



LIÈGE
université

INFO0012-2 - Computation structures

Lecture notes

2018-2019

DEFLANDRE Guilian
Applied sciences faculty
Liège university

1 Introduction.

Generally speaking, computers can be classified into three generations. Each of these generation lasted for a certain period of time.

First generation: 1937-1946 - In 1937 the first electronic digital computer was built by Dr. John V. Atanasoff and Clifford Berry for military applications. It was called the Atanasoff-Berry Computer (ABC). It is said that this computer weighed 30 tons, and had 18,000 vacuum tubes which was used for processing. Computers of this generation could only perform single task, and they had no operating system.

Second generation: 1947-1962 - This generation of computers used transistors instead of vacuum tubes which were more reliable. In 1951 the first computer for commercial use was introduced to the public; the Universal Automatic Computer (UNIVAC 1). In 1953 the International Business Machine (IBM) 650 and 700 series computers made their mark in the computer world. During this generation of computers over 100 computer programming languages were developed, these computers already used memory and operating systems. Storage media such as tape and disk were in use also were printers for output.

Third generation: 1963-Today - The invention of integrated circuit brought us the third generation of computers. With this invention computers became smaller, more powerful more reliable and they are able to run many different programs at the same time. In 1980 Microsoft Disk Operating System (MS-Dos) was born and in 1981 IBM introduced the personal computer (PC) for home and office use. Three years later Apple gave us the Macintosh computer with its icon driven interface and the 90s gave us Windows operating system.

As we can see, computers already have been improved a lot in a short period of time, this courses will give us bases to understanding how a computer work.

From <http://people.bu.edu/baws/brief%20computer%20history.html>

1.1 General components of a modern computer.

1.1.1 The motherboard.

The motherboard is the main printed circuit board (PCB) found in general purpose computers and other expandable systems. It holds, and allows, communication between many of the crucial electronic components of a system, such as the central processing unit (CPU) and memory, and provides connectors for other peripherals. On the figure 1.1 we can see in red the processor (here an

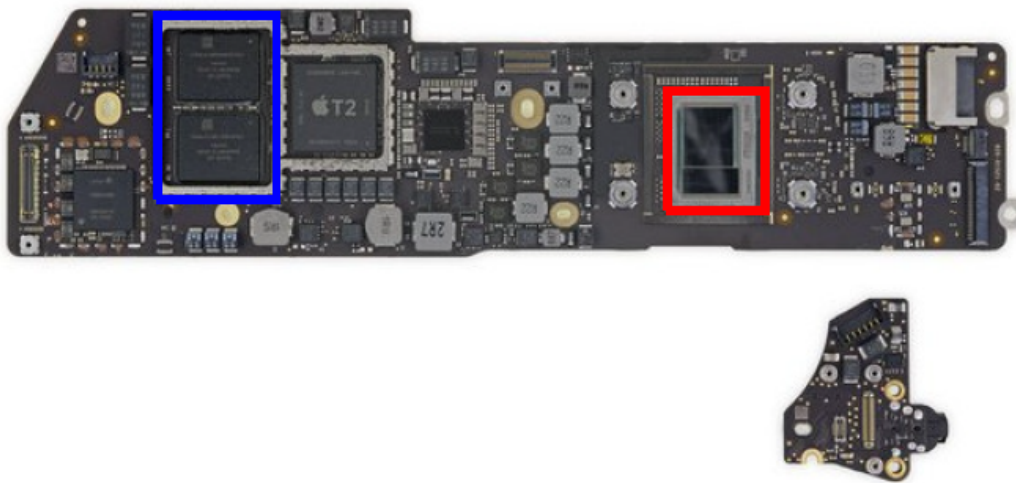


Figure 1.1: Motherboard of a Macbook Air 13" Retina 2018

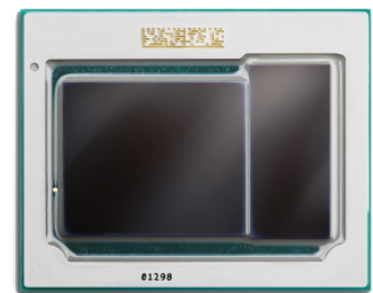


Figure 1.2: Processor Intel Core i5-8210Y (2018).

Intel Core i5-8210Y) and in blue the solid-state drive (SSD) of 128GB which is a device that uses integrated circuit assemblies as memory to store data persistently.

1.1.2 The processor.

The processor (or central processing unit, CPU) is the electronic circuitry within a computer that carries out computer program's instructions by performing the basic arithmetic, logic, controlling, and input/output (I/O) operations specified by instructions. Principal components of a CPU include the arithmetic logic unit (ALU) that performs arithmetic and logic operations, processor registers that supply operands to the ALU and store the results of ALU operations, and a control unit that

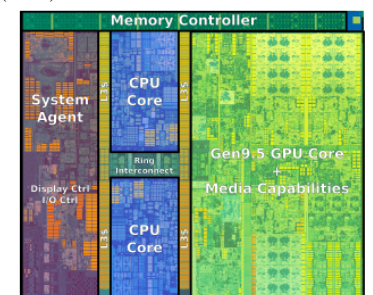


Figure 1.3: Inside view with different parts of the Intel Core i5-8210Y (2018).

orchestrates the fetching (from memory) and execution of instructions by directing the coordinated operations of the ALU, registers and other components.

SUMMARY

- The **motherboard** of a computer is its main circuit linking namely the **CPU** and the **memory**.
- The **central unit processing**, CPU or **processor** is probably the most important part of a **computer**. It is composed of an **ALU**, a **control logic**, **processor's registers** and other components.
- The **memory** is a device that is used to store **information** for **immediate** use in a computer or related computer hardware device.

2 Logic gates.

In order to design our computer we need to define a kind of base unit, as we do for instance in mathematics with variables. Here we will use the digital abstraction, which means that we treat only digital data. Digital circuits are usually implemented by electronic devices and operate in the real world. These devices are so called logical gates and in this section we will present the most famous of them, in which each circuit can be translated, associated to their truth table¹.

¹ A truth table is a mathematical table used in logic which sets out the functional values of logical expressions on each of their functional arguments.

2.1 Identity

A	B
0	0
1	1

2.2 Negation.

A	B
0	1
1	0

2.3 OR.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

2.4 AND.

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

2.5 NOR.

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

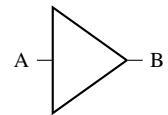


Figure 2.1: Buffer logical gate.

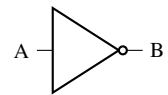


Figure 2.2: Negation (NOT) logical gate.

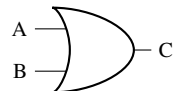


Figure 2.3: Disjunction (OR) logical gate.

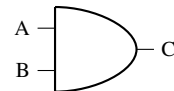


Figure 2.4: Conjunction (AND) logical gate.

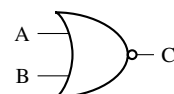


Figure 2.5: Not OR (NOR) logical gate.

2.6 NAND.

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

2.7 XOR.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

2.8 From silicon to logic gates.

Physically, Boolean information are represented by electrical signals, that is, a voltage. On figure 2.8 we can see for instance that $[0; 1]$ V represent a 0 and $[4; 5]$ V represent a 1.

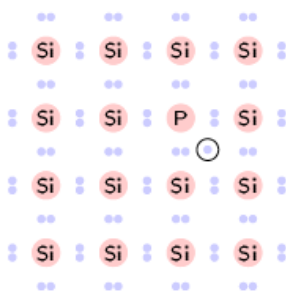
It's well known that the practice doesn't work as well as the theory pretend, so we added *margins* at our Boolean representation interval in order to minimize errors introduced by the noise² in a channel. Thenceforth our intervals are now $[0; 1, 5]$ V = 0, $[3, 5; 5]$ V = 1 and every signals between them are forbidden.

2.8.1 Semiconductors.

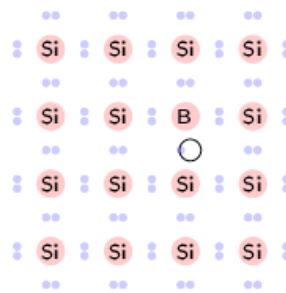
Basically, a semiconductor is a device mostly composed of silicon (Si). Occasionally you can find semiconductors made up of germanium (Ge), Gallium arsenide (GaAs), or from other chemical composition.

On figure 2.9 you can observe the structure of a silicon crystal. You can easily "doping" a crystal of this type by introducing impurities inside it. These impurities are from 2 kind

- **n-type** - phosphorus (P) or arsenic (As) (5 electrons in outermost shell): create an excess of electrons, which means that an electron is loosely bounded. Charge carriers are electrons.
- **p-type** - boron (B) (3 electrons in outermost shell): create a deficit of electrons, which means that an electron is missing. Charge carriers are holes.



(a) n-type impurities.



(b) p-type impurities

2.8.2 Diodes.

Using the impurities described above, we can create what is so called a diode. A diode is an electronic component that let the current pass in only one direction.

The figure 2.11 show how a diode is created. In this diagram we can see the depletion zone, which is at the junction. P-type semiconductor captures electrons from n-type semiconductor. There is a non-conducting zone (with no carriers). We have than 2 observable cases

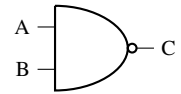


Figure 2.6: Not AND (NAND) logical gate.

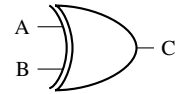


Figure 2.7: Exclusif OR (XOR) logical gate.

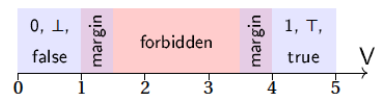


Figure 2.8: Boolean representation by voltage.

² In signal processing, noise is a general term for unwanted (and, in general, unknown) modifications that a signal may suffer during capture, storage, transmission, processing, or conversion



Figure 2.9: Crystal of Silicon.

Figure 2.10: Illustration of the different types of impurities in a Si crystal.



Figure 2.11: P-n junction who compose a diode.



Figure 2.12: Diode schematic representation.

- If one applies a positive voltage at the anode (p -type) with respect to the voltage at the cathode (n -type), loosely bound electrons in the n region will migrate towards the junction, and loosely bound "holes" in the p region will also migrate towards the junction. At the junction, holes and electrons recombine. Holes/electrons continue appearing on anode and cathode respectively, the current then flows.
- If one applies a negative voltage at the anode (p -type) with respect to the voltage at the cathode (n -type), loosely bound electrons in the n region will now to the cathode, and holes in the p region to the anode. The depletion zone (without carriers) gets larger and no current flows.

2.8.3 Metal-oxide-semiconductor field-effect transistor.

The metal–oxide–semiconductor field-effect transistor (MOSFET), also known as the metal–oxide–silicon transistor (MOS transistor, or MOS), is a type of field-effect transistor that is fabricated by the controlled oxidation of a semiconductor, typically silicon. It's composed of a *gate*, a *drain* and a *source*. The gate is insulated from the rest (no current in static mode) used SiO_2 or more recently SiO_xN_y as insulator. Source and drain (quite symmetric), supply and drain the FET. Drain and source are highly doped (p or n for p -FET or n -FET resp.) while the body (or channel) is lightly doped (n or p resp.)

2.8.3.1 N-FET.

Let V_{th} be the threshold and V_{GS} be the voltage between the gate and the source, as shown in figure 2.13.

- If $V_{GS} < V_{th}$: there is no conduction between drain and source;
- If $V_{GS} > V_{th}$: the gate is positive and will attract electrons. At some point, there will be a channel between drain and source with negative charge carriers, as in drain and source. If $V_{DS} > 0$, electrons will flow from source to drain;
- If $V_G = V_1$: then $V_D = V_S$;
- If $V_G = V_0$: then V_D can be V_0 or V_1 , whatever V_S is;

2.8.3.2 P-FET.

- If $V_{GS} < V_{th}$: the gate is negative, and will attract holes. At some point, there will be a channel between drain and source with positive charge carriers, as in drain and source. If $V_{DS} < 0$, holes will flow from source to drain;
- If $V_{GS} > V_{th}$: there is no conduction between drain and source;
- If $V_G = V_1$: then V_D can be V_0 or V_1 , whatever V_S is;
- If $V_G = V_0$: then $V_D = V_S$;

2.8.3.3 CMOS: Complementary metal-oxide-semiconductor.

The complementary metal–oxide–semiconductor is a type of MOSFET fabrication process that uses complementary and symmetrical pairs of p -type and n -type MOSFETs to implement logic functions. This technology is energy efficient since we have low static power consumption (very low current when not changing state). The figure 2.15 show how this technology is used in order to implement the NOT logic gate (see subsection 2.2).

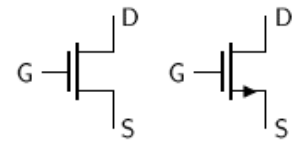
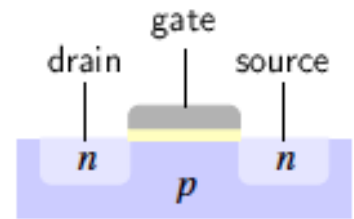


Figure 2.13: N-field effect transistor.

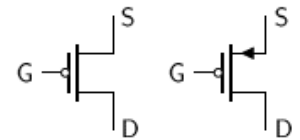
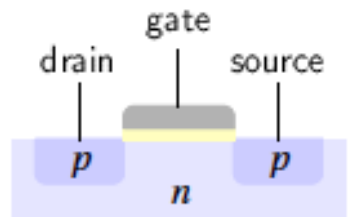


Figure 2.14: P-field effect transistor.

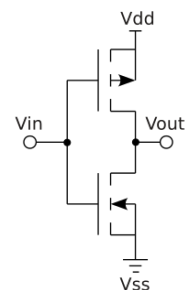


Figure 2.15: Example of a CMOS technology, the CMOS inverter (NOT logic gate).

SUMMARY

- The **digital abstraction** serves as basics tool for **digital circuits** implementation.
- **Universal logic gates** can be used to produce any other logic or boolean function with the **NAND** and **NOR** gates being minimal.
- **Boolean information** is represented in practice by a **voltage**. Since **noise** can appear in reality, **security margins** are used in the representation.
- A **semiconductor material** has an electrical conductivity value falling between that of a conductor and an insulator. **Impurities** change the **conductivity characteristics** of the semiconductor. The **n-type** ones create an **excess** of electrons (**negative charge**) while the **p-type** ones create a **deficit** of electrons (**positive charge**).
- A **diode** is an electrical component exploiting **semiconductors impurities** in order to only let the **current** pass in a

single direction.

- A **metal–oxide–semiconductor field-effect transistor** (MOSFET), is a transistor made of a **gate**, a **drain** and a **source**. The gate is **insulated** from the rest, source and drain (quite symmetric), supply and drain the FET.

3 Combinational circuits.

A combinational circuit is a circuit composed of several logic gates. The characteristics of this kind of circuits are the following.

- Any gate's input is connected to at most one output. Note that a single output might be connected to several inputs;
- In combinational circuits, cycles are outlaws;
- For x_1, \dots, x_n the inputs and y_1, \dots, y_m the outputs. Each y_i is a Boolean function over the inputs;
- Some delays are necessary so that the right output is computed, when inputs are modified;
- Any binary function can be implemented by a digital circuit;

In this section we will describe some famous combinational circuits who are basics components of computers, even the most recent one.

3.1 Adder.

The binary addition remains a crucial operation even if it's very simple one. Adders are hardware implementation of this operation, there are responsible of several main operation in computers, namely loops, program counters,... The conception of this device for x bits (full adder) is a pool of several adders of 1 bits each (half-adder).

3.1.1 Half-adder.

The purpose of the half-adder is to implement the addition of 2 bits A and B. Let A and B the inputs, and S and C the outputs of the circuit with S the sum of A and B and C the carry bit of this addition. We can design the circuit computing the 2-binary addition (3.1) using the binary function represented in the table 3.1.

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 3.1: Truth table of the half adder.

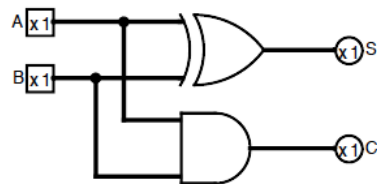


Figure 3.1: Combinational circuit of an half-adder.

A	B	C	S	C'
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3.2: Truth table of the full adder.

3.1.2 Full adder.

In order to perform the full binary addition of 2 bits, we need to add an input to our implementation. For the outputs, as far as they are concerned, 2 are still sufficient. Based on the table 3.2, we can easily design the full adder which use the half-adder design.

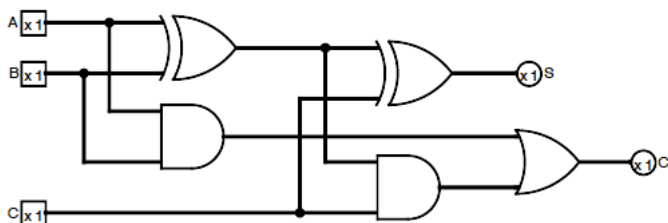


Figure 3.2: Combinational circuit of an full adder.

3.1.3 8-bits adder.

The 8-bits adder is simply composed of 8 half-adder boxes, figure 3.3 show the construction of such a circuit. From this one, we can easily deduce the shape of adders for any number of bits.

