be taken.

The BHT initializes all branch predictions to the neutral state "00", to avoid any specific bias for or against any branch direction.

As branch instructions are executed, the BHT updates its entries based on the actual outcomes. This is crucial for maintaining the accuracy of predictions.

Prediction update happens in the following way:

- **Branch Taken**: if a branch is taken and the prediction is not "Strongly Taken" (the associated counter is smaller than 3), the prediction may be incremented, indicating an increased likelihood of the branch being taken in the future.
- **Branch Not Taken**: if a branch is not taken and the prediction is not "Strongly Not Taken" (the associated counter is greater than 0), the prediction may be decremented, indicating a decreased likelihood of the branch being taken in the future.

8.2 BHT implementation in Almond-32 processor.

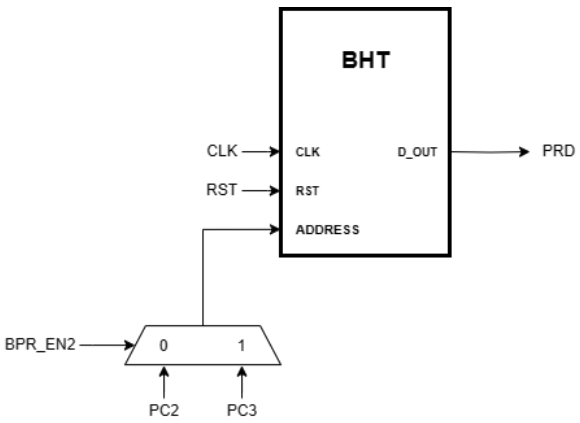

Figure 8.2: Block scheme of Almond-32 Branch History Table. `ADDRESS` derives either from the PC of the instruction in ID (PC2) or in EXE (PC3), to enable both prediction, during ID, and BHT entry update, during EXE.

The BHT operates during the decode phase and features 6 ports, including 5 inputs and one output.

- `CLOCK`: this port receives the clock signal.
- `RST`: it's an active-high asynchronous reset. When set to '1', the BHT is reset to all zeros.

- `D_IN`: this is a 1-bit input port, which is set to '1' when the branch outcome is taken and '0' otherwise. This input is used to update the BHT FSM, which in this case is implemented as a 2 bit saturating counter. If `D_IN` is '1' and the content in the corresponding cell is not equal to '11', the counter is incremented because it is not yet saturated. If `D_IN` is '0', the content of the corresponding cell is decremented by 1 if it's not already '00'. In Almond-32, the unit which evaluates the branch condition is the Zero Detection Unit (ZDU), thus `D_IN` is connected to its output.
- `W_EN`: This is the port which receives the write enable signal (1-bit). The BHT and update it only when this signal is high. In our case, it is connected to `BPR_EN2`, which is the `BPR_EN` signal from the CU, delayed by one clock cycle thanks to the flip-flop `JR`. In other words, the BHT is updated only if the instruction that was in the decode stage in the previous clock cycle was indeed a branch instruction.
- `ADDRESS`: this port receives the 32-bit address from the PC. In particular, if `BPR_EN2` = 1, we use the value of the PC currently contained in the `PC3` register; otherwise, `ADDRESS` receives the content of the `PC2` register.
- `D_OUT`: it is the 1-bit output port representing the prediction. The prediction is compared in the execute stage (hence, after being delayed by one clock cycle by the `PRD` flip-flop) with the output of the ZDU. If the two are different and `BPR_EN2` = 1 (a condition detected by a logic XOR gate followed by an AND gate), the misprediction signal (`BMP`) is set to '1'. `BMP` coincides with the output of the AND gate.

8.2.1 Latency improvements due to BHT

As stated in **8.1**, the main goal of a Branch History Table is to reduce the overall pipeline latency by predicting the outcome of a conditional branch instruction. In the Almond-32 architecture, a branch outcome is evaluated during the execution stage (EXE). This means that, without branch prediction, every BNEZ or BEQZ would cause the fetch of a wrong instruction, i.e. it would cause a one cycle delay. The following example depicts this situation.

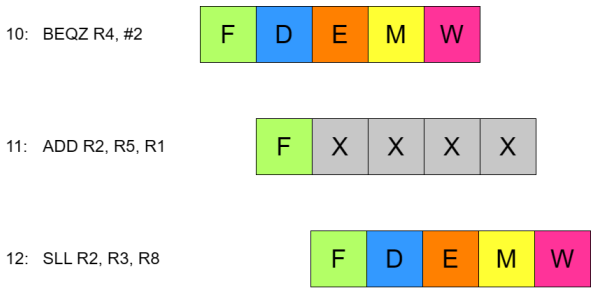

Figure 8.3: Without prediction, the outcome of BEQZ is evaluated during EXE, causing the erroneous fetch of ADD

In the case of a misprediction, the BHT looses a cycle as in the previous case. A correct prediction, on the other hand, eliminates the delay by fetching directly the correct instruction, as depicted below:

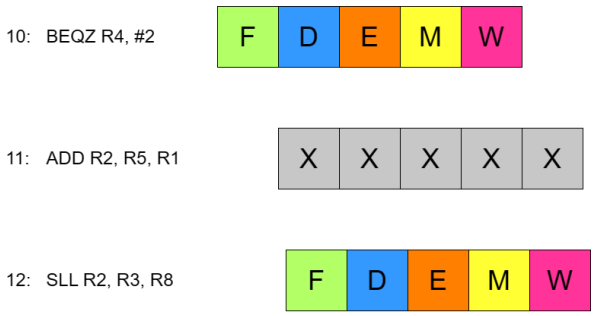

Figure 8.4: With a correct prediction, the outcome of BEQZ is evaluated during EXE, eliminating the erroneous fetch of ADD

The Branch History Table mechanism is particularly effective in *loops*, in fact after at most two iteration of the same branch instruction the BHT is able to predict correctly, thus effectively improving the pipeline efficiency.

CHAPTER 9

Forwarding Detection Unit

9.1 Data Hazards Significance in Operational Integrity

Data hazards are issues caused by data dependencies and like control or structural hazards, they have the potential to disrupt the functionality of the instruction flow.