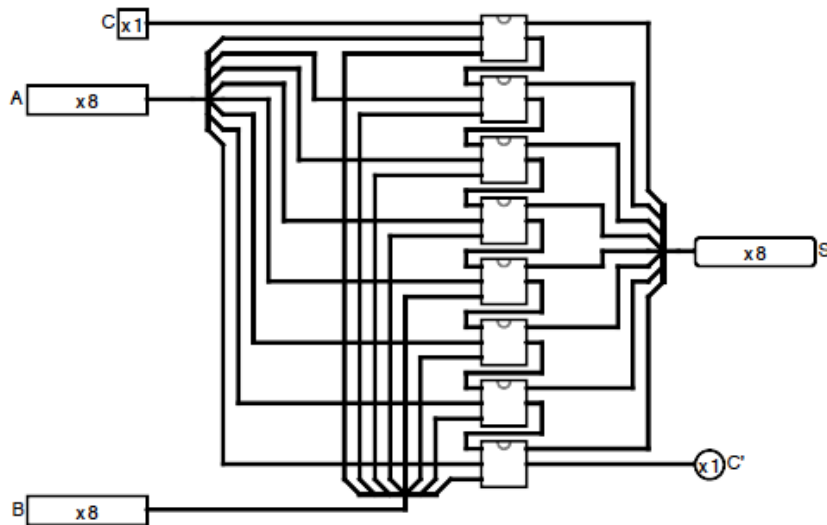


Figure 3.3: Combinational circuit of an 8-bits adder.

D0	D1	S0	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 3.3: Truth table of the multiplexer.

3.2 Multiplexer.

A multiplexer is a device that selects between several analog or digital input signals and forwards it to a single output line. It own two kind of input:

- **Sources:** the differents input signal that the multiplexer can select as output;
- **Selection bits:** which determine which source is routed to the output;

The simplest multiplexer is composed of two binary inputs and a selection bit. From its definition, we can deduce the truth table of a 2-to-1 multiplexer (see table 3.3). We represent in electronic the multiplexer as shown in figure 3.4. From its truth table, we can again easily construct the combinational circuit of the multiplexer, represented at figure 3.5.

From this definition, it is trivial that the multiplexer may have a higher number of sources. It will be enough simply to increase the number of selection bits. For instance, for a 3-bits selection multiplexer, we can have until $2^3 = 8$ sources, as represented in the figure 3.6.

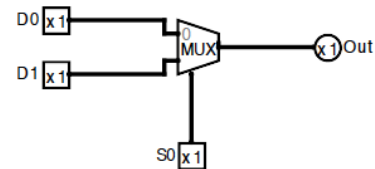


Figure 3.4: 2-to-1 multiplexer.

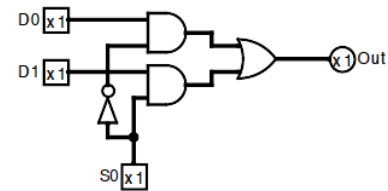
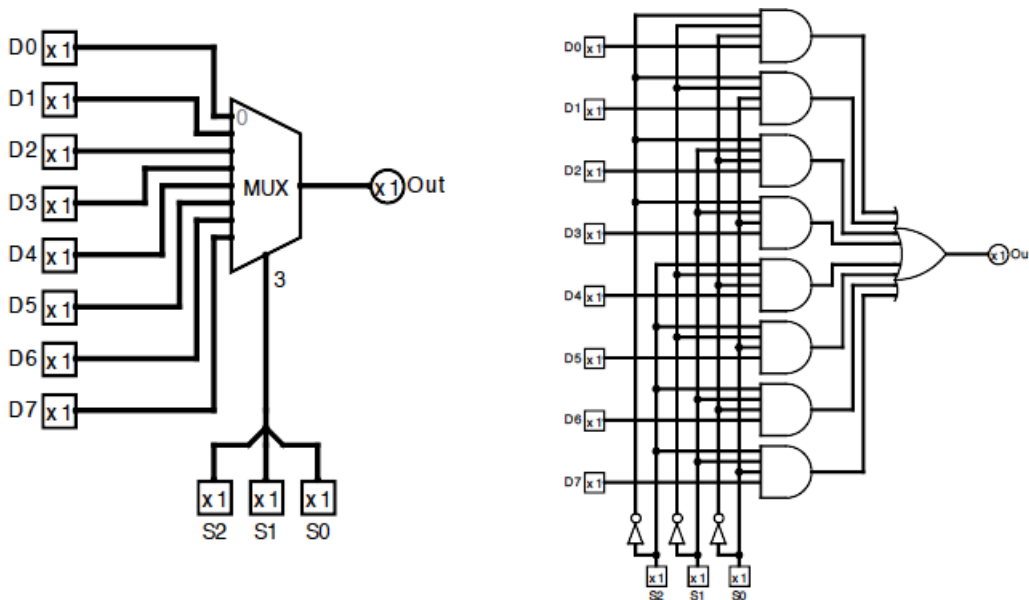


Figure 3.5: Combinational circuit of the multiplexer.

Figure 3.6: 8-to-1 multiplexer and its combinational circuit.



3.3 Decoder.

Basically, the decoder is the reverse circuit of the multiplexer. It take as input n selection bits and 1 supplementary bit then have 2^n outputs. If the input is 0, all outputs are 0. Otherwise, it behaves like the decoder. A decoder for 3 selection bits is sketch on figure 3.7.

3.4 Read-Only Memories (ROM).

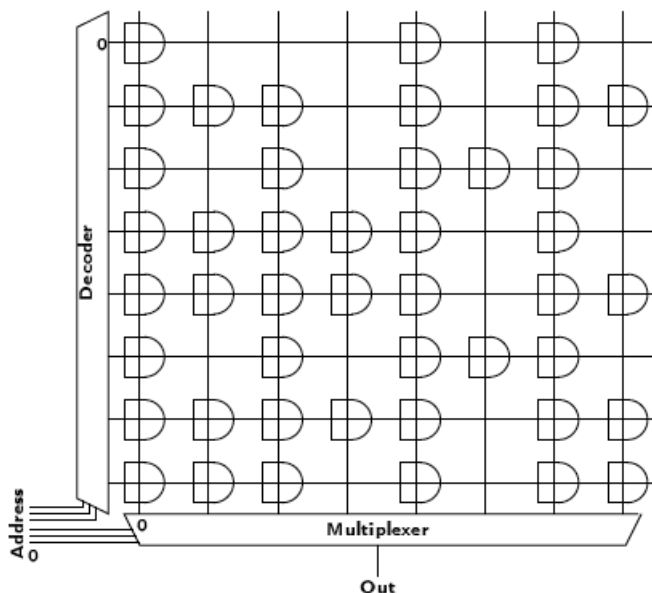
Read-only memory (ROM) is a type of non-volatile memory where data cannot be electronically modified after the manufacture of the memory device. There are several types of ROM

- **Mask-programmed ROMs:** contents set at conception-time.
- **Field-programmable ROMs (PROMs):** contents set by user, at initialization time (once and for all).
- **Erasable PROMs (EPROMs):** contents set by user, at initialization time (but can be reinitialized several time, *e.g.* flashable firmware)

ROM are composed of a special circuit element, the pulldown (see figure 3.8). In this component

- If the horizontal line is 1, then the vertical line is 0;
- Otherwise, the it's inactive;

If all pulldowns on a vertical line are inactive, the line is 1. With this new component, a decoder and a multiplexer, we can now build a ROM of random size. As an example, we drew a ROM of 6-bits addresses and 1-bit data in figure 3.9 (also noted 64x1 ROM).



3.5 Arithmetic and Logic Unit (ALU).

The ALU is a combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers. It is a fundamental building block of many types of computing circuits, namely the central processing unit (CPU) of computers. Typically, a modern ALU should perform the following operations

- Addition;
- Subtraction;
- Multiplication;
- Division;
- AND logical;
- OR logical;
- XOR logical;
- Comparison;

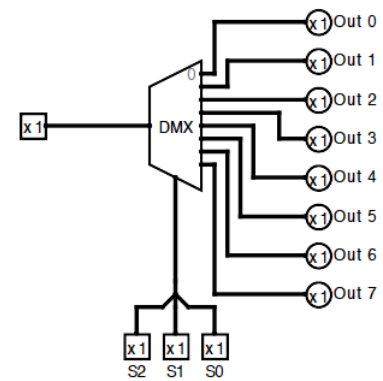


Figure 3.7: 1-to-8 demultiplexer.

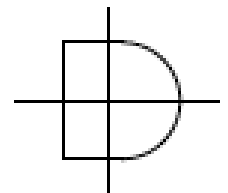


Figure 3.8: Pulldown logical gate.

Figure 3.9: Example of a ROM combinational circuit.

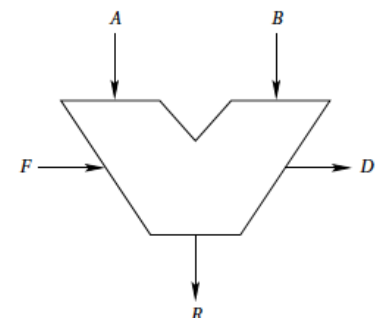


Figure 3.10: Arithmetic and Logic Unit representation.

- Shift;

An ALU have as input

- **A and B**: the arguments of the performed operation;
- **F**: that specifies the operation and some additional arguments, namely the number of shifted bits and the code of the performed operation.

and have R (the result) and D (additional) as outputs. If you have access of all components that perform the ALU operations, you can build it for 32-bits operation as represented in figure 3.11.

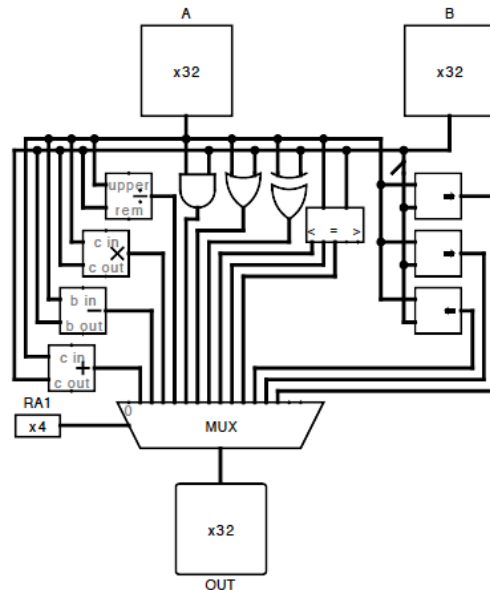


Figure 3.11: ALU combinational circuit.

SUMMARY

- In **digital circuit** theory, **combinational logic** is a type of digital logic implemented by **Boolean circuits**, where the **output(s)** is a **pure function** of the present **input(s)** only.
- **Adders** implements **binary addition** operation for **2 or 3-bits inputs**. These n **half-adder** can be placed in **parallel** in order to implement the addition of n -bits inputs.
- A **multiplexer** permits to choose a **signal** as output among n **inputs** signals.
- A **decoder** permits to chose among n **outputs** the one to forward **the single** input.
- **Read-only memory** are **non-volatile** memory resulting from the assembly of a **decoder**, a **multiplexer** and **pull-down** logical gates.
- An **arithmetic and logic unit (ALU)** is a king of **binary calculator** giving as output basic **arithmetic** and **logic** operations performed on its **2 inputs**.

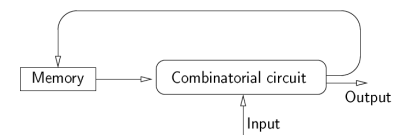


Figure 4.1: Schematic representation of a basic sequential circuit.

4 Sequential circuits, states.

A sequential logic is a type of logic circuit whose output depends not only on the present value of its input signals but also on the sequence of past inputs, the input history as well. In combinational logic only the present input was consider but now we have a notion of memory for the past output that is required. The figure 4.1 represent the backbone of a sequential circuit where the memory is about the past states of the circuit.

4.1 Basic memory element.

4.1.1 Basic R/S latch (flip-flop).

The R/S latch is a combinational circuit with two inputs R (Reset) and S (Set) and an output Q whose value is 1 if S was 1 more recently than R. It can be constructed from a pair of cross-coupled NOR logic gates as represented at figure 4.2. The truth table associate with each respective action from a given state is given in the table

While the R and S inputs are both low, feedback maintains the Q output in a constant state. If S (Set) is pulsed high while R (Reset) is held low, then the Q output is forced high, and stays high when

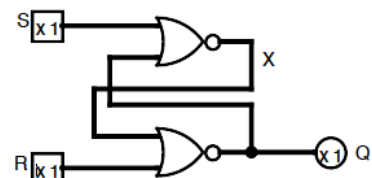


Figure 4.2: Combinational circuit of a R/S latch.

Characteristic table				Excitation table			
S	R	Q_{next}	Action	Q	Q_{next}	S	R
0	0	Q	Hold state	0	0	0	X
0	1	0	Reset	0	1	1	0
1	0	1	Set	1	0	0	1
1	1	X	Not allowed	1	1	X	0

S returns to low. Similarly, if R is pulsed high while S is held low, then the Q output is forced low, and stays low when R returns to low.

4.1.2 Clocked R/S latch.

A simple clocked SR flip-flop built from AND-gates in front of a basic SR flip-flop with NOR-gates as it can be see on figure 4.3. This circuit has two inputs (D and G) and one output (Q). When G is high, it memorizes D and keeps the memory as long as G stay low. Q is then the memorized bit.

4.1.3 Registers.

In digital electronics, especially computing, registers are circuits typically composed of flip-flops, often with many characteristics similar to memory, such as:

- The ability to read or write multiple bits at a time;
- The ability of using an address to select a particular register in a manner similar to a memory address;

Hardware registers are used in the interface between software and peripherals. Software writes them to send information to the device, and reads them to get information from the devices. Some hardware devices also include registers that are not visible to software, for their internal use.

Figure 4.4 sketch a register for one bit made of 2 boxes who are a clocked R/S latch and a reverser. In practice we represent such a register as shown in figure 4.5. This circuit work as follow

- While G is 0, the first latch is open. The memorized bit is in the second latch;
- When G becomes 1, the first latch is closed and D is memorized, and the second opens. The memorized bit (in the first latch) traverses the second latch;
- When G becomes 0, the second latch closes and memorizes the bit that was in the first latch.

The output does not change, but the first latch opens;

We can use this circuit in order to design registers able to store larger data. Let's draw a 8-bit register, its draft is given in figure 4.6. Its practical representation is given in figure 4.7. This means that a register is just a set of one bit memories with the same clock. Note that modern computers work with 32-bits and even 64-bits registers.

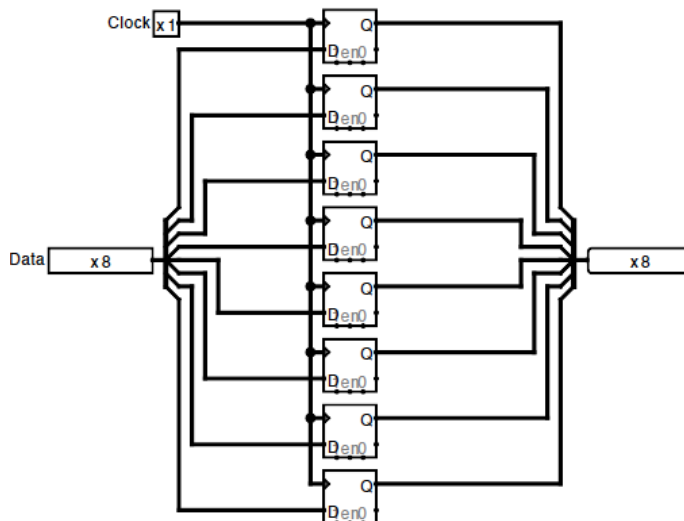


Table 4.1: Truth table of a R/S latch.

REMARK

- There are constraints on components
- Some small computing delays are required (fine in practice).
 - If the inputs of a NOR gate change, but the result is preserved, the output should be stable (not even a temporary change/glitch).
 - R and S are stable sufficiently long so that the latch stabilizes.

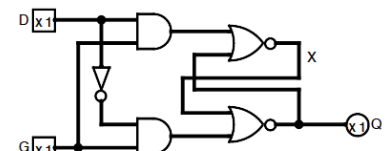


Figure 4.3: Clocked R/S latch combinational circuit.

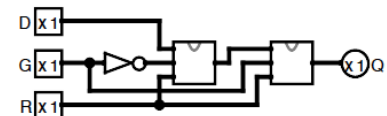


Figure 4.4: One bit register.

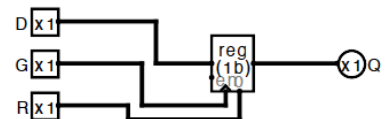


Figure 4.5: Representation of a one bit register in practice.

Figure 4.6: 8-bits register circuit.

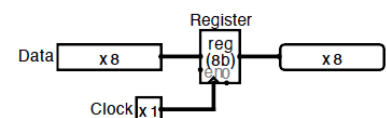


Figure 4.7: Representation of a 8-bits register in practice.

4.2 Synchronous circuits.

A synchronous circuit is a digital sequential circuit in which the changes in the state of memory elements are synchronized by a single clock signal. It's composed of a bunch of combinational acyclic circuits linked by registers. It is mandatory that the clock period (inverse of frequency) be larger than the maximum computation time of the combinational circuits, otherwise transiting data would not be the expected ones. In this course we adopt the rise-edge policy, which means that each time the clock changes from 0 to 1, the input of registers is memorized, then transmitted to the output. Several example of such circuits can be design, you can refer to the practical work 1 session for more details.

4.3 Finite state machine.

A Finite State Machine (FSM) is a model of computation based on a hypothetical machine made of one or more states. Only one single state of this machine can be active at the same time. It means the machine is able to transit from one state to another in order to perform different actions. A FSM is any device storing the state of something at a given time. The state will change based on inputs, providing that the resulting output for the implemented one changes. Formal definitions of FSMs are provided in definitions 4.1-4.2.

There are two types of finite state machines

- **Deterministic finite state machines:** accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string. Deterministic refers to the uniqueness of the computation run.
- **Non-deterministic finite state machines:** each of its transitions is uniquely determined by its source state and input symbol, and reading an input symbol is required for each state transition.

Definition 4.1 - Formal definition of a deterministic finite state machine.

A deterministic finite automaton (DFA) is described by a five-element tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q : a finite set of state.
- Σ : a finite, nonempty input alphabet.
- δ : a series of transition functions.
- q_0 : a series of transition functions.
- F : the set of accepting states.

There must be exactly one transition function for every input symbol in Σ from each state.

Definition 4.2 - Formal definition of a deterministic finite state machine.

A nondeterministic finite automaton (DFA) is described by a five-element tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q : a finite set of state.
- Σ : a finite, nonempty input alphabet.
- δ : a series of transition functions.
- q_0 : a series of transition functions.
- F : the set of accepting states.

Unlike DFAs, NDFAs are not required to have transition functions for every symbol in Σ , and there can be multiple transition functions in the same state for the same symbol. Additionally, NDFAs can use null transitions, which are indicated by ϵ . Null transitions allow the machine to jump from one state to another without having to read a symbol.

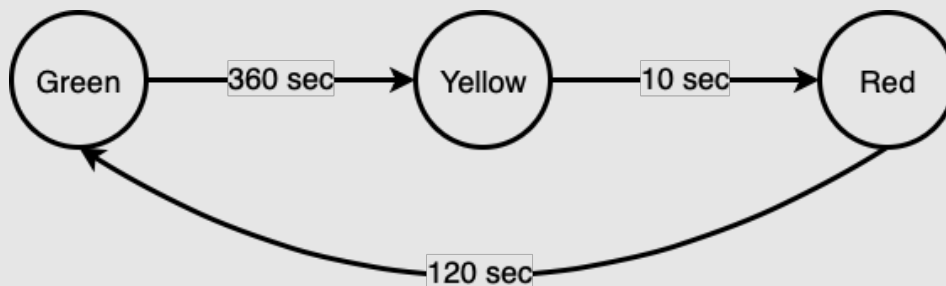
Example 4.1: Construction of the finite state machine for a traffic light.

A simple traffic light system can be modeled with a Finite State Machine. Let's look at each core component and identify what it would be for traffics lights

- **States:** a traffic light generally has three states: Red, Green and Yellow.

- **Initial State:** Green
- **Accepting States:** in the real world traffic lights run indefinitely, so there would be no accepting states for this model, as long as lights are red, yellow or green.
- **Alphabet:** positive integers representing seconds.
- **Transitions:**
 - If we're in state Green and wait for 360 seconds (6 minutes), then we can go to state Yellow.
 - If we're in state Yellow and wait 10 seconds, then we can go to state Red.
 - If we're in state Red and wait 120 seconds, then we can go to state Green.

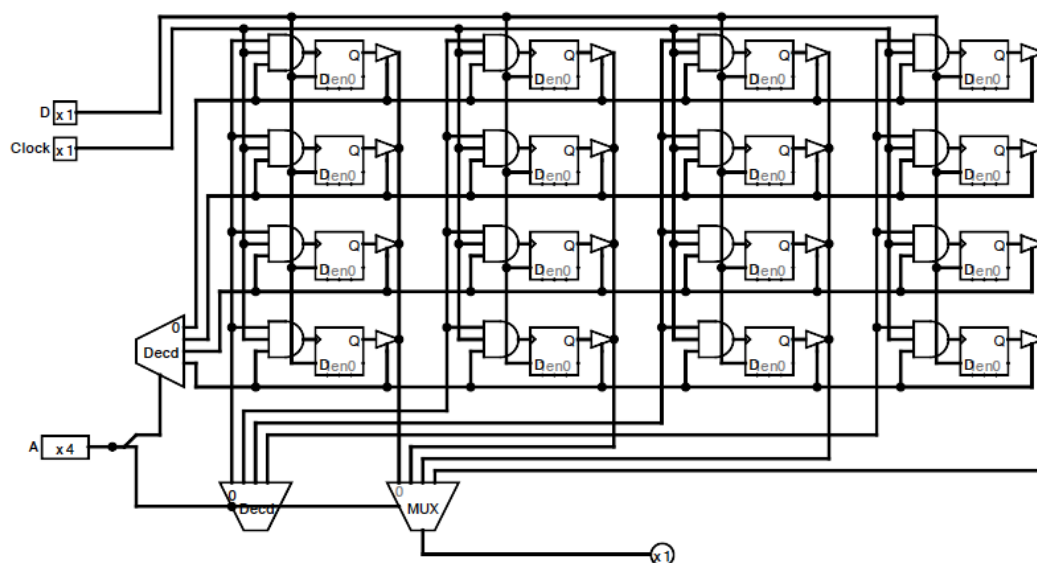
With this we can easily draw this traffic light finite state machine



4.4 Random-access memory (RAM).

The random-access memory (RAM) is a form of computer memory that can be read and changed in any order, typically used to store working data and machine code. A random-access memory device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory.

RAM are similarly to ROM except that they use one bit registers (SRAM). This type of memory contain multiplexing and demultiplexing circuitry, to connect the data lines to the addressed storage for reading or writing the entry. Usually more than one bit of storage is accessed by the same address, and RAM devices often have multiple data lines and are said to be "8-bit" or "16-bit", etc. devices. Figure 4.8 show as instance a RAM module of 4 bits addresses and one bit data.



REMARK

In contrast, with other direct-access data storage media such as hard disks, CD-RWs, DVD-RWs and the older magnetic tapes and drum memory, the time required to read and write data items varies significantly depending on their physical locations on the recording medium, due to mechanical limitations such as media rotation speeds and arm movement.

Figure 4.8: Example of a RAM module.

SUMMARY

- In **digital circuit** theory, **sequential logic** is a type of digital logic implemented by **Boolean circuits**, where the **output(s)** is a **function** of the present **input(s)** but also of the **previous** ones.
- **R/S latches**, **clocked RS/latch** and **registers** are basic **memory components** on which the whole **sequential logic** is based.
- A circuit is said **synchronous** when **all** its **memory elements** are synchronized to the **same clock**. Otherwise it will be said **asynchronous**.
- A **finite state machine** is a model that permit to **easily** notice all possible **machine's states** and their possible **transitions**.
- A **random-access memory** (RAM) is a **memory component** in which **data** can be **read** but also **modified** at any time. It is made of 2 **decoders**, a **multiplexer** and **RAM cells**.

5 Instruction set architecture (ISA) for the β -machine

5.1 Von Neumann architecture.

The Von Neumann architecture is one of the oldest and simplest computer architecture, based on a 1945 description by the mathematician and physicist John Von Neumann. It's nothing more than a special case of a sequential circuit with

1. A processing unit that contains an arithmetic logic unit and processor registers;
2. A control unit that contains instruction registers and program counter;
3. A unique memory that stores data and instructions for both program instructions and data;
4. External mass storage;
5. Input and output mechanisms;

5.2 Instruction Set Architecture.

An instruction set architecture (ISA) is an abstract model of a computer and its realization is called an implementation. Software that has been written for an ISA can run on different implementations of the same ISA, which offers compatibility between different generations of computers to be easily achieved, and the development of computer families.

An ISA defines everything a machine language programmer needs to know in order to program a computer. What an ISA defines differs from one to another. In general, ISAs define

- The supported data types;
- What state there is (such as the main memory and registers) and their semantics (such as the memory consistency and addressing modes);
- The instruction set (the set of machine instructions that comprises a computer's machine language)
- The input/output model;

There are two main family of ISA which are the following

- **Reduced instruction set computer (RISC):** a computer that allows to have fewer cycles per instruction (CPI), the general concept is that such a computer has a small set of simple and general instructions, rather than a large set of complex and specialized instructions.
- **Complex instruction set computer (CISC):** a computer in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.

5.3 Backbone of our instruction set architecture.

We want to design a computer that is simple, so we will prefer the RISC family to the CISC. We must also decide on the size of instruction. We could choose to have variable length but it would make everything more complicated. With constant length, the next instruction address is trivial (except for jumps). What is then the good power word length? 8 bits, 16, 32, 64,..., 32? The smallest number to

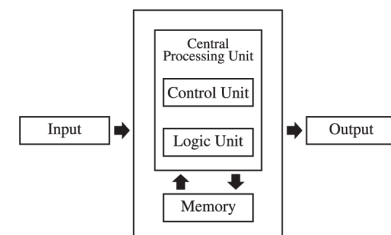


Figure 5.1: Von Neumann computer model.

get enough space for a complete instruction. For the registers size, let's again keep things simple by choosing 32 registers of 32-bits each. With this we complete our discussion in order to have a draft of our β -machine which will be

- A simple, pedagogical RISC computer;
- A 32-bits architecture, which means that
 - All registers are on 32-bits (4 bytes);
 - The memory address space is on 32-bits;
- With each instruction is on 32-bits (4 bytes);
- Having 32 registers (from 0 to 31) (register 31 (R31) is hardwired to 0). $\text{Reg}[x]$ denotes the content of register x ;
- Memory addresses on 32-bits. $\text{Mem}[x]$ denotes the word (32-bits) at address x . The two least significant bits of x are ignored (0);
- Variables are stored in memory. Registers hold temporary values. Computation occurs on registers. Memory operations to store to and fetch from memory;
- The program counter PC is on 32-bits, and can only be a multiple of 4 (the two least significant bits are 0);

5.4 The instruction set.

5.4.1 Instruction formats.

In the β -machine we have 32-bits to code an instruction. An instruction can use only registers or registers and constants depending of its type. In these 32-bits instructions we found the the OPCODE on the 6 first bits who will give the operation to perform (so there is $2^6 = 64$ possible operations). We then found several 5 bits arguments, each one specifying a register. After what the resulting bits are unused. If the operation is perform using constant than it still has OPCODEs and 2 registers fields, the constant value is provided in the 16 last bits in the literal field and none of them remains unused. Format of these instructions types are provided in figure 5.2.

OPCODE	Rc	Ra	16 bits constant	
OPCODE	Rc	Ra	Rb	unused

Figure 5.2: Formats of the instructions of the β -machine.

Example 5.1: ADD and ADDC instructions.

The ADD instruction can be describe as

ADDC(R_a, R_b, R_c): $PC \leftarrow PC + 4$
 $\text{Reg}[R_c] \leftarrow \text{Reg}[R_a] + \text{Reg}[R_b]$

or, in assembly: `ADD(R1, R2, R3)`. The instruction in binary can be thus write as follow

OPCODE	Rc	Ra	Rb	UNUSED
100000	00011	00001	00000	0000000000

The ADDC instruction can be describe as

ADDC($R_a, \text{literal}, R_c$): $PC \leftarrow PC + 4$
 $\text{Reg}[R_c] \leftarrow \text{Reg}[R_a] + \text{SXT}[\text{literal}]$

or, in assembly: `ADDC(R1, -3, R3)`. The instruction in binary can be thus write as follow

OPCODE	Rc	Ra	Literal
110000	00011	00001	1111111111111101

5.4.2 Arithmetic instructions.

In this course we defined a set on instructions with 6-bits OPCODE, meaning that we can define $2^6 = 64$ different instructions. For the ALU we have 4 basic arithmetic instructions, the addition, the subtraction, the multiplication and the division. These operations work with registers only or with a 16 bits literal, we thus count 8 different functions classified in table 5.1.

ADD(R_a, R_b, R_c): $PC \leftarrow PC + 4$
 $\text{Reg}[R_c] \leftarrow \text{Reg}[R_a] + \text{Reg}[R_b]$

Without constant		With constant	
OPCODE	Name	OPCODE	Name
0x20	ADD	0x30	ADDC
0x21	SUB	0x31	SUBC
0x22	MUL	0x32	MULC
0x23	DIV	0x33	DIVC

Table 5.1: Arithmetic instructions' OPCODE and name definition on the β -architecture.

```

ADDC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                      Reg[Rc]  $\leftarrow$  Reg[Ra] + SXT[literal]
SUB(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra] - Reg[Rb]
SUBC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                      Reg[Rc]  $\leftarrow$  Reg[Ra] - SXT[literal]
MUL(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra] * Reg[Rb]
MULC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                      Reg[Rc]  $\leftarrow$  Reg[Ra] * SXT[literal]
DIV(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra] / Reg[Rb]
DIVC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                      Reg[Rc]  $\leftarrow$  Reg[Ra] / SXT[literal]

```

Listing 5.1: Arithmetic instructions definition.

5.4.3 Logic instructions.

Logical instructions as already introduced in section 2 can be limited to 3 functions here for our architecture. Note that it's always possible to compute every logical function out of these 3 logical basic operations. They are defined in table 5.2 for our β -machine ALU with their OPCODE and name. The executions of these instructions on the β -machine are detailed in the listing 5.2.

```

AND(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra] & Reg[Rb]
ANDC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                      Reg[Rc]  $\leftarrow$  Reg[Ra] & SXT[literal]
OR(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra] || Reg[Rb]
ORC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                      Reg[Rc]  $\leftarrow$  Reg[Ra] || SXT[literal]
XOR(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra]  $\oplus$  Reg[Rb]
XORC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                      Reg[Rc]  $\leftarrow$  Reg[Ra]  $\oplus$  SXT[literal]

```

Listing 5.2: Logic instructions definition.

Without constant		With constant	
OPCODE	Name	OPCODE	Name
0x28	AND	0x38	ANDC
0x29	OR	0x39	ORC
0x2A	XOR	0x3A	XORC

Table 5.2: Logic instructions' OPCODE and name definition on the β -architecture.

5.4.4 Comparison instructions.

Inside the β -machine's ALU we found 3 comparison instructions. You can easily compute their inverse operation by exchanging the register or the constant used for the comparison. As for the previous point we will find the definitions of these instructions in listing 5.3 and their OPCODE definitions in table 5.3.

```

CMPEQ(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                if Reg[Ra] = Reg[Rb] then Reg[Rc]  $\leftarrow$  1
                else Reg[Rc]  $\leftarrow$  0
CMPEQC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                      if Reg[Ra] = SXT[literal] then Reg[Rc]  $\leftarrow$  1
                      else Reg[Rc]  $\leftarrow$  0
CMPLT(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                if Reg[Ra] < Reg[Rb] then Reg[Rc]  $\leftarrow$  1
                else Reg[Rc]  $\leftarrow$  0

```

Without constant		With constant	
OPCODE	Name	OPCODE	Name
0x24	CMPEQ	0x34	CMPEQC
0x25	CMPLT	0x35	CMPLTC
0x26	CMPLT	0x36	CMPLTC

Table 5.3: Comparison instructions' OPCODE and name definition on the β -architecture.


```

CMPLTC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                        if Reg[Ra] < SXT[literal] then Reg[Rc]  $\leftarrow$  1
                        else Reg[Rc]  $\leftarrow$  0
CMPLE(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                        if Reg[Ra]  $\leq$  Reg[Rb] then Reg[Rc]  $\leftarrow$  1
                        else Reg[Rc]  $\leftarrow$  0
CMPLEC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                        if Reg[Ra]  $\leq$  SXT[literal] then Reg[Rc]  $\leftarrow$  1
                        else Reg[Rc]  $\leftarrow$  0

```

Listing 5.3: Comparison instructions definition.

5.4.5 Shifting instructions.

A logical shift is a bitwise operation that shifts all the bits of its operand. You can shift all bits of a operand to the right or to the left and for one or several place. In our β -machine we define 6 different shift operations, as usual 3 working only with registers and 3 working with constants. For **SHL**, Ra is shifted to the left by the number of positions given in the 5 least significant bits in Rb (**SHL**) or literal (**SHLC**), and padding with 0. **SHR**(C) shifts to the right, similarly to **SHL**(C) on Ra. Finally, **SRA**(C) shifts to the right, similarly to **SHL**(C) but the sign bit ($\text{Reg}[Ra]_{31}$) is used for padding (because **SRA** is an arithmetical shift right meaning that with this technique we keep the sign bit intact). As for the other families of instructions presented above we list the shifting instructions' **OPCODE** and assembly name in table 5.4 and we expose their executions on the β -machine in listing 5.4.

```

SHL(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra]  $\ll$  Reg[Rb]4:0
                Reg[Rc]Reg[Rb]4:0:0  $\leftarrow$  0
SHLC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra]  $\ll$  literal4:0
                Reg[Rc]literal4:0:0  $\leftarrow$  0
SHR(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra]  $\gg$  Reg[Rb]4:0
                Reg[Rc]Reg[Rb]4:0:0  $\leftarrow$  Reg[Ra]31
SHRC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra]  $\gg$  literal4:0
                Reg[Rc]literal4:0:0  $\leftarrow$  Reg[Ra]31
SRA(Ra,Rb,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra]  $\sim\ll$  Reg[Rb]4:0
                Reg[Rc]Reg[Rb]4:0:0  $\leftarrow$  Reg[Ra]31
SRAC(Ra,literal,Rc): PC  $\leftarrow$  PC + 4
                Reg[Rc]  $\leftarrow$  Reg[Ra]  $\sim\ll$  literal4:0
                Reg[Rc]literal4:0:0  $\leftarrow$  Reg[Ra]31

```

Listing 5.4: Shifting instructions definition.