They can occur when a data dependency between one or more instruction would cause an inconsistency and disrupt the functionality of the instruction flow. They can be categorized in:

- **RAW**, Read After Write: they correspond to a *true data dependence*, i.e. the operand of an instruction is the target of the previous one:

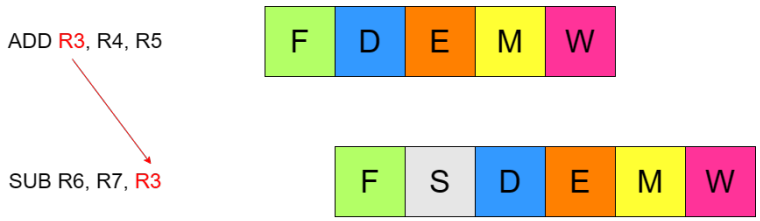

Figure 9.1: The ADD result is needed during the ID of the SUB

- **WAW**, Write after Write: they stem from *output dependencies*, i.e. an instruction could write into the destination befor the previous one, which shares the destination, has completed. However, these types of hazards are possible only if the architecture supports multi-cycle out of order instructions, which is not the case in Almond-32 architecture.

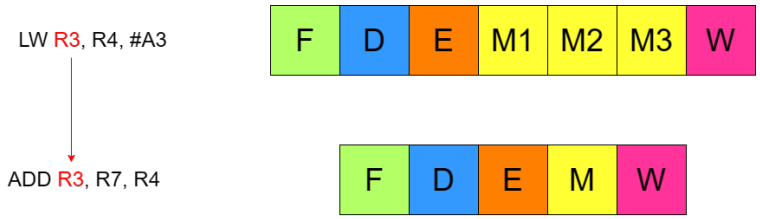

Figure 9.2: The ADD result is wrote back earlier than the LW result

- **WAR**, Write After Read: they are caused by anti-dependence between instructions, i.e. an instruction uses an operand modified by the following one. In Almond-32 architecture, as in most cases, anti-dependence doesn't generate any hazards.

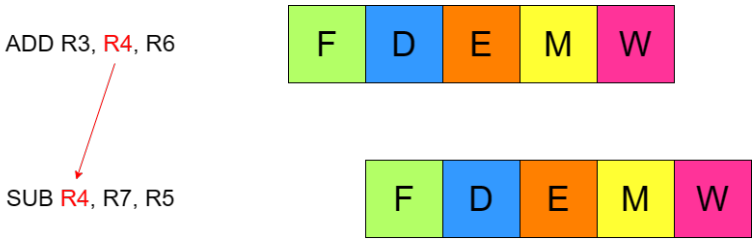

Figure 9.3: ADD needs an operand wrote by SUB

9.2 Forwarding Unit Implementation in Almond-32

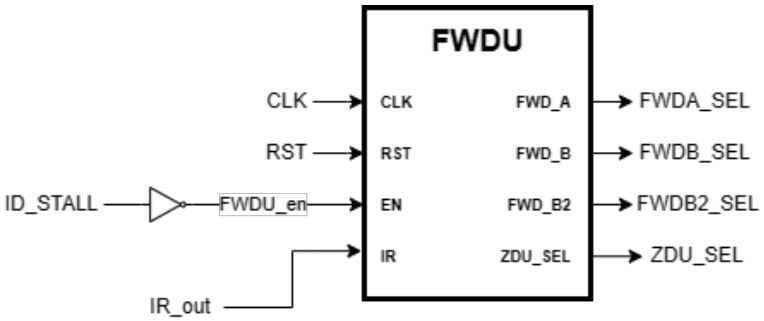

Figure 9.4: Block scheme of Almond-32 Forwarding Unit

In order to eliminate data hazards the forwarding mechanism can be exploited. This technique consist in sending the result of a computation or operation directly from one pipeline stage to another without having to wait for it to be written back to the register file or memory. Figure 9.5 shows a successful RAW hazard detection and elimination by forwarding data form the EXE stage of the first instruction to the EXE stage of the second one.

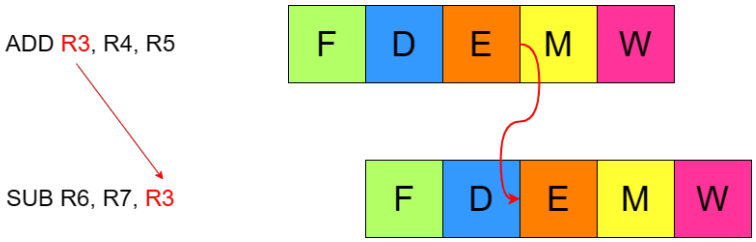

Figure 9.5: Thanks to forwarding, SUB can proceed without a stall and the RAW hazard is eliminated

The Forwarding Unit (FWDU) is a component of Almond-32 architecture that continuously monitors the data flow within the pipeline, playing a crucial role in identifying and managing data hazards, in particular RAW hazards. To address this, the FWDU can forward the required data from the appropriate pipeline stage to ensure the correct execution of the dependent instruction.

It receives as inputs:

- `CLOCK`
- asynchronous active-high `RESET`
- `EN`
- `IR`, that is the instruction in the decode stage at that clock cycle

It produces as outputs four selector signals:

- `FWD_A` (2 bits)
- `FWD_B` (2 bits)
- `FWD_B2` (1 bit)
- `ZDU_SEL` (2 bits)

All the selector signals mentioned above are generated in the decode stage (ID) and are then adequately delayed to present the correct value in the respective pipeline stage where they are needed. Specifically, `FWD_A`, `FWD_B`, and `ZDU_SEL` are delayed by one clock cycle, making then available during the EXE stage of the current instruction, while `FWD_B2` is delayed by two clock cycles, in order to make it available during the MEM stage.

9.2.1 Forwarding for ALU Operations

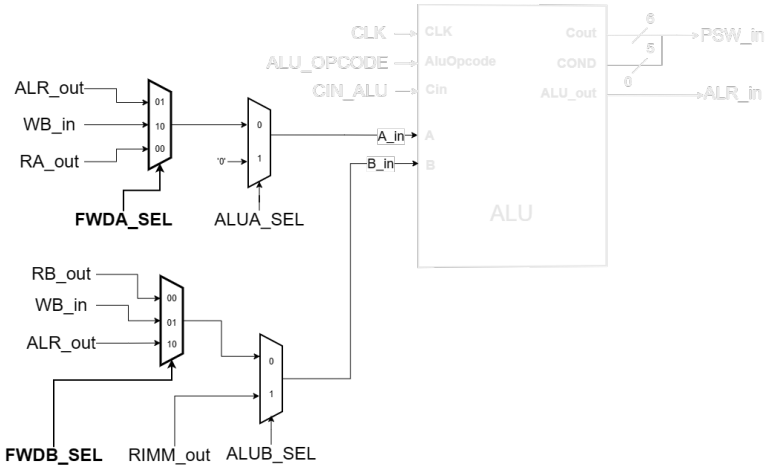

Figure 9.6: Highlight of the forwarding mechanism for the ALU via `FWDA_MUX` and `FWDB_MUX`

The signals responsible for the forwarding of ALU operations are `FWD_A` and `FWD_B`.