5.4.6 Memory read and write.

We don't just have instruction performing by the ALU in the β -machine but also operation performing on the memory, namely read and write operations. These operations are well named because their name exactly correspond of what they did, a read operation get back a value from the memory and a write operation place a value inside the memory. Table 5.5 list the operation **OPCODE** and names and listing 5.5 describe they execution on the β -machine.

```

LD(Ra,literal,Rc) : PC  $\leftarrow$  PC + 4
                    Reg[Rc]  $\leftarrow$  Mem[Reg[Ra] + SXT(literal)]

```

Without constant		With constant	
OPCODE	Name	OPCODE	Name
0x2C	SHL	0x3C	SHLC
0x2D	SHR	0x3D	SHRC
0x2E	SRA	0x3E	SRAC

Table 5.4: Shifting instructions' **OPCODE** and name definition on the β -architecture.

OPCODE	Name
0x18	LD
0x19	ST

Table 5.5: Memory instructions' **OPCODE** and name definition on the β -architecture.

```
ST(Rc, literal, Ra) : PC ← PC + 4
                     Mem[Reg[Ra] + SXT(literal)] ← Reg[Rc]
```

Listing 5.5: Memory instructions definition.

5.4.7 Branching instructions.

In order to re-execute part of a programs like it's namely the case with loop, you need to be able to go back again in a certain instruction in the memory instruction on the β -machine. This is the purpose of the branching instruction **JMP** that will place the address of the next instruction after the **JMP** instruction into **Rc**. Note that the two least significant bits in **Ra** are ignored (that is, considered to be 0) so that the address is aligned with the words in memory as shown in listing 5.6. Also, **JMP** instructions are not conditional so this instruction will create infinite loops. We then also need **BEQ**, **BNE** instructions that will execute **JMP** if only a certain condition is respected. These two last are then also interesting in order to manage potential condition in codes.

OPCODE	Name
0x1B	JMP
0x1D	BEQ
0x1E	BNE

Table 5.6: Branching instructions' OPCODE and name definition on the β -architecture.

```
JMP(Ra, Rc) : PC ← PC + 4
              Reg[Rc] ← PC
              PC ← Reg[Ra] & 0xFFFFF0
BEQ(Ra, label, Rc) : PC ← PC + 4
                   Reg[Rc] ← PC
                   if Reg[Ra] == 0 then
                       PC ← PC + 4 × SXT(literal)
BNE(Ra, label, Rc) : PC ← PC + 4
                   Reg[Rc] ← PC
                   if Reg[Ra] == 1 then
                       PC ← PC + 4 × SXT(literal)
```

Listing 5.6: Branching instructions definition.

SUMMARY

- A **Von Neumann** computer architecture is made of a **processing unit** (composed of an **ALU** and **processor registers**), a **control unit** (containing **instruction registers** and **program counter**), a **unique memory** (storing **data** and **instructions**), an **external mass storage** and **input/output mechanisms**.
- The **instruction set architecture** (ISA) of a computer gives the **backbone** of its **architecture**. We denote 2 families of ISA, the **Reduced instruction set computer** (RISC) family and the **Complex instruction set computer** (CISC) one.
- The **β -machine** we want to create will be
 - A **RISC** computer;
 - A **32-bits architecture**;
 - Each **instruction** will be **32-bits** long;
 - Having **32 registers**;
 - **Memory addresses** will be **32-bits** long;
 - The program counter **PC** will be **32-bits** long and will only be a **multiple of 4** (the two least significant bits are 0);
- The instruction set will be composed of **arithmetic**, **logic**, **comparison**, **shifting**, **memory read/write**, **branching** instruction. Each instruction will be identified by its **OPCODE**.

6 Building the β -machine.

Now that all standards (instructions' names, OPCODE, memory size, instructions size,...) has been defined we can start building the β -machine with hardware materials. In this section we will ran through all basics β -machine components and implement them using basics logical gates introduced in section 2. After what we build the β -machine with the architecture described in subsection 5.3.

6.1 β -machine main components.

6.1.1 Instruction memory.

The instruction memory as shown in figure 6.1 of the β -machine contain all the 32-bits instructions that will be executed. It will read the instruction in the program counter addresses (PC) and send it to

several others parts described below. This is a component with a single input and a single output

- **Input(s):**
 - **A:** an address on 32 bits;
- **Output(s):**
 - **B:** a β -machine instruction on 32 bits;

6.1.2 Data memory.

The data memory (represented on figure 6.2) contain all the data used by the β -machine. We found out inside this RAM memory the variables used on our operations but we can also store in it results from these functions. Basically this component is a 32-bits RAM with 8-bits addresses ($2^8 = 256$ words of 32-bits). It have 3 inputs for a single output

- **Input(s):**
 - **Adr:** an address on 32-bits;
 - **Wr:** a single bit set to 1 if we want to write inside the RAM and on 0 if its a read operation;
 - **WD:** the 32-bits writing data, it's what it place in the RAM at address Adr if Wr is active;
- **Output(s):**
 - **RD:** the 32-bits reading data;

6.1.3 ALU.

The arithmetic and logic unit has already been defined in subsection 3.5. This is the heart of our architecture who perform basics operations on its inputs. As it can be seen in figure 6.3 this element has 3 inputs for a single output

- **Input(s):**
 - **A:** a 32-bits ALU's argument;
 - **B:** a 32-bits ALU's argument;
 - **ALUFN:** the 4-bits label of the performing operation by the ALU;
- **Output(s):**
 - The 32-bits result of the performing operation;

6.1.4 Register files.

The register file is simply an array of processor registers in a central processing unit (CPU). These registers contain current arguments of the beta machine value for an executing program. In our β -machine the register files contain $2^5 - 1 = 31$ registers ($R0$ is reserved to the constant 0), have the ability of write in one register and read 2 of them simultaneously (*cf.* figure 6.5). This component has 5 inputs and 2 outputs

- **Input(s):**
 - **RA1:** a first 5-bits register address for reading;
 - **RA2:** a second 5-bits register address for reading;
 - **WA:** a 5-bits register address for writing;
 - **WD:** data on 32-bits to write in register addressed by WA;
 - **WE:** a single bit for enable writing (1) or not (0), at the next clock trigger;
- **Output(s):**
 - **RD1:** 32-bits data in register addressed by RA1;
 - **RD2:** 32-bits data in register addressed by RA2;

6.2 The β -machine.

In this subsection we will construct the β -machine step-by-step starting from its simplest from to its most elaborate. Note that in this paper, we consider the most basic β -machine already able to perform operation with constant while in the course we also consider β -machine without ALU and only able to perform ALU's operations between registers.

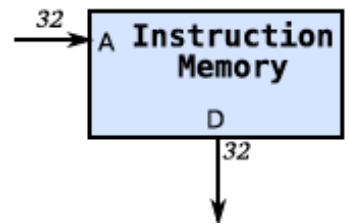


Figure 6.1: Instruction memory of the β -machine.

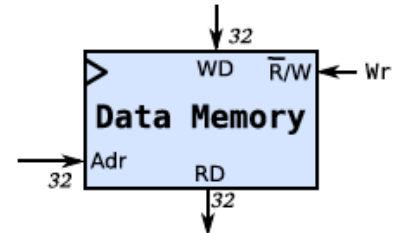


Figure 6.2: Data memory of the β -machine.

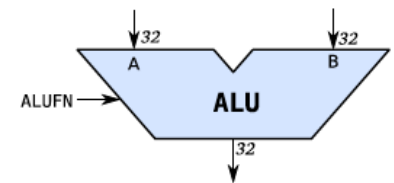


Figure 6.3: ALU of the β -machine.

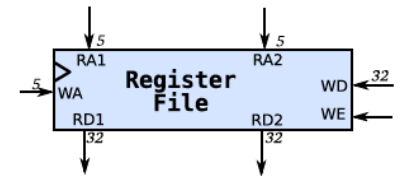


Figure 6.4: Register file of the β -machine.

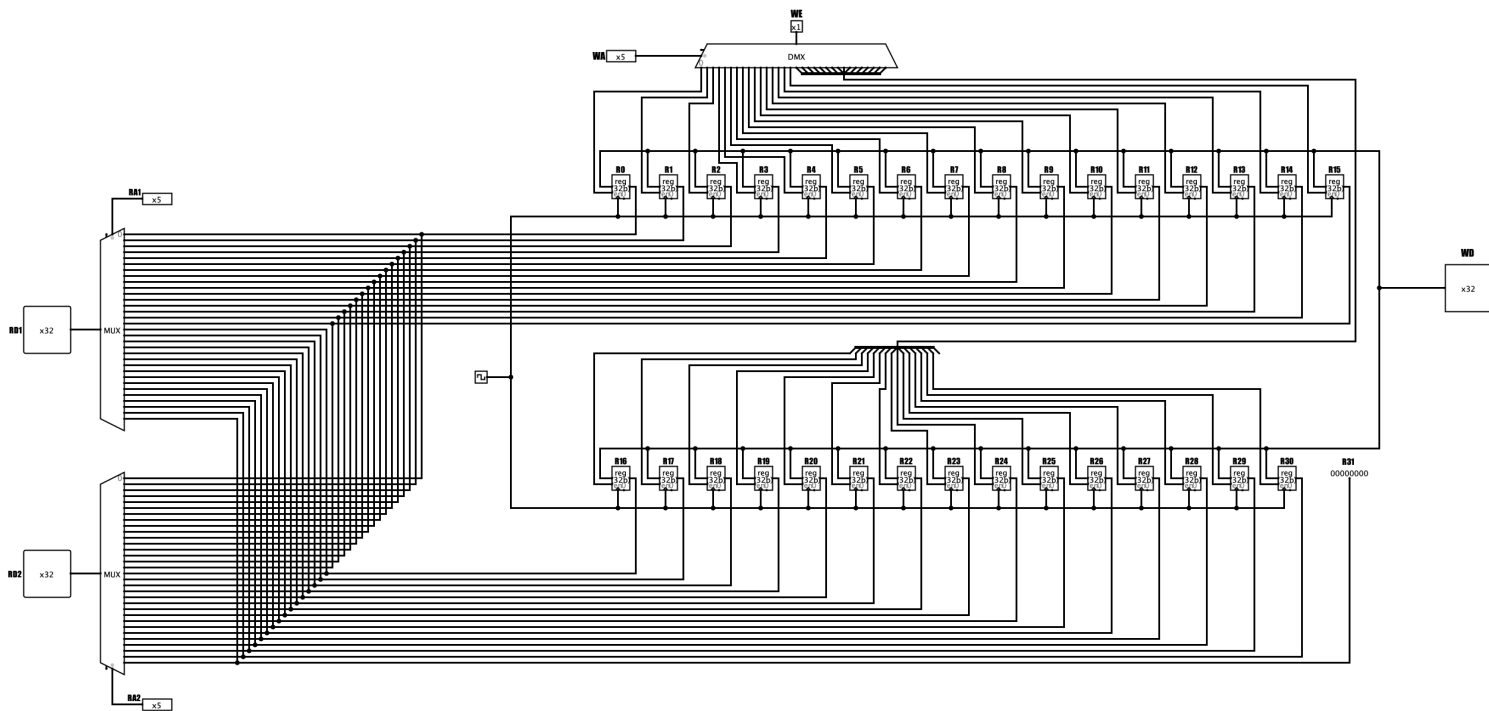


Figure 6.5: Inside of the complete register file of the β -machine.

6.2.1 Basic β -machine.

Basically, the β -machine is simply composed of a program counter (PC), an instruction memory, a register file, a control logic and an ALU. as shown in figure 6.6, the program counter is simply a 4-by-4 counter who read the executed instructions by the β -machine in the instruction memory. The program counter thus send a 32-bits address to the instruction memory and the instruction in this address is then load on the output of the instruction memory. In this 32-bits instruction we found out

- **On bits <0-15>:** the literal, or the constant (C:SXT(lit)) sometimes used for ALU's operations.
- **On bits <11-15>:** the Rb value, or the RA2 value in terms of register files input. Note that this value is not used if we perform an operation with constant, that's why the value of Rb's address in register file can collides with the constant field. It's the purpose of the multiplexer just before the ALU's B field to determine whether or not the performing operation is between registers only or involve a register and a constant. In this register the information is simply read and pass to RD2.
- **On bits <16-20>:** the Ra value, or the RA1 value in terms of register files input. In this register the information is simply read and pass to RD1.
- **On bits <21-25>:** the Rc value which is the register where information is write inside the register file.
- **On bits <26-31>:** the OPCODE field who permit to the control logic to load β -machine's controls signals at the address OPCODE inside its own ROM.

As said above, the control logic is a ROM containing 32-bits control signals relatives to the differents OPCODE inside the β -machine. These signals will dictate the behaviour of the β -machine. From now, with the basic form on which we are working they are limited to 3

- **On bit <3>:** the BSEL signal who will, as explain earlier, select if the executed operation by the ALU implies 2 registers or a register and a constant.
- **On bits <6-9>:** the ALU function flag which is an ALU's output informing it which operation to execute on his entries.
- **On bits <11>:** the WERF signal activating the writing inside the register Rc in the register file.

REMARK

In the course there is no specified bit values for output signals of the control logic. In this writing I choose to order the control signals as on my own β -machine. You can change them the way you want but make sure that your binary instruction on your own control logic ROM will fit to the wanted β -machine's signals for the given OPCODE operation.

As illustrate clearly by figure 6.6, A is the 32-bits first ALU input and is the RD1 value while B is its second 32-bits input and is either a constant or RD2's value.

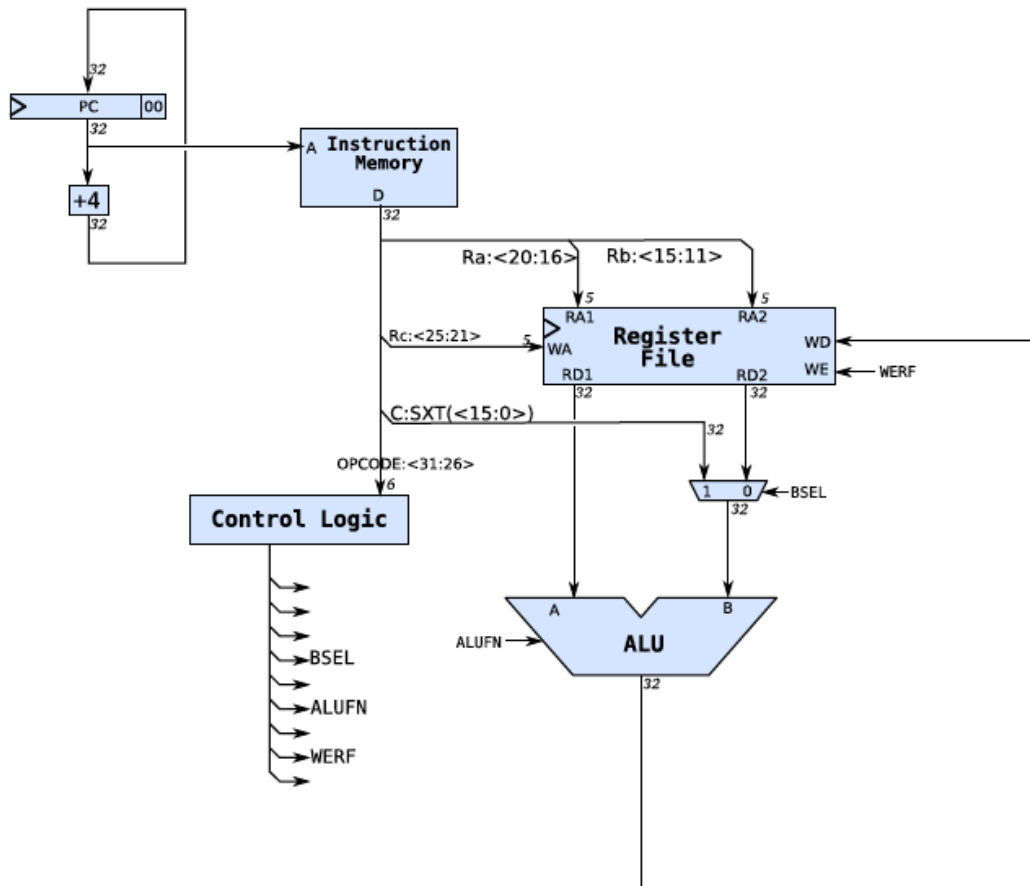


Figure 6.6: β -machine 1.1 with ALU (1.0 only implement ALU' operations between registers).