- `FWD_A`: this selector is connected to `FWDA_MUX` as seen in fig.datapath. When an instruction in EXE has the same OP1 (see instruction structure in appendix ??) of the destination of the instruction in ID, `FWDA_MUX` connects the operand A of the ALU to the output of the EXE stage, i.e. the signal `ALR_out`. On the other hand, when the same phenomena happens w.r.t. ID and MEM stages, the MUX connects the operand A to the output of the MEM stage, i.e. `WB_in`.

- `FWD_B` this selector is connected to `FWDB_MUX` as seen in fig.datapath. When an instruction in EXE has the same OP1 (see instruction structure in appendix ??) of the destination of the instruction in ID, `FWDB_MUX` connects the operand B of the ALU to the output of the EXE stage, i.e. the signal `ALR_out`. On the other hand, when the same phenomena happens w.r.t. ID and MEM stages, the MUX connects the operand B to the output of the MEM stage, i.e. `WB_in`.

Additional checks and conditions are implemented, e.g. on the absence of jump instructions, the presence of a load instruction or the difference between R-type instructions and I-type instructions, in order to ensure the correct behaviour of the FWDU. For the sake of simplicity these checks are not reported here, further information can be extracted from the VHD files if needed. In every case that a forwarding is not needed, `FWDA_MUX` and `FWDB_MUX` connects the ALU to the respective output ports of the register file.

9.2.2 Forwarding for Zero Detection Unit

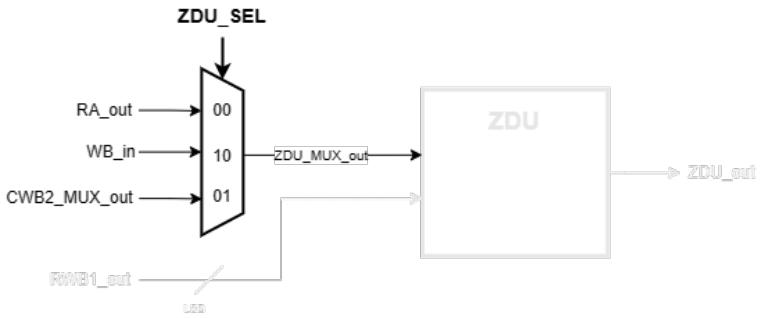

Figure 9.7: Highlight of the forwarding mechanism for the ZDU via `ZDU_MUX`

For the proper operation of the ZDU (Zero Detection Unit, 7), the FDWU plays a crucial role in identifying data hazards during the evaluation of conditional branches. Since the ZDU is external to the ALU, another MUX is needed in order to enable the correcti data forwarding, i.e. `ZDU_MUX`, driven by the signal `ZDU_SEL`.

Depending on the case, the FWDU could connect the ZDU input either to the output of the MEM stage, `WB_in` , or to `CWB_MUX2_out` , i.e. the output of the dual MUX chain that determines the outcome for every set-like instruction, e.g. `SGT`, `SLTE` etc.

This mechanism ensures that the ZDU receives the necessary data inputs, allowing it to perform zero detection accurately for branch instructions. The FWDU's role in selecting the input source based on data hazard detection enhances the efficiency and correctness of the pipeline's execution.

9.2.3 Forwarding for Store Instructions

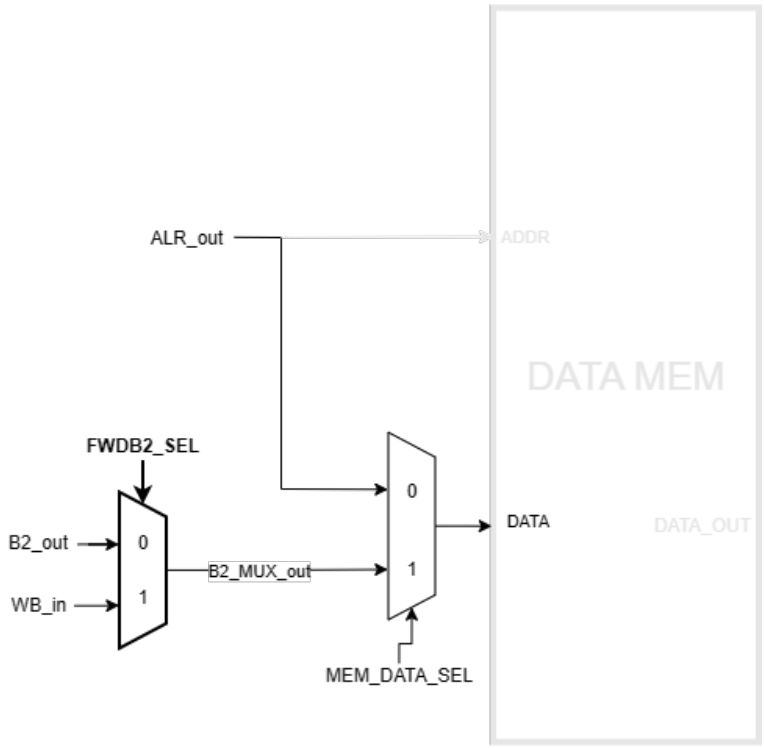

Figure 9.8: Highlight of the forwarding mechanism for the DRAM in case of a store instruction via `B2_MUX`

A store instruction requires saving a specific data value, or a portion thereof, from the register in the Register File, which we will denote as R[regb], to a particular memory address. The FWDU is in charge to detect and eliminate RAW hazards in the case of store instructions too.

In a similar manner to previous cases, the FWDU checks for correspondence between the source of the store instruction in the EXE stage and the destination of the instruction in the MEM stage. In order to do so, the multiplexer `B2_MUX` is driven by the signal `FWD_B2` to connect the data input of the DRAM to the output of the MEM stage itself, i.e. `WB_in` with the appropriate timing.

CHAPTER 10

Hazard Detection Unit

The Hazard Detection Unit (HDU) is a component in the Almond-32 architecture, responsible for identifying and addressing potential hazards that can disrupt instruction execution and lead to data inconsistencies.

In this specific context, two types of hazards are primarily dealt with: RAW (Read-After-Write) and Structural hazards.

Due to the architecture's focus on single-cycle operations and the absence of pipelined multi-cycle operations, more complex hazards like WAR (Write-After-Read) and WAW (Write-After-Write) do not occur.

This is achieved because single-cycle instructions are intentionally structured to write to registers sequentially. This design ensures that no new instruction can write to a register while a previous instruction is reading from it, and it guarantees that one instruction completes its write before another instruction attempts to write to the same register.

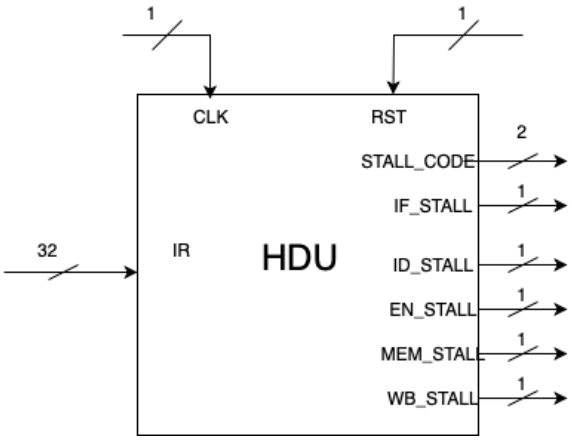

Figure 10.1: Hazard Detection Unit

10.1 Architecture of HDU

The Hazard Detection Unit (HDU) has the following ports:

- `CLK`: this port receives the clock signal to synchronize operations.

- `RST`: this port receives an asynchronous reset signal, active high, used to clear the memory.
- `IR`: this port receives the instruction in the decode stage.
- `STALL_CODE`: an output port that provides stall control codes.
- `IF_STALL`, `ID_STALL`, `EX_STALL`, `MEM_STALL`, `WB_STALL` (outputs): these ports are used to disable the registers in the corresponding stages of the pipeline when necessary, as explained in the following.

10.2 RAW Hazard

A RAW hazard takes place when a new instruction requires data from a register that is being written to by a previous instruction in the pipeline.

10.2.1 Hazard mitigation with forwarding

Forwarding is a technique used to resolve RAW hazards by directly transferring data from the execution stage of one instruction to the decode stage of another instruction, without introducing unnecessary pipeline stalls (refer to the Chapter 9 for more details).

Since the microprocessor includes the FDU, the only case in which one should stall an instruction is when it needs a register which is loaded by a previous instruction.

10.2.2 Managing RAW Hazard

The HDU in Almond-32 architecture employs a combination of signals and registers to track instructions in the pipeline and their dependencies. Based on their opcodes, the HDU determines whether a RAW hazard exists and whether a stall is required.

Output signals from each of the output port `IF_STALL`, `ID_STALL`, `EX_STALL`, `MEM_STALL`, and `WB_STALL` are connected to an `AND` gate with the enable signals from the CU, which serves as a control mechanism. The `AND` gate's output then controls the enable signals for the registers in their respective pipeline stages: when any of the stall signals are active (high), that stage is stalled.

Certainly, before being connected to an `AND` gate, the output stall signals of HDU are first negated to handle the fact that they are active-high.

10.3 Structural Hazard

A structural hazard occurs when multiple instructions require access to the same hardware resource, but it can only serve one request at a time.

In the context of the Almond-32 architecture, a structural hazard is observed when the ALU is needed for performing multiplication operations, since it is a not-pipelined multicycle one.

During the execution of multiplication, the HDU must manage the structural hazard by stalling instructions that are in the decode and fetch stages of the pipeline. This stall operation typically lasts for three clock cycles, to ensure that the ALU is reserved exclusively for the ongoing multiplication operation.

CHAPTER 11

Conditional Write Back Unit

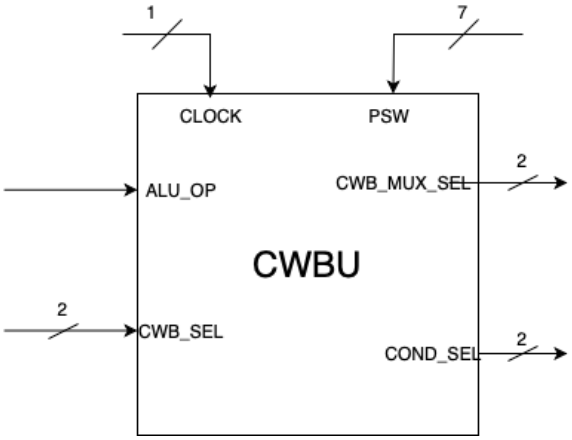

Figure 11.1: Block diagram of the Conditional Write Back Unit

The Conditional Write Back Unit (CWBU) serves as a driver for the write-back MUX in order to implement set operations, such as SGE, SLE, SNE, SEQ, SLT.

It needs to know which type of operation the ALU is performing, and the result of the Processor Status Word (PSW).

PSW contains the results of the operations performed by the comparator: each bit of the output `COND` of the ALU (refer to Chapter 6) corresponds to one of the first 6 bits of PSW (Table 11.1).

Operation	**Bit of PSW to store the result**
$A < B$	PSW(0)
$A \leq B$	PSW(1)
$A > B$	PSW(2)
$A \geq B$	PSW(3)
$A = B$	PSW(4)
$A \neq B$	PSW(5)

Table 11.1: PSW Mapping

Thus, the only inputs which are required are

- `CLOCK` to receive the clock signal.
- the ALU Opcode, generated by the Control Unit and sent to the ALU. A pipeline register is employed to synchronize the ALU operation.
- `PSW`, representing the Processing Status Word
- `CWB_SEL`

The generated outputs are two selectors for MUXs:

- `COND_SEL` (2 bits)
- `CWB_MUX_SEL` (2 bits)

When an operation such as SNE, SLT, SGT, SLE, or SGE is identified, the selector `CWB_MUX_SEL` is always set to "10".

Regarding the selector `COND_SEL`, it depends on the specific operation and the corresponding value of PSW. Please refer to Table 11.2.

If `COND_SEL` = "00", the normal operation proceeds without the need for the CWBU module.