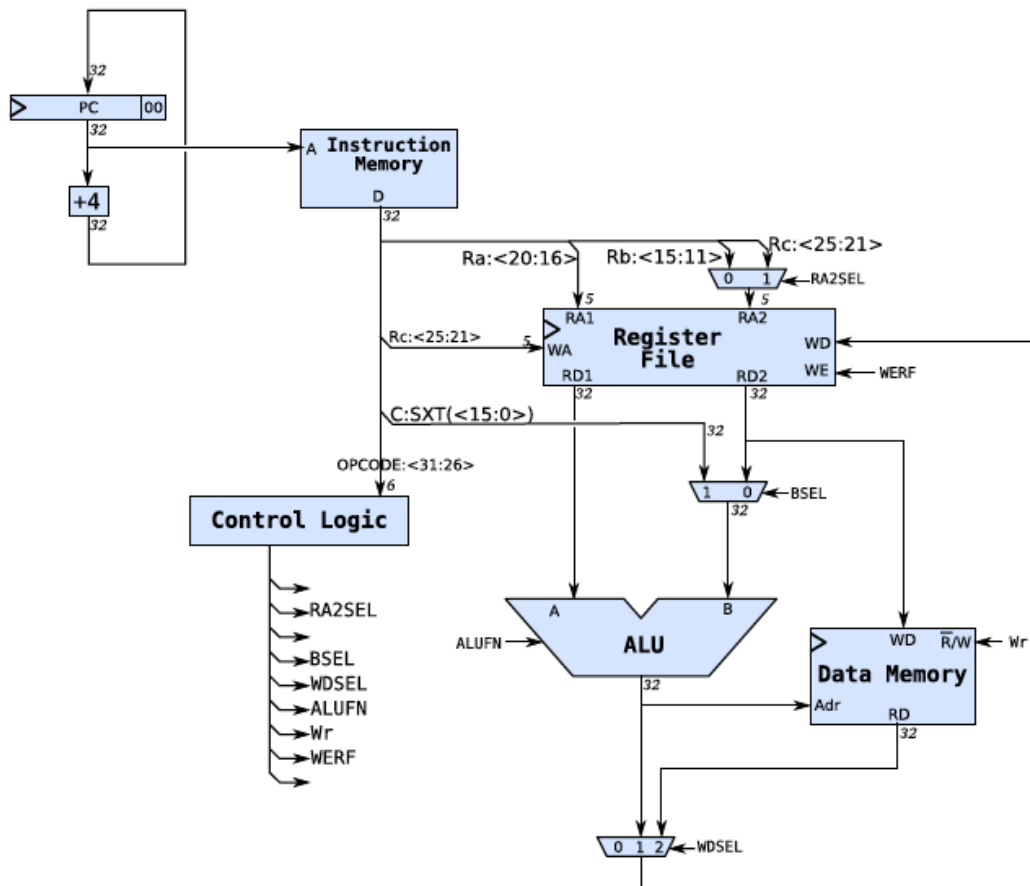
6.2.2 β -machine with memory instructions.

The memory instructions are described in listing 5.5. In order to perform these instructions, we need a data memory zone in our β -machine, that's the major update here as shown in figure 6.7. In order to perform these new operations we need new control signals from the control logic output.

- **On bit <2>:** RA2SEL that will permit to use either Rb or Rc as second operation register.
- **On bits <4-5>:** is the WDSEL that selects either PC + 4 (0), the ALU result (1), or the read data from the memory (2) as write data for the register file.
- **On bit <10>:** Wr who will said to the data memory whether or not the β -machine store data in memory (will perform a write operation inside the data memory at the next clock trigger).

As input of the data memory, as it can be seen in the figure 6.7, we have WD set to the 32-bits RD2 value, ADr who is the 32-bits ALU's output value and as explain Wr given by the control logic that inform the data memory if we perform a LD (Wr is 0 and we simply read data) or a ST (Wr is 1 and we write data in address WD). The output of the data memory is then the 32-bits value located at the ADr in the RAM contained by the data memory.

Figure 6.7: β -machine 2.1 with data memory (2.0 only implement the LD instruction).



6.2.3 β -machine with branching instruction.

A branching instruction is performing by simply execute instructions from an other place that the expected (at address PC+4) in the instruction memory. In this course we studied 3 different branching instruction, JMP (jump), BEQ (branching equality) and BNE (branching non-equality) defined in listing 5.6. The β -machine able to perform these new operations is illustrated in figure 6.8 where we can again notice the apparition of a new control signal.

- **On bit <0-1>:** PCSEL who will permit to select the input of the instruction memory, meaning that we can select the next executed instruction by the β -machine. PCSEL allows to select 3 instruction memory' 32-bits input addresses, either PC + 4 (0), the sum 4 * SXT(Literal) with PC + 4 (1) or the first output (JT) of the register file (2).

With these new mechanisme we can now perform loop and condition with our β -machine. The address where of the next executed instruction after a branch operation can depend of the PC value or can be completely independent of it by using JT.

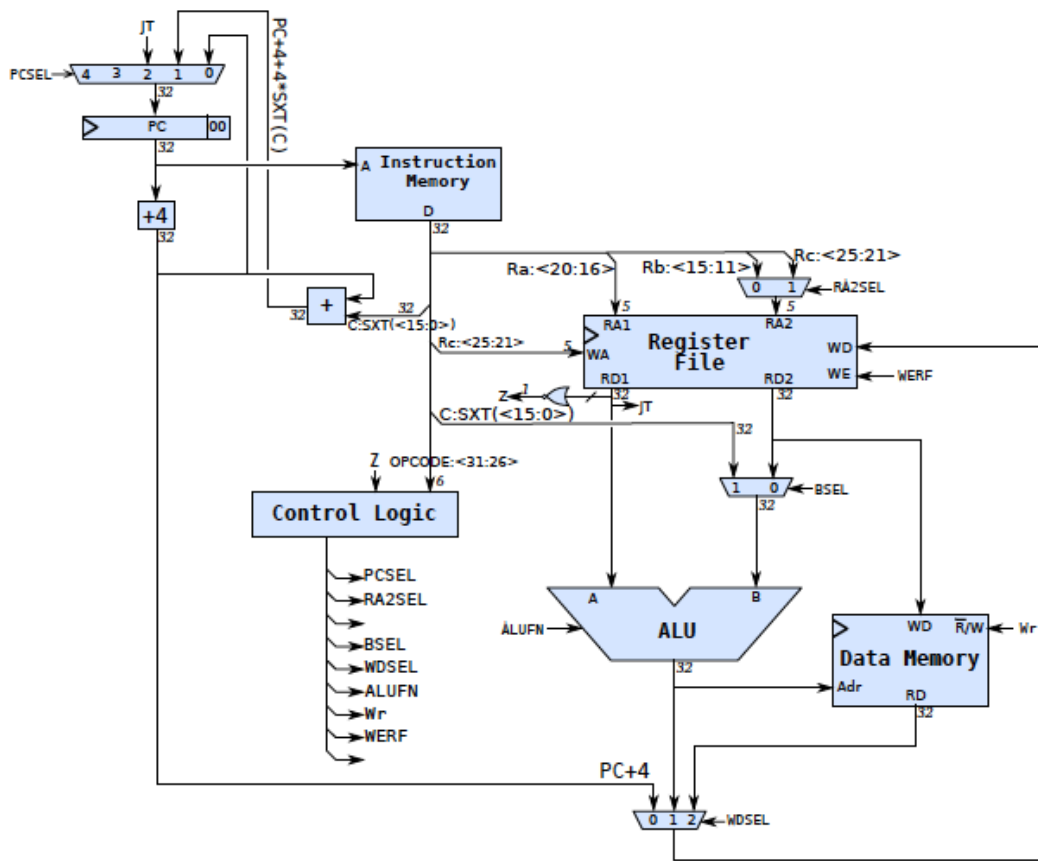


Figure 6.8: β -machine 3.1 with branching operations (3.0 only implement the JMP instruction).

6.2.4 The full β -machine.

With the full β -machine we introduce the management of trap inside our program. With this come again new output of the control logic

- **On bit <12>:** ASEL that permit the selection either of RD1 (0) or $PC+4+4*SXT(literal)$ (1) as input A of the ALU.
- **On bit <13>:** WASEL that save the current program counter into register R30 (1), now called XP or exception pointer or simply continue the execution normally if nothing bad occurs (0).

The purpose of these last updates in the β -machine is to handle possible exceptions during the program execution, such as segmentation fault as instance. When an exception is triggers, the β -machine will save the place where the exception occurs in terms of program counter inside XP. After what it will launch an exception management context by selection ILL0P (PCSEL = 3, to manage illegal operations, with wrong OPCODE values) or XAdr (PCSEL = 4, for other interruptions) value as input of the program counter. After managing the triggered exception, it can return to the base program execution by simply reset the value of the program counter by the one saved in R31 or XP.

You can also see that now the control logic has a new input signal IRQ, for interruption request. Exceptions can be of several types, such as out-of-bound memory load or write, invalid addresses jumps, wrong OPCODE given value,... Each of these issues need to have its own programs stored in the instruction memory to detect and solve it.

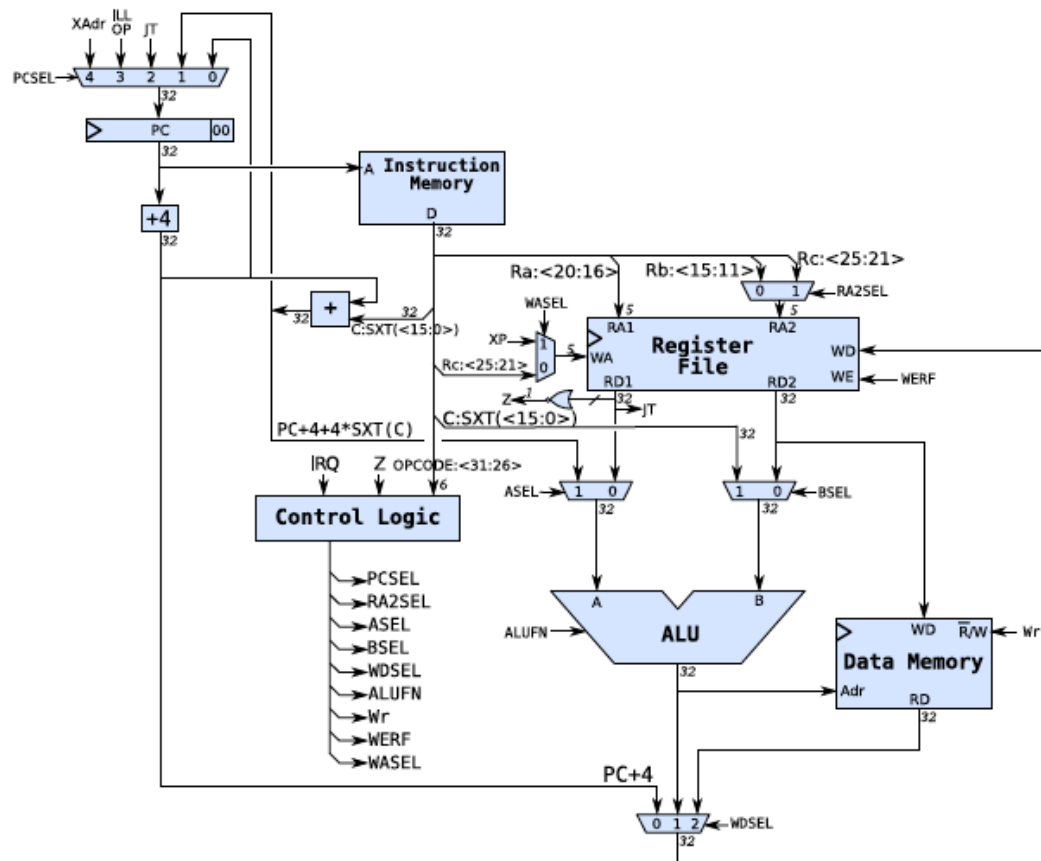


Figure 6.9: β -machine 4.0 with trap and interrupt control.

SUMMARY

- The **instruction memory** of the β -machine contain all **binary instructions** that will be **executed** by the β -machine.
- The **data memory** of the β -machine is the equivalent of a disk where all **data** used by running **programs** will be **load** and **store**.
- The **register file** is an array of processor **registers** located in the **CPU** itself.
- The **β -machine 1.0** is composed of a **program counter (PC)**, an **instruction memory**, a **register file**, a **control logic** and an **ALU**.
- The **β -machine 1.1** add the functionalities of **performing operations with constant**.
- The **β -machine 2.0** implement the **LD** memory instruction additionally to the previous defined.
- The **β -machine 2.1** implement either the **LD** and the **ST** memory instruction additionally to the previous defined.
- The **β -machine 3.0** implement the **JMP** instruction additionally to the previous defined.
- The **β -machine 3.1** implement either the **JMP** and the **BR** memory instruction additionally to the previous defined.
- The version **β -machine 4.0**, which is the final, implement a **trap** and **interrupt manager**.

7 Programming the β -machine.

As a reminder, inside the β -machine the instructions have the following format (where x is a binary value) depending if they used constants or not.

xxxxxx	xxxxx	xxxxx	xxxxx	xxxxxxxxxxxx
OPCODE	Ra	Rb	Rc	Unused

(7.1)

xxxxxx	xxxxx	xxxxx	xxxxxxxxxxxxxxxx
OPCODE	Ra	Rc	SEXT(literal)

(7.2)

When a code in high level programming language (C, python, Java,...) is executed in a computer, the machine will translate it into a processor readable language (the assembly) and than these assembly

instructions will be translate again in machine code, meaning that from a high level instruction you get one of the two format (7.1)-(7.2).

Example 7.1: Programming an addition.

In an high level programming language when you want to perform the addition of two variable you will most of the time simply write

$$a = b + c$$

In the β -machine assembly language, we will write this instruction

ADD(R2,R3,R4)

For the β -machine this means

$$\text{Reg}[4] \leftarrow \text{Reg}[2] + \text{Reg}[3]$$

Or, in format (7.1)

100000	00100	00010	00011	00000000000
--------	-------	-------	-------	-------------

= 0x80821800

In computer science there are two way to execute a program, by interpreting it or by compiling it. When you interpret a program the high-level code induce sequence of low-level instructions and the interpreter emulates, step by step, the instructions of the high-level code by running low-level instructions as it is namely the case for Python scripts. On the contrary when you compile a program the high-level code is translated to low-level instructions and the compiler translates the whole program as a pre-process. Then you are able to run the program, as it's the case for the C language.

7.1 Compiling to assembly.

In order to facilitate our reasoning for the translation of high-level scripts to assembly ones we will make some assumptions here

- We only consider a minimalist subset for variables
 - No floats, enum, struct, union,... Only 4 bytes integers and pointers (*i.e.* arrays);
 - No memory allocation;
 - No library functions (*e.g.* on strings);
- We consider expressions of several kind
 - Constants (*e.g.* 1234);
 - Variables (*e.g.* x, a[12], *p);
 - Assignments (*e.g.* x = 1);
 - Operations (*e.g.* 3 * x);
 - Function calls (*e.g.* Fibonacci(10));
- We consider statements as
 - Simple: expr;
 - Compound: { statement1 statement2 ... };
 - Conditional: if(expr) then_statement else else_statement;
 - Loop: while(expr) statement, do statement while(expr), for(expr_init; expr_cond; expr_inc);

7.1.1 Compiling expressions.

In order to compile an expression, we will use the COMPILEEXPR() function. This function takes an expression in argument, creates code to compute the expression and then returns the number of the register that will store the result of computing the expression. This could be perform in several ways as shown in example 7.2-7.4.

Example 7.2: COMPILEEXPR() examples of execution for constant.

- COMPILEEXPR(1234) could produce assembly code ADDC(R31,1234,Rx) and return Rx, where x is the number of an unused register;
- For large constants (greater than 16 bits): COMPILEEXPR(123456) would store 123456 in some memory (4 bytes) at address c1, would produce the assembly code LD(R31,c1,Rx) and return Rx, where x is the number of an unused register;
- Alternatively (123456=0x0001E240): COMPILEEXPR(123456) would produce assembly code ADDC(R31,0x0001,Rx) SHLC(Rx,16,Rx) ORC(R31,0xE240,Rx) and return Rx,

REMARK

A good compiler will always take care of simplifying expressions in order to use as few registers as possible.

where x is the number of an unused register;

Example 7.3: COMPILEEXPR() examples of execution for variables.

- **COMPILEEXPR(x):** it is assumed that declaration of x has allocated 4 bytes in memory at address vx initially set to 0. The compiler would produce the assembly code `LD(R31,vx,Rx)` and return Rx , where x is the number of an unused register;
- **COMPILEEXPR(array[$expr$]):** if $array$ is a declared array, it is assumed declaration of a has allocated sufficiently many bytes in memory at address va . Assume $expr$ is stored in register Rx . For simplicity consider the case of an array of integers. The compiler would produce the assembly code `SHL(Rx, 2, Rx) LD(Rx,va,Rx)` and return Rx ;

Example 7.4: COMPILEEXPR() examples of execution for assignments and operations.

- **COMPILEEXPR($x = expr$):** it is assumed declaration of x has allocated 4 bytes in memory at address vx initially set to 0. Assume $expr$ is stored in register Rx . The compiler would produce the assembly code `ST(Rx,vx,R31)` and return Rx .
- **COMPILEEXPR($expr1 + expr2$):** Assume $expr1$ (resp. $expr2$) is stored in register Ra (resp. Rb). The compiler would produce the assembly code `ADD(Ra,Rb,Rx)` and return Rx .

7.1.2 Compiling statements.

In order to compile a statement, we will use the **COMPILESTMT()** function. Compiling a simple statement is the same as compiling an expression but the returned register is discarded. We have the basic rules in (7.3)-(7.4).

$$\text{COMPILESTMT}(expr;) = \text{COMPILEEXPR}((expr)) \quad (7.3)$$

$$\begin{aligned} \text{COMPILESTMT}(\{statement1 \ statement2 \ \dots \ statementN\}) = \\ & \text{COMPILESTMT}(statement1) \\ & \text{COMPILESTMT}(statement2) \\ & \vdots \\ & \text{COMPILESTMT}(statementN) \end{aligned} \quad (7.4)$$

Examples 7.5-7.6 explain how the 2 statements found in practice (conditions and loops) are in reality compiled by a machine.

Definition 7.1 - Labels, address, relative jumps and BEQ.

As a reminder, labels are used so that the compiler from assembly to machine code is aware of some important addresses for the execution. A label is the address of an instruction. While converting to machine code, labels are used to compute the relative jumps in BEQ/BNE instructions:

$$\text{literal} = \frac{\text{label} - \text{PC}}{4} - 1$$

Example 7.5: COMPILESTMT() examples of execution for conditional statements.

- **COMPILESTMT(if ($expr$) then statement):**
 1. Generate code for the $expr$ expression and assume that Rx is the returned register;
 2. Add instruction `BEQ(Rx,label_endif,R31)`;
 3. Compile **then** statement;
 4. Add label `label_endif`;
- **COMPILESTMT(if ($expr$) then statement else else_statement):**
 1. Generate code for the $expr$ expression and assume that Rx is the returned register;
 2. Add instruction `BEQ(Rx,label_else,R31)`;

3. Compile then_statement;
4. Add instruction BEQ(R31,label_endif,R31);
5. Add (unique) label label_endif;
6. Compile else_statement;
7. Add (unique) label label_endif;

Which leave us with pseudo-codes in listings 7.1-7.2.

```

CompileExpr(expr) → Rx
BEQ(Rx,label_endif,R31)
CompileStmt(then_statement)
label_endif:

```

Listing 7.1: Assembly pseudo code of a condition statement **if**.

```

CompileExpr(expr)→ Rx
BEQ(Rx,label_else,R31)
CompileStmt(then_statement)
BEQ(R31,label_endif,R31)
label_else:
    CompileStmt(else_statement)
label_endif:

```

Listing 7.2: Assembly pseudo code of a condition statement **if...else...**

Example 7.6: COMPILESTMT() examples of execution for loop statements.

- **COMPILESTMT(while (expr) statement):**
 1. Add (unique) label label_while_test;
 2. Generate code for the expr expression and assume that Rx is the returned register;
 3. Add instruction BEQ(Rx,label_while_end,R31);
 4. Compile statement;
 5. Add instruction BEQ(R31,label_while_test,R31);
 6. Add (unique) label label_while_end;
- **COMPILESTMT(do statement while (expr)):**
 1. Add (unique) label label_while_start;
 2. Compile statement;
 3. Generate code for the expr expression and assume that Rx is the returned register;
 4. Add instruction BNE(Rx,label_while_start,R31);

Which leave us with pseudo-codes in listings 7.3-7.4.

```

label_while_test:
    CompileExpr(expr) → Rx
    BEQ(Rx,label_while_end,R31)
    CompileStmt(statement)
    BEQ(R31,label_while_test,R31)
label_while_end:

```

Listing 7.3: Assembly pseudo code of a loop statement **while**.

```

label_while_start:
    CompileStmt(statement)
    CompileExpr(expr) → Rx

```

REMARK

The for loops can always been computed with while loops. Indeed, the first is just a shortcut of the second and we have for(expr_init; expr_cond; expr_inc) equivalent to

```

expr_init;
while(expr_cond){
    statement
    expr_inc;
}

```

```
BNE(Rx,label_while_start,R31)
```

Listing 7.4: Assembly pseudo code of a loop statement **do...while**.

7.1.3 Compiling functions.

In computer sciences, a function is a sequence of program instructions that performs a specific task, packaged as a unit. This unit can then be reused in programs wherever that particular task should be performed. A function thus allow to use part of program in an other place in it. This code part uses arguments, which are specified to the function when it's invoke.

With our actual machine we could try to execute assembly programs but several issues would appear

- How can we be able to pass arguments to the function?
- How will we handle local variables?
- How should a function return a value?
- How calling functions inside a function, and maybe a function calling itself (recursive call)?
- How should we memorizing the return address?

In order to solve these problems, in this courses we defined some conventions. First of all, in order to memorize the return address of the function, we will simply designate a register who will retain this value. The used register will be R28 by conventions, called the linkage pointer (LP).

Calling a function is thus simply done with instruction `BEQ(R31,label,LP)`, which automatically stores the return address (*i.e.* the current address plus 4) into LP.

After the function execution, returning from a function call is simply done with instruction `JMP(LP)`.

7.1.4 The stack.

A stack is a basic data structure well know in computer science. This data structure is of variable memory size, LIFO (Last Input, First Output) oriented and its interface is composed of two operations only, the PUSH operation who simply add a value in the stack and the POP operation who get back the top value of the stack. Since only those functions are used, allocation and disallocation always happen at the top of the stack and are very efficient since it's essentially updating a pointer to the top of this data structure. Last but not least, the way a stack works really suits the way functions are called (last started/first ended vs. last in/first out).

For this course we implemented a stack which contain sequence of (groups of 4) bytes in memory, starting from a predefined address. We also selected a specific register R29 (called SP for Stack pointer) to point to the next free slot on the stack. The interface of our stack has been implemented using macros who are given in listing 7.5

```
; Adding a word (4 bytes) on the stack
.macro PUSH(RA)  ADDC(SP,4,SP) ST(RA,-4,SP)
; Removing a word (4 bytes) from the stack
.macro POP(RA)   LD(SP,-4,RA)  ADDC(SP,-4,SP)
; Allocate space on the stack for N words
.macro ALLOCATE(N)  ADDC(SP,N*4,SP)
; Free space on the stack (N words)
.macro DEALLOCATE(N) SUBC(SP,N*4,SP)
```

Listing 7.5: Interface of the stack used in the programmable β -machine

In order to make full use of the stack capabilities, several new conventions has to be consider during the assembly translation of a code. First, the function arguments will be put on the stack before its calling and local variables could also be put in it during its execution. BP (Base of frame Pointer) will be set to the position of local variables. BP and SP are also saved on the stack, to provide a

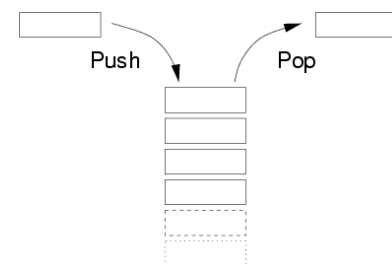


Figure 7.1: Stack representation and interface illustration.

recursion-safe environment and allow to recover the full state of the machine before the function call. A draft of the stack for a typical execution of a function of n arguments and m local variables is shown at figure 7.2. A final note for the stack is that we always will put parameters on it from most right argument to most left in the prototype, this represent the last convention presented here.

Example 7.7: Illustration of the stack used, recursive factorial implementation.

```
int fact(int n){
    if(n==0)
        return(1);
    return(n*fact(n-1));
}
```

Listing 7.6: C implementation of the factorial function

In C language we can implement the factorial function as described in listing 7.6. We can translate this code in β -assembly language using the stack as shown and commented in listing 7.7

```
fact:
    PUSH(LP)      | save the return address
    PUSH(BP)      | save the previous frame
    ADDC(SP,0,BP) | initialize the current frame
    PUSH(R1)      | R1 is used in the function, save it
    LD(BP,-12,R1) | load argument n into R1
    BNE(R1,big)   | compare n to 0
    ADDC(R31,1,R0) | n=0, return 1
    BEQ(R31,rtn,R31) | go to the return sequence
big:
    SUBC(R1,1,R1) | compute n-1 into R1
    PUSH(R1)      | put the argument for recursive call on stack
    BEQ(R31,fact,LP) | recursive call
    DEALLOCATE(1) | free space
    LD(BP,-12,R1) | load n into R1
    MUL(R1,R0,R0) | n*fact(n-1) into R0
rtn:
    POP(R1)       | restore R1
    ADDC(BP,0,SP) | restore SP
    POP(BP)       | restore BP
    POP(LP)       | restore return address
    JMP(LP,R31)   | return
```

Listing 7.7: β -assembly translation of the factorial function

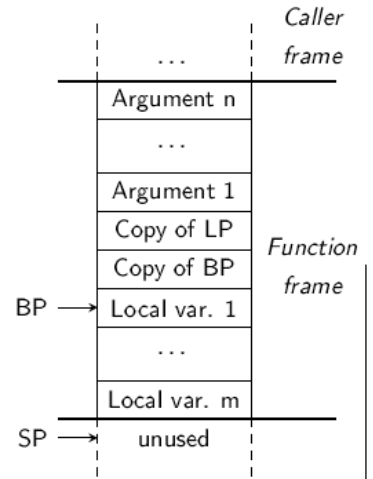


Figure 7.2: Course stack organisation for programs execution.

SUMMARY

- When a **code** in **high level programming language** is executed in a computer, the **compiler** will translates it into a **processor readable language** than the **assembler** translates this translation in **machine code**.
- They are **two** ways of **executing** a program, either by **interpreting** or by **compiling** it.
- labels are used to compute the relative jumps in BEQ/BNE instructions and they are calculated as

$$\text{literal} = \frac{\text{label} - \text{PC}}{4} - 1$$

- A **stack** is a **LIFO** (Last Input,First Output) oriented **data structure** of **variable memory size** used to pass **functions'** **arguments** and saving **return address** in the **linkage pointer** (LP, R28).
- The **base pointer** (BP) and the **stack pointer** delimit the stack by respectively giving its **start** and its **end** address.
- Parameters** will always be **put on the stack** from **most right** argument to **most left** in the function prototype.

8 The β -machine with a bus architecture.

Now that we have implemented a first β -machine, we can improve it in order to have better performances. The β -machine with a bus architecture is a Von Neumann machine, which means that it use only one memory for both instructions and data. Also as its name suggests, this new machine use a bus in order to pass data through its components. In early computer, buses were parallel electrical wires with multiple hardware connections, but the term is now used for any physical arrangement that provides the same logical function as a parallel electrical bus. Modern computer buses can use both parallel and bit serial connections, and can be wired in either a multi-drop (electrical parallel) or daisy chain topology, or connected by switched hubs, as in the case of USB. With these improvement comes the microcode of the β -machine, because now the executed instructions takes more than a single clock cycle. The new β -machine architecture is shown at figure 8.1.

8.1 The updated β -machine.

With this new machine the instruction set architecture remains the same as for our previous β -machine. Now the bus (shared interconnection lines) transmit between the components. This is a bidirectional bus because information is not always going in the same direction. An extensive use of controlled buffers (*i.e.* three states buffers) is mandatory here because an element connected to a bus can be

- **Active:** it can set 0 or 1 on each binary line;
- **Inactive:** it has no influence on the value on the bus, but it can read;

The components show in figure 8.1 share resemblance with the previous ones for the simple β -machine, but are adapted to the bus architectures. The biggest changes are on the memory components that are now two in number (SRAM and DRAM respectively for static and dynamic RAM) instead of 3 before (instruction and data memory, and register file). New registers are now set to store results locally where needed. Those, as well as the RAM, the machine registers (R0-R31), PC, and the Control unit, share the same clock. The bus is a shared device used to move 32-bit data like instructions, PC, data

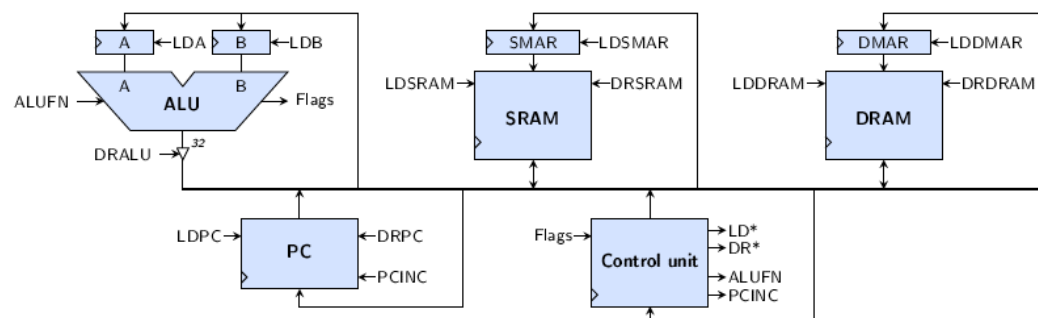


Figure 8.1: β -machine with a bus architecture.

to write to or read from memory (DRAM), data to write to or read from register file (SRAM), address for memory, (5 bit) address for register file. The information on the bus can also be stored locally as represented in figure 8.1

- **A and B:** these are two 32-bit registers that stores the two ALU's arguments;
- **SMAR:** a 5-bit register that stores the address for the register file;
- **DMAR:** a 32-bit register that stores the address for the memory;
- A 32-bit register that stores the instruction in the control unit;
- **Flags:** from the ALU are stored in the Control unit;

8.2 The PC.

As for the basic β -machine, the program counter is basically a register which initial value is 0 as shown in figure 8.2. Its value is updated, at the next clock trigger if either PCINC or LDPC is set to 1.

- **PCINC is 1:** PC gets PC + 4 value;
- **LDPC is 1:** and PCINC is at 0 then PC gets the value on the bus;

Finally, if the if the DRPC signal is set to 1, PCINC's value is put on the bus.

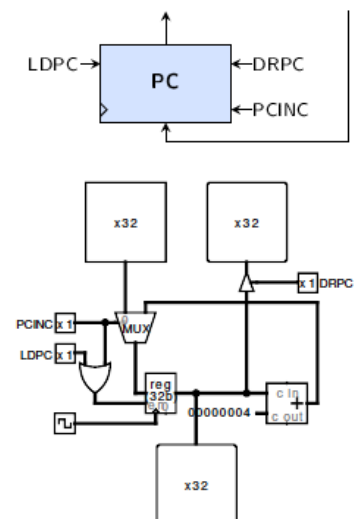


Figure 8.2: Program counter of the β -machine with a bus architecture.

8.3 The ALU.

From the figure 8.3 we can see that the β -machine with a bus architecture's ALU has almost the same structure as the one of the basic β -machine. Anyway, 3 more feature are added

- A function to output the first argument;
- A function to output the double of the first argument;
- **Flags**: that is set to 1 if and only if the result is 0;

Some signals govern the ALU behaviour and his interactions with its neighbor component, namely

- **DRALU**: is set to 1 in order to put the output of the ALU on the bus;
- **LDA**: is 1 when the first argument is loaded from the bus;
- **LDB**: is 1 when the second argument is loaded from the bus;
- **ALUFN**: still give the OPCODE of the performed operation by the ALU, such as before;

8.4 The Static RAM.

The static random access memory is place inside the SRAM module illustrated in figure 8.4. It is basically of the same structure that the register file in the β -machine already presented in sub-section 6.1.4. Note that in figure 8.4, the simplify register file is represented but the full β -machine with a bus architecture has still 32 registers available (even if some registers have conventional uses as explain above). There are still some differences with this SRAM architecture and the register file one

- A single register address is given as input of the module;
- A single output is then read a the time and put on the bus;

Again we have several controls signals that dictate the behaviour of the SRAM coponent

- **LDSMAR**: (LoaD Static Memory Address Register) if is set to 1, SMAR stores the value on the bus at the next clock trigger as a register address;
- **LDSRAM**: plays the same role as WERF did previously, it stores the value from the bus into the register whose address is SMAR;
- **DRSRAM**: set to 1, the value in register whose address is SMAR is put on the bus;

8.5 The Dynamic RAM.

The memory module of the β -machine with a bus architecture is illustrate in figure 8.5. It is composed of a random access memory module controlled by several input

- **LDDMAR**: (LoaD Dynamic Memory Address Register) is set to 1 to place the value stored in DMAR on the bus. It is used as the address for DRAM;
- **LDDRAM**: when is set to 1 then the value on the bus (WD) is stored in the RAM at address DMAR;
- **DRDRAM**: set to 1 (LDDRAM is then 0) to put the value in the RAM at address DMAR on the bus (RD);

WD and RD are simply the output of the DRAM module connected to the bus. A shift box (performed a division by 4) is used to transform a byte address into a word address and then obtain a valid one.

8.6 The control unit.

The control unit is the crux in our β -machine with bus architecture study. Indeed, from until now we haven't note big difference between our simple β -machine and this new architecture. Now that we are in a Van Neumann context, data read from memory can be stored to register file, or used as instruction. The control unit simply temporarily stores the current instruction which can be perform in several steps (called *phases*). At each phase, some information is exchanged using the bus, and stored locally where it will be used. Here, signals give the state of the control unit but also dictate the behaviour of the whole β -machine with a bus architecture.

- **LDINSTREG**: is set to 1 in order to put the data on the bus in the register Instr. If this signal is set to 1, the phase value is reset at the next clock trigger;
- **DRLit**, **DRRa**, **DRRb** and **DRRc**: are respectively set to 1 in order to put the constant value, the register address a, the register address b or the register address c;
- **Flags**: this is a signal provided by the ALU as explain before used for BEQ and BNE instructions;

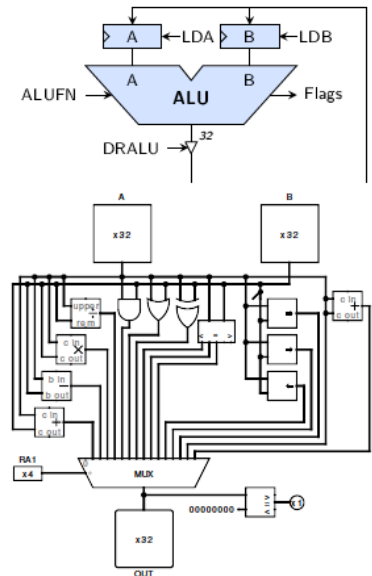


Figure 8.3: ALU of the β -machine with a bus architecture.

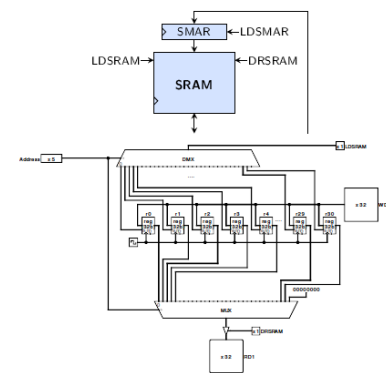


Figure 8.4: SRAM of the β -machine with a bus architecture.