If `COND_SEL` = "01" condition is verified, so data to be written back to the RF (`WB_in`) must be '1'.

If `COND_SEL` = "10" condition is not verified, so data to be written back to the RF (`WB_in`) must be '0'.

ALU Opcode	**PSW**	**COND_SEL**
OP_SNE	PSW(5) = '1'	"01"
OP_SNE	PSW(5) = '0'	"10"
OP_SLT	PSW(0) = '1'	"01"
OP_SLT	PSW(0) = '0'	"10"
OP_SGT	PSW(2) = '1'	"01"
OP_SGT	PSW(2) = '0'	"10"
OP_SLE	PSW(1) = '1'	"01"
OP_SLE	PSW(1) = '0'	"10"
OP_SGE	PSW(3) = '1'	"01"
OP_SGE	PSW(3) = '0'	"10"

Table 11.2: How `COND_SEL` is set depending on ALU Opcode and PSW

CHAPTER 12

DRAM

DRAM memory is comprised of an array of locations, holding 32 bits each. It has the following ports:

- `CLOCK`, to synchronize operations
- `RST`: a reset signal to clear the memory. È asincrono e attivo alto.
- `WR`: a write enable to write data to memory
- `RD`: a read enable to read data from memory
- `BLC`: the byte lane control, which determines how to store a word or a part of it (refer to section 12.2).
- `ADDR`: the address bus to select the memory location (32 bits)
- `DATAIN`: data to be written into memory (32 bits)
- `DATAOUT`: data read from memory (32 bits) (it is the only output port)

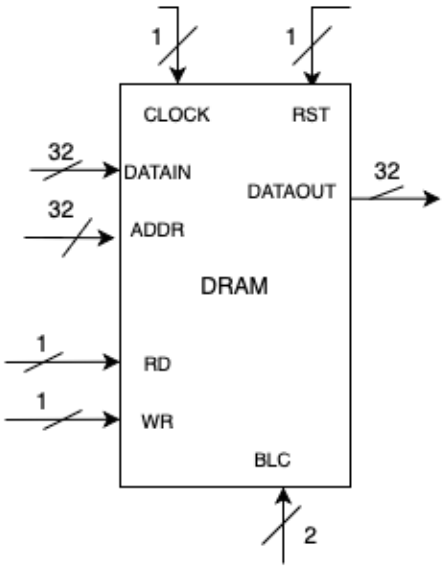

Figure 12.1: Hazard Detection Unit

12.1 Big-Endian Byte-Addressable Memory

In the Almond-32 architecture, DRAM is organized as a big-endian, byte-addressable memory.

Big endian memory organization means that the most significant byte of a multi-byte data word is stored at the lowest memory address, while the least significant byte is stored at the highest memory address. When reading a data word from memory, you start with the most significant byte at the lowest address and proceed to the least significant byte at the highest address. This organization is in contrast to little-endian, where bytes are stored in reverse order. Note that the bit stream represented by a given number of stored bytes doesn't attempt to be big- or little-endian.

Being *byte addressable* means that individual bytes in memory can be accessed as discrete addressable units.

12.2 Byte Lane Control for Store Instructions

Memory writes occur at the rising edge of the clock signal when the write enable is set to '1', and the reset is '0'. Assuming that there is a Memory Management Unit (MMU) that plays a crucial role in controlling how data is written to memory during store operations. For store instructions (store word, store byte, store half), the Byte Lane Control (BLC) is employed. This system is inspired by a feature present in ARM Synchronous Static Memory Controller (SSMC), an AMBA AHB module that provides an interface between an AMBA AHB system bus and external memory devices. [2]. In ARM application, it is used to specify which bytes of data should be written at a specific location within a memory word, plus additional information not required by Almond-32. This allows for flexible data management and precise control over memory allocation.

In this context, BLC is a 2-bit control signal generated by CU that defines three possible configurations, indicating which bytes or bits need to be written and how they should be positioned within the memory word.

- When BLC is set to "00," it indicates a full-word write operation. In this mode, the entire word (typically 32 bits) at the specified address is accessed and modified. All the bytes of the word can be read from and written to, provided the address is valid.
- Setting BLC to "01" signifies a byte write operation. In this mode, only the least significant byte of the word at the indicated address is accessed and modified. The other bytes in the word remain unchanged, provided the address is valid.
- When BLC is configured as "10," it corresponds to a half-word write operation. In this mode, both bytes (bits 15 downto 0) of the word at the specified address are accessed and modified, while the other bytes remain unchanged, given that the address is valid.

12.3 LMD MUX for Load instructions

In the Datapath, a 5-to-1 MUX is introduced to handle load instructions from memory. It selects the appropriate bytes based on the type of load instruction, by using the selector signal for this MUX is generated by the Control Unit.

- For `lw` (load word), represented by "000," the MUX retrieves a 32-bit word from memory.
- For `lb` (load byte), represented by "001," the MUX retrieves a single byte from memory.
- For `lbu` (load byte unsigned), represented by "010," the MUX retrieves a single byte from memory and extends it to 32 bits. This instruction treats the byte as unsigned data.

- For `lh` (load halfword), represented by "011," the MUX retrieves a 16-bit halfword from memory, expanding it to sign-extend it to 32 bits.
- For `lhu` (load halfword unsigned), represented by "100," the MUX retrieves a 16-bit halfword from memory, treated as unsigned data.

CHAPTER 13

IRAM

The Instruction RAM (IRAM) is responsible for storing and providing instructions for execution, playing a crucial role in the overall functionality of the microprocessor.

IRAM depth is choosen to be equal to 48. It stores instructions on 32 bits each.

It has 3 ports:

- `RST`: an input port that acts as an active-high reset signal for the IRAM. Memory initialization occurs when the processor is in a reset state.
- `ADDR`: an input port used to provide the address for reading instructions, if valid.
- `DOUT`: an output port that carries the instruction data read from the IRAM.

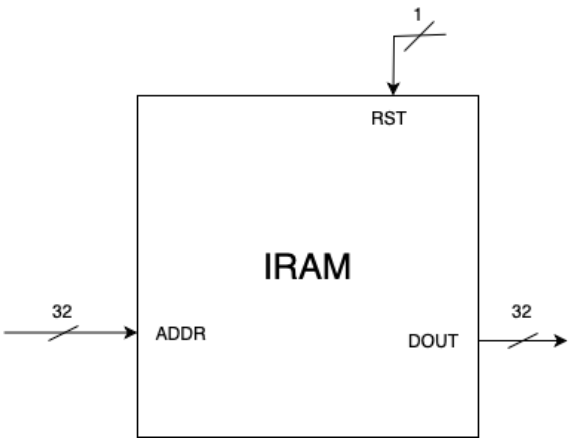

Figure 13.1: Hazard Detection Unit

CHAPTER 14

Synthesis

Synthesis was performed by using Synopsys Design Compiler. The whole process was made automatic by using the TCL Scripting Language. The analysis of every .vhd file has been executed through the script "analysis.scr" located in syn/scripts/analysis.scr.

Three synthesis has been performed in order to test the behaviour and the performances under various timing and power constraints :

1. Synthesis with timing constraint CLK = 2 ns (syn/scripts/compile.scr): this constraint is rather light because it does not set other restrictive constraint and was not forced with high efforts, but only the standard command *compile*.

2. Optimized synthesis with timing constraint CLK = 1.5 ns (syn/scripts/compile_OPT.scr) : the following step has been to use the slack obtained from the previous analysis in order to synthesize the design with more restrictive timing constraints and effort requested to the compile engine. This allowed us to achieve the lowest critical path possible and hence the maximum achievable frequency, at the cost of an increase in area and power used.

3. Optimized synthesis with power and timing constraints (syn/scripts/compile_POPT.scr) : this case represents the most elaborated synthesis version because of the quality of the generated netlist. In fact it sets limits both in time and power consumed, try to reduce power by about 30% (with upper power limit of 6 mW) with respect to the optimized version described at the point before, while forcing the clock period at 2 ns as in the first case.

   As expected, this synthesis required an quite impressive elaboration time but the result was particularly efficient since it almost obtained the desired power consumption while not affecting much the speed performances of the architecture.

The synthesis for each timing constraint were performed separately, cleaning the working environment before performing the following synthesis, to avoid the case of having different results every time. After each synthesis, we extracted the files .sdc and .v netlist in order to perform the Place and Route through the Cadence and Innovus environment.

14.1 Results

In the following table a summary of the timing, area and power analysis is reported. Note that the considered cases are the one listed above, in the same order. The 3^{rd} case present the probably the best scenario in terms of overall efficiency, because it consumes about 30% less than the first but mantains a very good performance in terms of timing.

CASE	TIME	CELLS	POWER	REPORT
1st case	0.9 ns	24029	6.28 mW	syn/results/area_DLX_NOOPT.txt
2nd case	0.71 ns	24095	8.59 mW	syn/results/area_DLX_NOOPT.txt
3rd case	0.88 ns	23400	6.29 mW	syn/results/area_DLX_NOOPT.txt

Table 14.1: Post synthesis analysis

CHAPTER 15

Placing and Routing

For executing the Place and Route, we utilized the software Innovus developed by Cadence within the working directory "innovus," where all the relevant files are stored. The steps we followed closely resemble those of Lab06 in the 2022/23 course. The primary difference lies in the ".globals" file, which is used to set the correct path to the netlist file created by Synopsys. The Place and Route process consists of the following steps:

- **Configuring:** We opened Innovus and imported the files "DLX.globals" and "dlx.v." These files contain the post-synthesis cell view, references to the layout view of the cell library (.lef file), and definitions of the power supply names, "vdd" and "gnd." Additionally, we included the "Default.view" file, which stores information related to the MultiMode MultiCorner (MMMC) analysis.
- **Structuring the Floorplan:** We defined the area to be assigned to the cell ensemble and the area where the Power Supply Ring will be routed. This step helps determine the number of rows needed for the design.
- **Inserting Power Rings:** We filled the channel with two metal rings for power supply and ground. Subsequently, we connected the metal stripes to VDD and GND pads, hierarchically distributing power and ground throughout the chip.
- **Inserting Stripes:** Vertical metal wires were added to connect the rings from top to bottom, serving as VDD and GND lines. This process helps distribute power and ground signals toward the center of the chip. The number of vertical stripes affects the distribution but can lead to congestion when routing other signals.
- **Standard Cell Power Routing:** This step involves placing horizontal wires to prepare VDD and GND wires for the standard cells. These wires will be connected to the external ring and the vertical stripes.
- **Placement:** Now, the cells are placed. Up to this point, the only known information was the total circuit area and the number of rows to be used. Each cell layout is assigned a unique position within one of the predefined rows.
- **Placing I/O Pins:** All I/O pins for all signals are positioned in this step.
- **Post Clock-Tree-Synthesis (CTS) Optimization:** Before the routing process begins, it's possible to optimize the design to meet required timing constraints.

- **Place Filler:** To ensure technological continuity, it's helpful to complete the placement with filler cells. These elements fill the empty spaces in the layout, ensuring continuity of the N+ and P+ wells in each row.
- **Routing:** All the cells are connected, taking into consideration the available metal layers.
- **Post-Routing Optimization:** In the final step, we attempted to optimize the design to meet the required timing constraints.

15.1 Results

After all the following steps the final result is the one shown in Fig (without showing the external I/O pins).

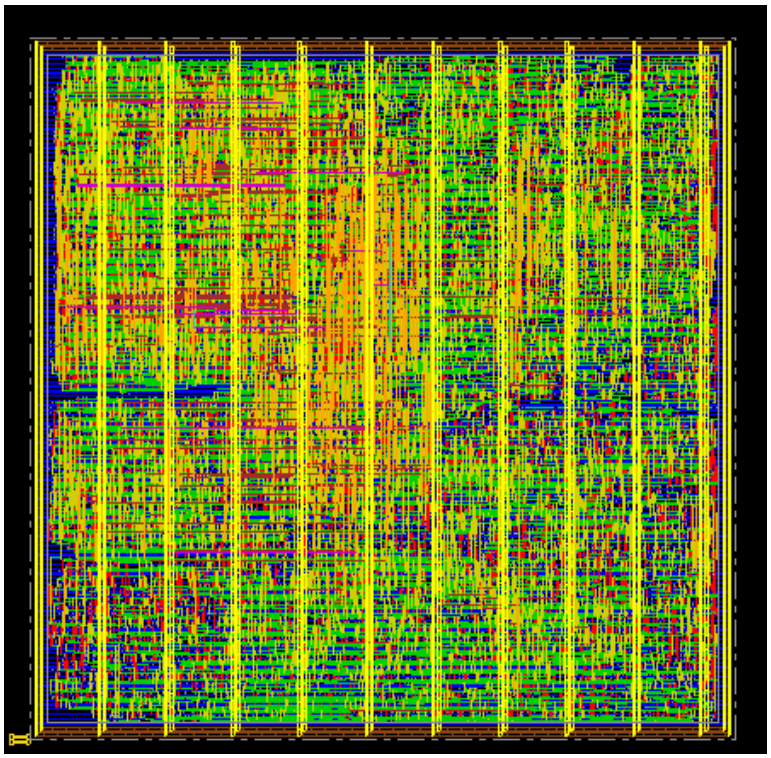

Figure 15.1: Almond-32 DIE (external pins not shown)