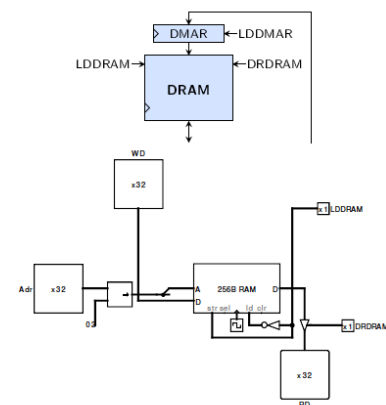


Figure 8.5: DRAM of the β -machine with a bus architecture.

- **Phases:** this signal keep track of the phase performed by the control unit;

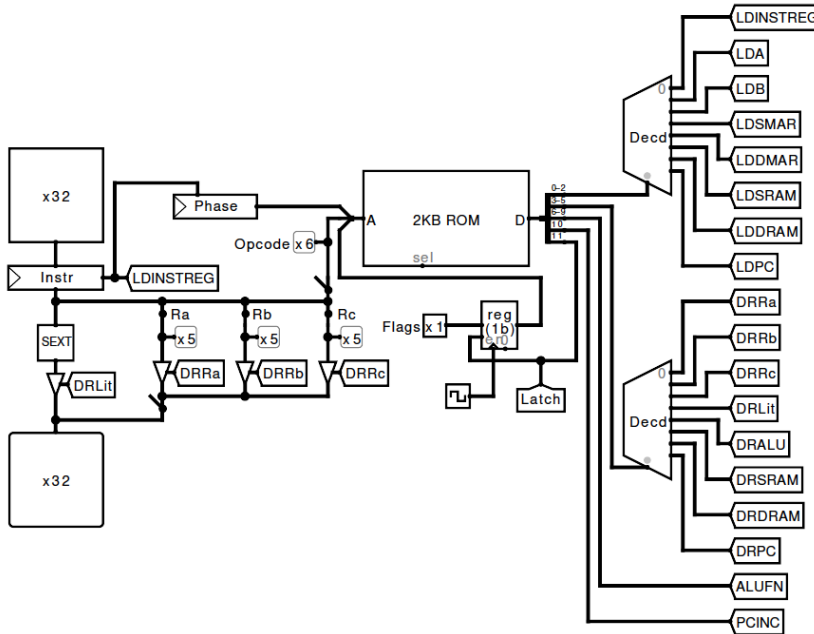


Figure 8.6: Control unit circuit of the β -machine with a bus architecture.

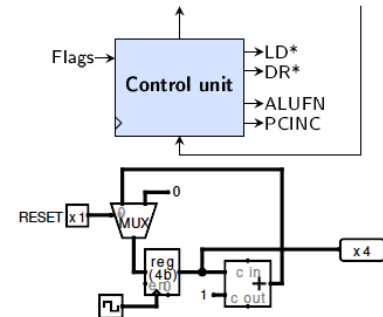


Figure 8.7: Control unit of the β -machine with a bus architecture.

Depending on the Phase, the OPCODE and the value stored in the Flags register, the ROM outputs an 11-bits control signal. It sets one LD* and one DR* bit to true, chooses the ALUFN, set PCINC to 1 or 0 and forces the Flags register to store the register (or not) at the next clock trigger.

Example 8.1: Microcoding the β -machine with a bus architecture.

As we compute instruction in the binary level, we can microcoding this new machine. Now the instructions are no more performed in a single clock cycle but in several phases. We will illustrate this process by microcoding the instruction given in listing 8.1.

ADD(Ra,Rb,Rc): $PC \leftarrow PC + 4$
 $Reg[Rc] \leftarrow Reg[Ra] + Reg[Rb]$

Listing 8.1: Add instruction for microcoding.

We need to perform several cutouts of these operations in order to permit to our new machine to perform them. These cutouts are illustates in listing 8.2

Ra $\xrightarrow{\text{bus}}$ SMAR
 Reg[SMAR] $\xrightarrow{\text{bus}}$ A
 Rb $\xrightarrow{\text{bus}}$ SMAR
 Reg[SMAR] $\xrightarrow{\text{bus}}$ B
 Rc $\xrightarrow{\text{bus}}$ SMAR
 ALU $\xrightarrow{\text{bus}}$ Reg[SMAR]
 PC $\xrightarrow{\text{bus}}$ DMAR
 Mem[DMAR] $\xrightarrow{\text{bus}}$ Instruction
 (and $PC \leftarrow PC + 4$)

Listing 8.2: Cutouts of the add instruction for microcoding.

We have then 8 phases performed by our machine in order to perform the ADD instructions. These instruction are implemented with binary signals inside the control unit as represented in the table above.

ROM address		ROM data				
OPCODE	Phase	LD*	DR*	ALUFN	PCINC	Value
100000	0000	LDSMAR/011	DRRa/000	/ (0000)	0	0000000011
100000	0001	LDA/001	DRSRAM/101	/ (0000)	0	00000101001
100000	0010	LDSMAR/011	DRRb/001	/ (0000)	0	00000001011
100000	0011	LDB/010	DRSRAM/101	/ (0000)	0	00000101010
100000	0100	LDSMAR/011	DRRc/010	/ (0000)	0	00000010011
100000	0101	LDSRAM/101	DRALU/100	ADD (0001)	0	00001100101
100000	0110	LDDMAR/100	DRPC/111	/ (0000)	1	10000111100
100000	0111	LDINSTREG/000	DRDRAM/110	/ (0000)	0	00000110000

Note that the ROM addresses also contain a Flags bit and data contain Latch. But here in this example the action does not depend on Flags, and Latch should always be 0.

SUMMARY

- The **β -machine** with a **bus architecture** is a **Von Neumann** machine.
- The **bidirectional bus**, made of shared interconnection lines, **transmit** between the components.
- In the **bus architecture** there are **two** memory elements, **SRAM** and **DRAM**, instead of 3 for the previous simple **β -machine** (instruction and data memory, and register file).
- The **bus architecture** works in a **Von Neumann** context, therefore **data read from memory** can be stored to **register file**, or **used as instruction**.

9 Pipelining the processor.

For the machines seen until now we have a β -machine that can perform an operation in a single clock cycle, and a bus β -machine performing instructions in several clock cycles, their number depending on the instruction. In both case however the performed instruction is fully finished before the next starts. Here, with a simple solution, we will permit to the β -machine to perform several instructions in a single clock cycle, which means that an instruction has no more need to be completely achieve before an other begins. The solution is the *pipelining* of the processor. This technique attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps performed by different processor units with different parts of instructions processed in parallel. But even if it allows faster CPU throughput than would otherwise be possible at a given clock rate, it may increase latency due to the added overhead of the pipelining process itself.

The figure 9.1 give the longest possible path for an instruction inside the β -machine to be performed. This instruction set the frequency limit given the time of each component to perform their part of the instruction. Here the instruction is obviously a LD instruction, reading in the data memory at address RD1. So even if all instructions use one clock period, it is necessary to leave time between two clock triggers so that the information is computed and reaches the input of registers.

9.1 Choosing the stages.

In the β -machine we have four possible places where applying the pipelining methods. We can then split the logic into 4 stages

1. **Instruction Fetch (IF)**: maintains PC, fetches instruction and pass on;
2. **Register File (RF)**: reads operands from register file, pass on;
3. **ALU**: perform the ALU operation, pass on;
4. **Memory (MEM)/Write back (WB)**: uses ALU result as address for LD, read memory, write result in register file;

9.2 The 2-stages pipeline.

In order to set the context, let's first study a useless pipeline of the β -machine by only consider the If and the EXE (execute). With this method we have no gain of performance at all but we studying how pipelining is indeed done. On figure 9.2 is illustrate how the pipelining is actually done in hardware. We can see in this scheme that registers are added in order to temporarily store the necessary information at each stage of the pipeline and that the stages proceed more or less independently.

REMARK

An analogy here can be made with one of the first application of the pipelining process in the Ford T manufacturing. The assembly chain method reduced the construction time of this car from 12 hours to 1 hour and an half.

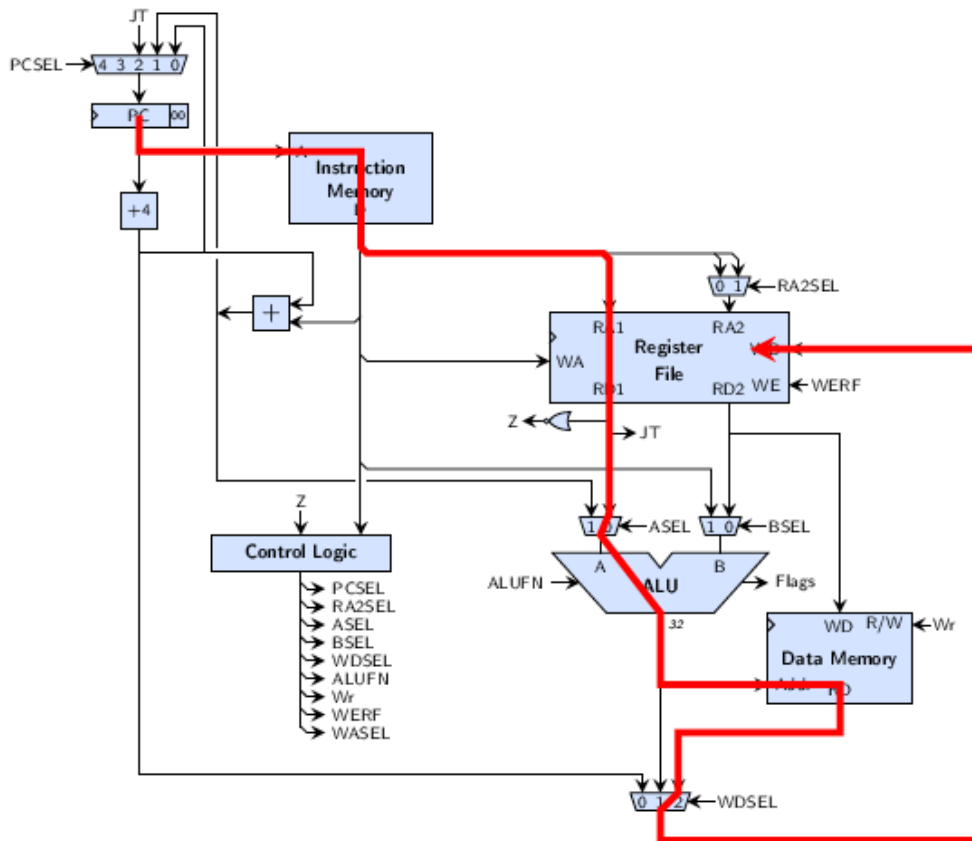


Figure 9.1: Longest possible path for an instruction inside the β -machine (here a read).

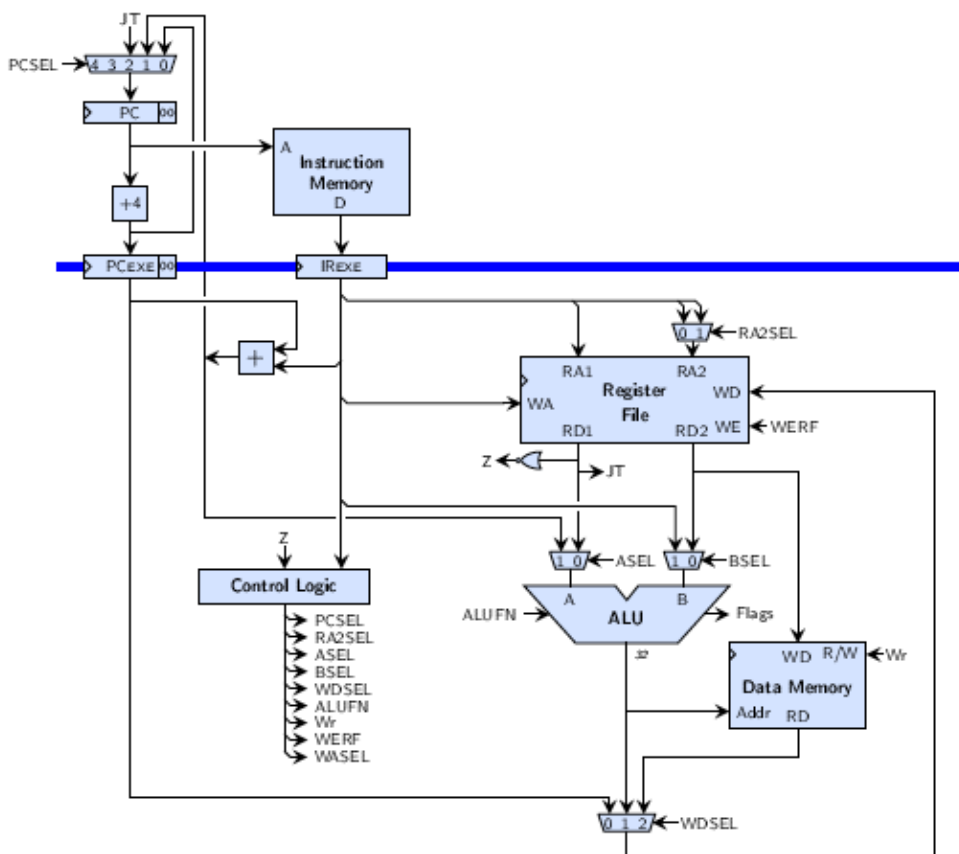


Figure 9.2: Two stages pipelining of the β -machine using the instruction fetch and execute.

Example 9.1: Illustration of two instructions executing simultaneously on the 2-stages pipelined β -machine.

Consider here the sequence of instructions performed inside the β -machine given in listing 9.1.

```

ADDC(r1, 1, r2)
SUBC(r1, 1, r3)
XOR(r1, r5, r1)
MUL(r2, r6, r0)

```

Listing 9.1: Sequence of performing instructions

Instructions at stages IF and EXE would be executed such as in the table above

Time slot	t_1	t_2	t_3	t_4	t_5	...
IF	ADDC	SUBC	XOR	MUL
EXE	...	ADDC	SUBC	XOR	MUL	...

Example 9.2: Illustration of the branching issue for the 2-stages pipelined β -machine.

Consider the sequence of instructions performed in the 2-stages pipelined β -machine given in listing 9.2.

```

loop: ADD(r1, r3, r3)
      CMPLEQ(r3, 100, r0)
      BEQ(r0, loop, r31)
      XOR(r3, r5, r3)

```

Listing 9.2: Sequence of performing instructions with branch

Again we can listed executed instruction by the 2-stages for given slots as in the table above.

Time slot	t_1	t_2	t_3	t_4	...
IF	ADD	CMPLEQ	BEQ	XOR	...
EXE	EXE	ADD	CMPLEQ	BEQ	?

9.2.1 Solving the branch problem of the β -machine by software solution.

Here we can see from example 9.2 that if branching occurs ($r0$ is 0), then the next instruction will not be XOR but ADD. The problem can be solved by a software manner by simply add as much NOP operation that it's required for the loop's instructions execution being performed the right way.

Example 9.2: (Continuation)

We can modify the code in listing 9.2 by adding NOP operation in right place as shown in listing 9.3

```

loop: ADD(r1, r3, r3)
      CMPLEQ(r3, 100, r0)
      BEQ(r0, loop, r31)
      NOP
      XOR(r3, r5, r3)

```

Listing 9.3: Software solution for sequence of performing instructions with branch

And now we have the right behavior of the execution given in the table below.

Time slot	t_1	t_2	t_3	t_4	t_5	...
IF	ADD	CMPLEQ	BEQ	NOP	XOR/ADD	...
EXE	EXE	ADD	CMPLEQ	BEQ	NOP	...

REMARK

A NOP operation is an assembly language instruction, programming language statement, or computer protocol command that does strictly nothing.

9.2.2 Solving the branch problem of the β -machine by hardware solution.

We can also modify the pipelined β -machine directly in its hardware in order to solve the branching execution issue. This hardware solution can be seen in figure 9.3. The idea is to simulate the software solution by adding a NOP control signal to insert a NOP instruction, and freeze the program counter. In order to activate this new signal for all branching instructions, the control logic has now a new output NOPSEL to insert a NOP instruction. Actually, NOPSEL could be only set when the instruction is actually branching, in that case, PCSEL is never 0, and the machine can be simplified.

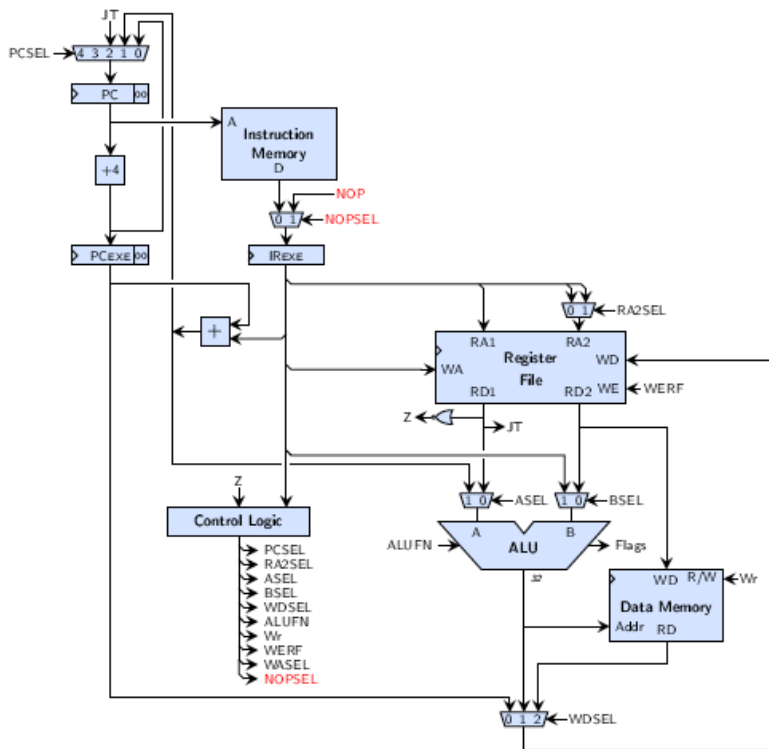


Figure 9.3: Hardware solution for the pipelining of the β -machine in order to perform branching instructions.