TYPE	RESULT	REPORT
Connectivity	OK	layout/analysis/DLX.conn.rpt
DLX Gate count	27851	layout/analysis/DLX.gateCount
DLX Area	22225 um^2	layout/analysis/DLX.gateCount
ALU Area	10421 um^2	layout/analysis/DLX.gateCount
ALU Area	4968 um^2	layout/analysis/DLX.gateCount

Table 15.1: Post-Route analysis

APPENDIX A

Simulations

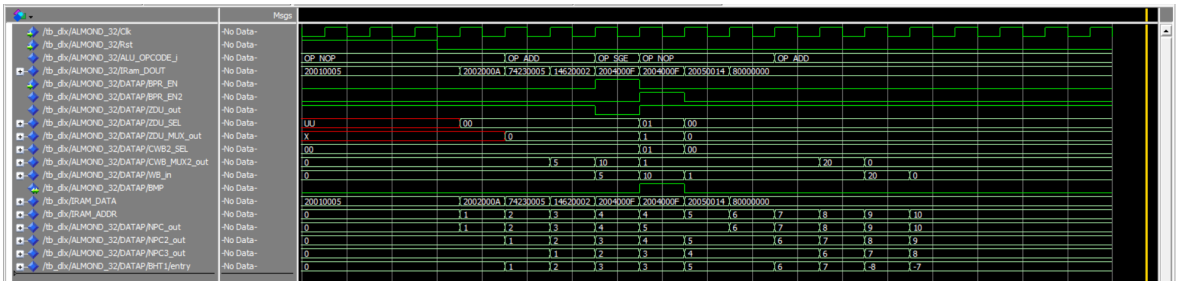

Figure A.1: Branch if not equal BNEZ command

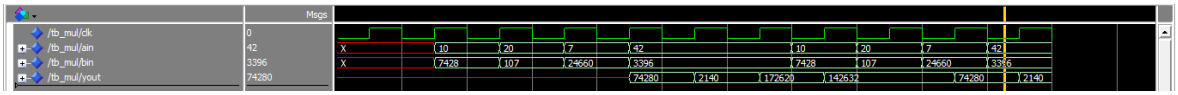

Figure A.2: Pipelined multiplier

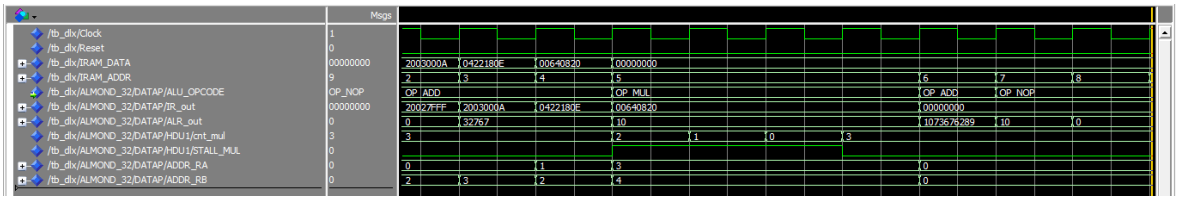

Figure A.3: Multiplication performed by the Almond-32, with a 4 clock cycle latency

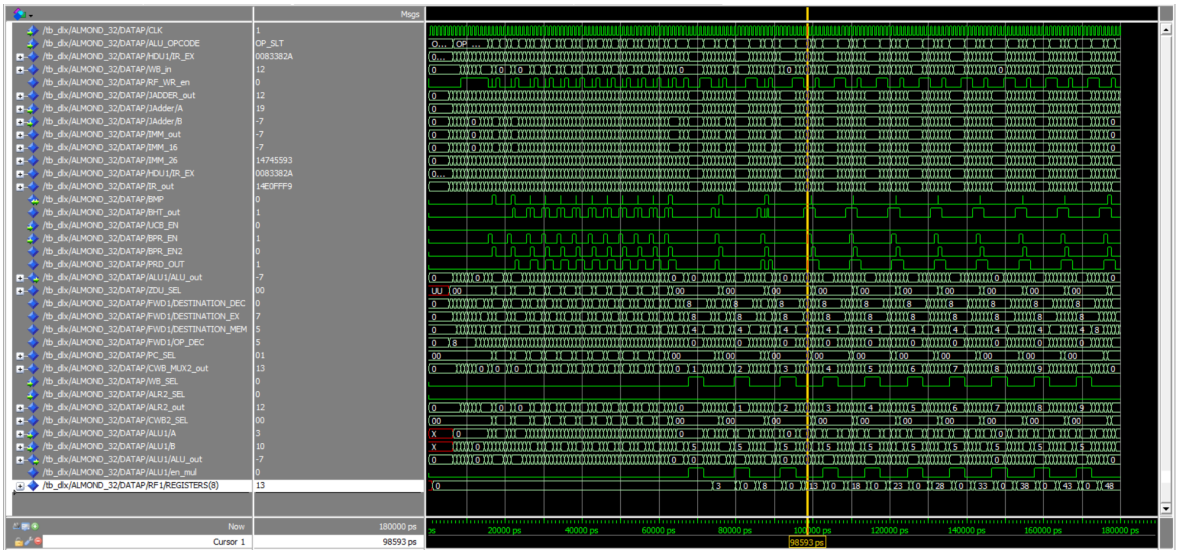

Figure A.4: Code simulation

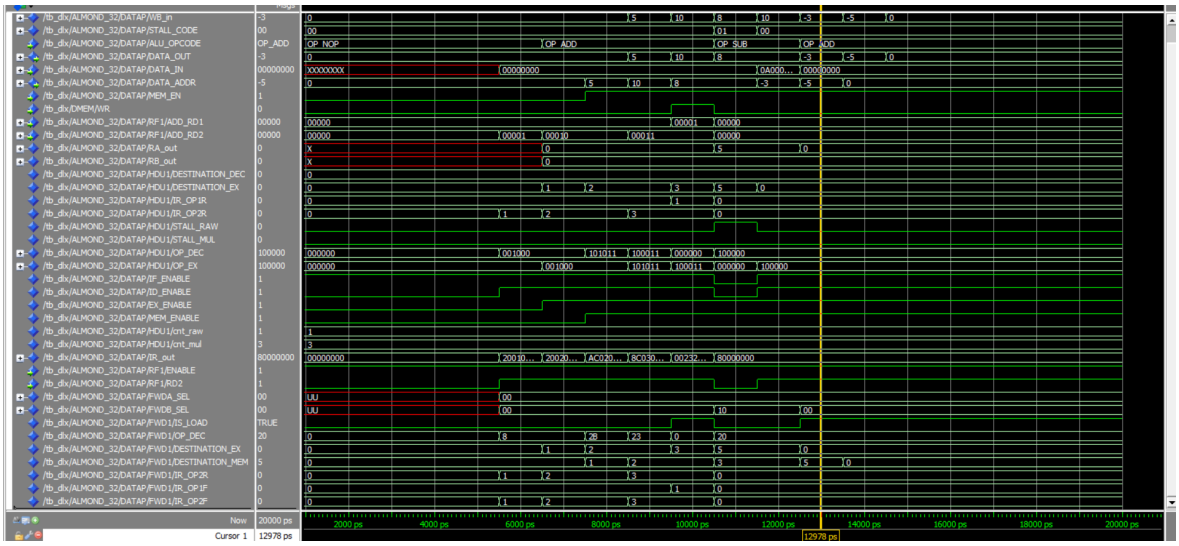

Figure A.5: Stall

APPENDIX B

System block diagram

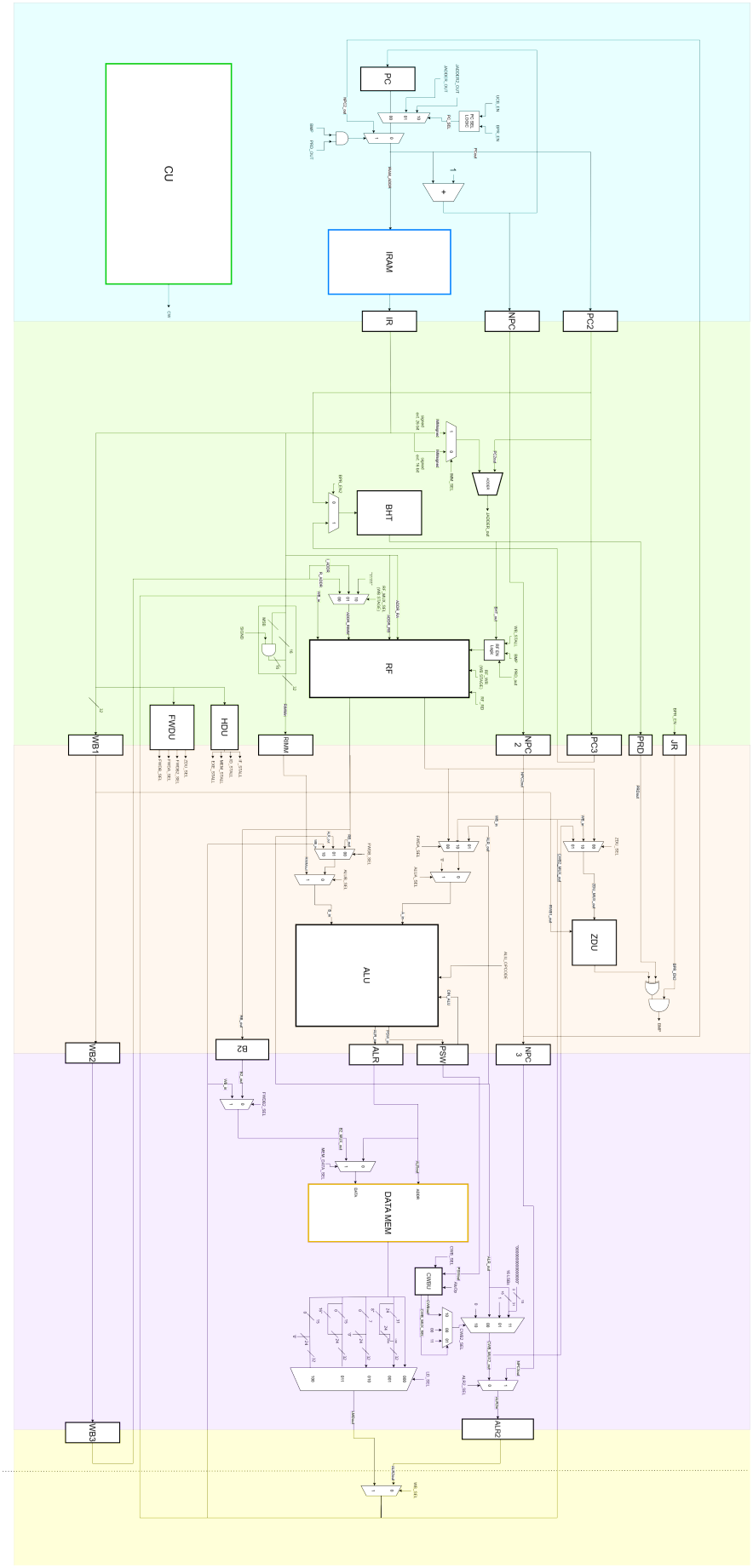

Figure B.1: Vertical view of Almond-32 diagram

Bibliography

[1] Booth Algorithm. https://www.semanticscholar.org/paper/design-of-radix-4-booth-multiplier-using-mgdi-and-p-verma-manchanda/e0e6ac616120244fbfbd3933e5756f1b8182d55f.

[2] Documentation Arm Developer. https://developer.arm.com/documentation/ddi0236/h/i1044275.

[3] David A. Patterson John. L. Hennessey. Computer architecture - a quantitative approach. *Elsevier Inc.*, 2012.

[4] Intel Pentium 4 Processor Optimization. *Copyright 1999-2001 Intel Corporation.*