9.3 The 4-stages pipeline.

In order to create the 4-stages pipelined β -machine, which stages are given in 9.1, we will perform some hardware modifications on our basic β -machine. First we will duplicate the register file so that it can be used at two different stages (one for reading, the other for writing). Note that even if we duplicate it, it is still the same register file in both of its action stage. Any of the values coming from one another are always the same, these register files are always in exact same states.

Afterward we will make the control logic implicit, to simplify the picture. This means that from now, each stages has a control logic. We then need to add some new registers responsible for separation of each phases (IF, RF, ALU and MEM/WB) of our newly pipelined β -machine illustrate in figure 9.4.

Example 9.3: Illustration of four instructions executing simultaneously on the 4-stages pipelined β -machine.

Consider here the sequence of instructions performed inside the β -machine given in listing 9.4.

```
ADDC(r1, 1, r2)
SUBC(r1, 1, r3)
XOR(r1, r5, r1)
MUL(r2, r6, r0)
```

Listing 9.4: Sequence of performing instructions

Instructions at stages IF, RF, ALU and MEM/WB would be executed such as in the table above

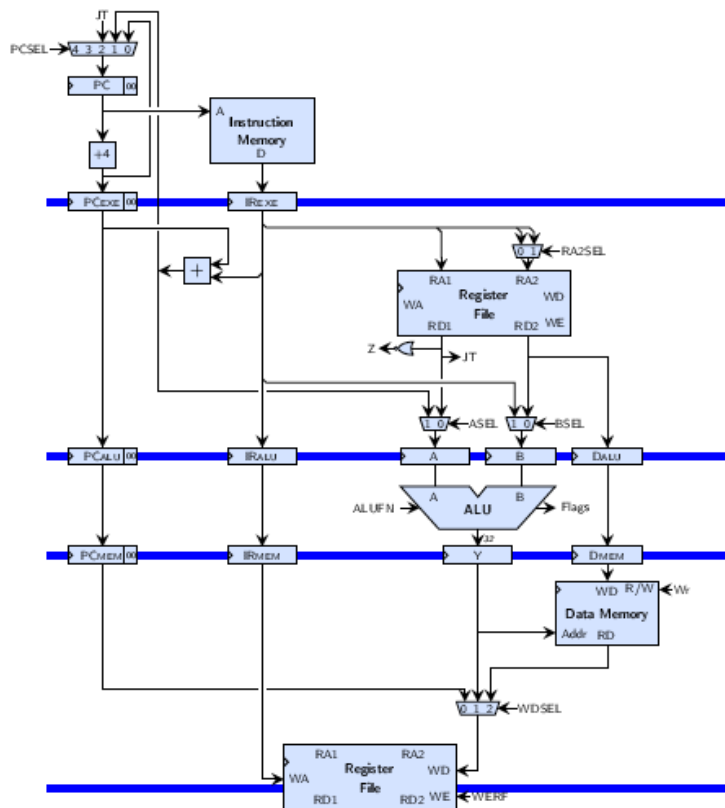


Figure 9.4: Four stages pipelining of the β -machine using the instruction fetch, the register file, the ALU and the memory and write back.

Time slot	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
IF	ADDC	SUBC	XOR	MUL
RF	...	ADDC	SUBC	XOR	MUL
ALU	ADDC	SUBC	XOR	MUL
MEM/WB	ADDC	SUBC	XOR	MUL	...

Example 9.4: Illustration of an execution issue for the 4-stages pipelined β -machine.

Consider the sequence of instructions performed in the 4-stages pipelined β -machine given in listing 9.5.

```

ADD(r1, r2, r3)
CMPLQ(r3, 100, r0)
MULC(r1, 100, r4)
SUB(r1, r2, r5)

```

Listing 9.5: Sequence of performing instructions with branch

Again we can listed executed instruction by the 4-stages for given slots as in the table above.

Time slot	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
IF	ADDC	CMPLQ	MULC	SUB
RF	...	ADDC	CMPLQ	MULC	SUB
ALU	ADDC	CMPLQ	MULC	SUB
MEM/WB	ADDC	CMPLQ	MULC	SUB	...

Here we can see an issue for execution in time slot t_3 . Indeed, from the listing 9.5, we can see that the CMPLQ uses a register that will only be saved an instruction later in the memory by stages MEM/WB.

9.3.1 Solving the 4-stages pipelined β -machine issue by rearranging the program.

A first solution in order to avoid instructions being performed with wrong registers value is to rearrange the execution order of these without changing the correctness of the program.

Example 9.4: (Continuation)

We can simply rearranging the code in listing 9.5 by placing the CMPEQ operation at the end of these executed instructions, as shown in listing 9.6

```
ADD(r1, r2, r3)
MULC(r1, 100, r4)
SUB(r1, r2, r5)
CMPEQ(r3, 100, r0)
```

Listing 9.6: First (software) solution for sequence of performing instructions

And now we have the right behavior of the execution given in the table below.

Time slot	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
IF	ADDC	MULC	SUB	CMPEQ
RF	...	ADDC	MULC	SUB	CMPEQ
ALU	ADDC	MULC	SUB	CMPEQ
MEM/WB	ADDC	MULC	SUB	CMPEQ	...

With the solution presented in example 9.4, we can see that the sequence of instructions is executed in a different order but without alter the purpose of the initial program. Today's compilers take care of the organization of instructions to maximize pipelining.

9.3.2 Solving the 4-stages pipelined β -machine issue by stalling.

An other solution, hardware type, to solve the issue presented in example 9.4 is the introduction of NOP operations, as for the 2-stages β -machine.

Example 9.4: (Continuation)

We can simply add to the code in listing 9.5 some NOP instructions as, as shown in listing 9.7

```
ADD(r1, r2, r3)
NOP
NOP
CMPEQ(r3, 100, r0)
MULC(r1, 100, r4)
SUB(r1, r2, r5)
```

Listing 9.7: Second (hardware) solution for sequence of performing instructions

And now we have the right behavior of the execution given in the table below.

Time slot	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
IF	ADDC	NOP	NOP	CMPEQ	MULC	SUB
RF	...	ADDC	NOP	NOP	CMPEQ	MULC	SUB	...
ALU	ADDC	NOP	NOP	CMPEQ	MULC	...
MEM/WB	ADDC	NOP	NOP	CMPEQ	...

And now we can see that when the CMPEQ instruction arrive in the MEM/WB stages, the right value if r3 is already set.

Anyway, the solution presented in example 9.4 increase the number of cycles needed to perform the given instruction set. This represent a complete waste of cycles and then a waste in matter of performances.

9.3.3 Solving the 4-stages pipelined β -machine issue by bypassing.

Here we present the best solution to the data hazards presented by the example 9.4, the *bypass* option. With this issue, the problem is that information need to be saved in the register file before use, bypassing this step would allow for a quicker treatment. This can be performed simply by adding a pair of bypass multiplexers. These multiplexers sit at the end of the decode stage, and their flopped outputs are the inputs to the ALU. Each multiplexer selects between:

- A register file read port (*i.e.* the output of the decode stage, as in the naive pipeline);
- The current register pipeline of the ALU (to bypass by one stage);
- The current register pipeline of the access stage (which is either a loaded value or a forwarded ALU result, this provides bypassing of two stages);

Add bypasses is basically add a choice at each stage to either take the content of the register from the pipeline, or the result that has not yet been saved if the content is outdated. As we seen earlier in section 3.2, when we introduce a choice in a digital circuit, we use multiplexer. as shown in figure 9.5, we can simply implement the bypass by adding multiplexers in each output of the EXE stage register file, allowing to either chose the register file output or the ALU output. Also we can bypassing the same way ALU's input because a value not already set in the register file can be needed in the ALU stage too.

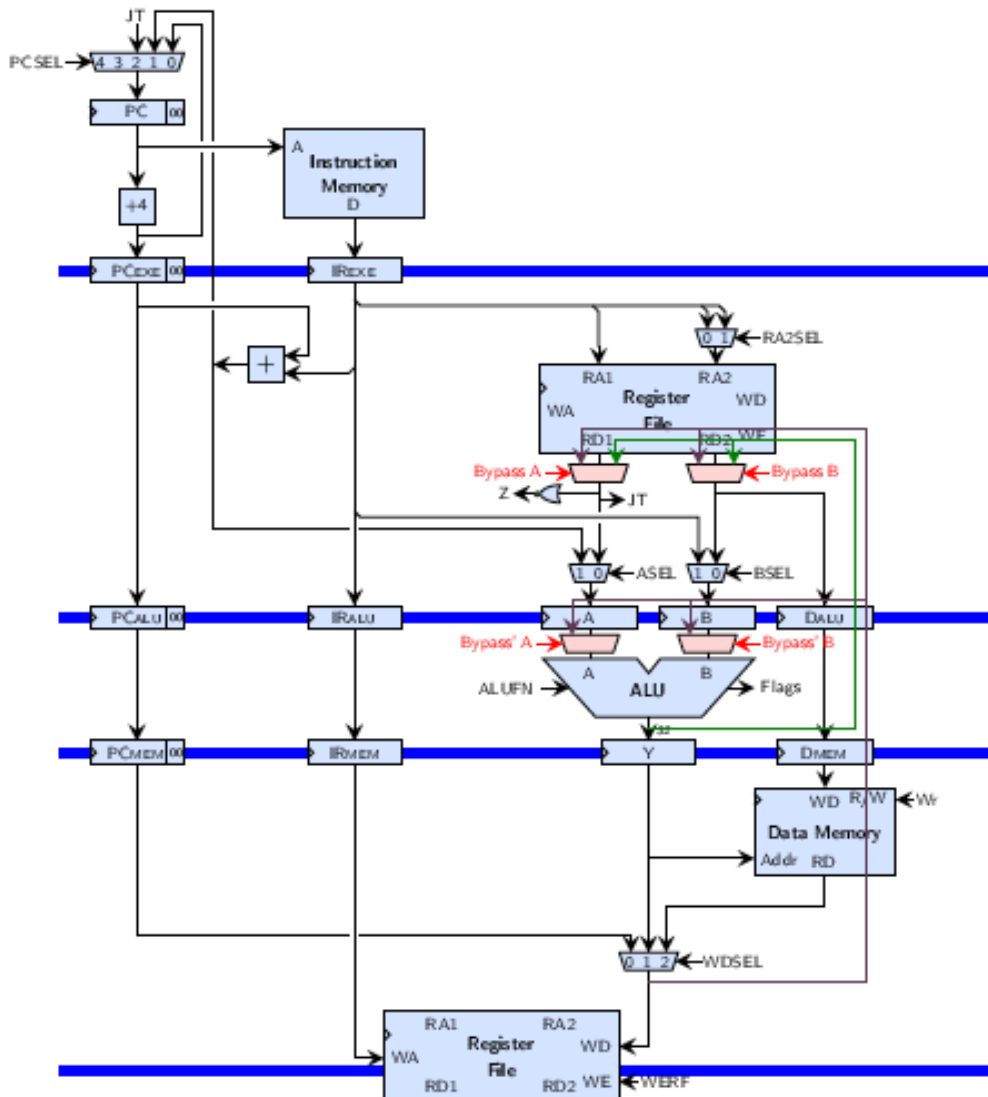


Figure 9.5: Four stages pipelined β -machine with bypass.

SUMMARY

- **Pipelining** is a method used in **central processing units (CPUs)** to allow **overlapping execution** of multiple instructions with the same circuitry.
- In the **β -machine** we have four possible stages
 - ⇒ Instruction Fetch (IF);
 - ⇒ Register File (RF);
 - ⇒ ALU;
 - ⇒ Memory (MEM)/Write back (WB);
- In the **4-stages pipelined β -machine** the **register file** is **repeated** in two different stages (one for reading, the other for writing). Note that even if it seems duplicate, it is still **the exact same** register file in both stages.
- In the **4-stages pipelined β -machine** each stages need its own **control logic**.
- In the **4-stages pipelined β -machine** new registers are need, responsible of the **separation** of each phases.
- **Bypassing** use **bypass multiplexers** in order to **pass data** at a required stage from **another stage**.

10 Memory hierarchy.

In this section we are interested by the memory components of the β -machine. Here we will study theses machine parts, going into details of their characteristics but also presenting new concepts such as the caching. Memory is probably the most important concept inside each machine, such as for the human. Its keep all information about actions performed or that will be performed by the machine. That's naturally that these components have been improved over time, reducing their costs but also making them more efficient.

10.1 Memory technologies.

As it can be seen in table 10.1, each memory element being on the market actually has its own strengths. Depending of its use, a component will be preferred compared to another. For instance if you need a lot of memory but you don't care about the time to access the data inside it, it will be better to choose a HD which is cheaper than the others memory types. Anyway for each use, a memory type will be preferable to others but to choose wisely, the differences between those components need to be know.

	Capacity	Latency	Cost/GB
Register	1000s of bits	20 ps	\$\$\$\$ ³
SRAM	~ 10KB-10 MB	1-10 ns	~ \$1000
DRAM	~ 10GB	80 ns	~ \$10
Flash	~ 100 GB	100 us	~ \$1
Hard disk	~ 1TB	10 ms	~ \$0.10

Table 10.1: Comparative of memory elements by typical capacity, latency and cost/gi-gabyte.

³25 pieces of 8-bits registers cost around \$3 on Banggood, but a gigabyte is 10^9 bytes.

10.2 Static random access memory.

As represented in figure 10.1, a static RAM module is an array of SRAM cell given in figure 10.2. In each of these cells, two CMOS inverters (4 Metal Oxide Semiconductor Field Effect Transistor (MOSFET)) forming a bi-stable element, which is a good way to represent binary information.

10.2.1 SRAM read.

Reading data inside a SRAM module is perform in several steps which are the following

1. Drivers charge all bitlines (vertical lines) to V_{dd} (representing value 1), and leave them floating;
2. Address decoder activates a single wordline (horizontal lines);
3. Each cell in the activated word slowly pulls down one of the bitlines to GND (representing value 0);

4. Sense amplifiers detect change in bitline voltages and produce the output data;

These step are illustrate inside a SRAM cell in figure 10.3.

10.2.2 SRAM write.

Writing data inside a SRAM module is also perform in several steps which are the following

1. Drivers set and hold bitlines to desired values (V_{dd} and GND for 1, GND and V_{dd} for 0);
2. Address decoder activates one wordline;
3. Each cell in word is overpowered by the drivers, stores value;

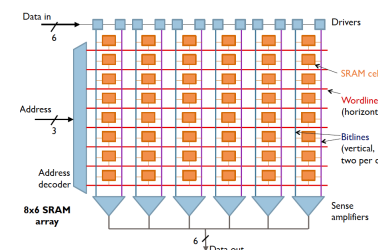


Figure 10.1: 8x6 SRAM module composed of SRAM cells (orange), wordlines (red), bitlines (blue and purple). Addresses are 3-bits long and data are on 6 bits.

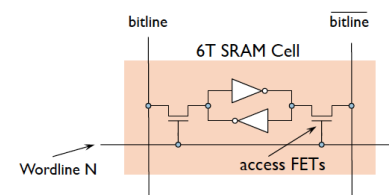


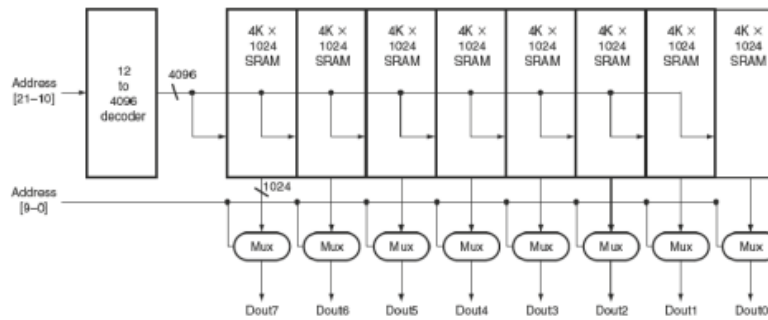
Figure 10.2: 6-MOSFET SRAM cell.

10.2.3 Multiported SRAMs.

SRAM so far can do either one read or one write operation by clock cycle. But with small modifications, we can perform multiple reads and writes with multiple ports by adding one set of wordlines and bitlines per port. This is illustrate in figure 4.8 and for N ports we obtain N wordlines, $2N$ bitlines and $2N$ access FETs.

10.2.4 Scaling SRAM.

As word capacity increases, SRAM needs huge address decoders but building fast decoders with millions of lines is extremely difficult and such decoders would also occupy a large amount of memory chip real estate. A solution to avoid the use of such electronic devices is the 2D organization. This type of organization is show in figure 10.4.



In the diagram in figure 10.4 we can see that from now single signal coming from a multiplexer permit the acces in several SRAM modules. Once signals are stable on MUXes, reading adjacent words is fast, this is a fast sequential access burst.

10.3 Dynamic random access memory.

With the DRAM each bit of data is store in a memory cell consisting of a tiny capacitor and a transistor, both typically based on metal-oxide-semiconductor (MOS) technology. Figure 10.6 show a simple example of a 4x4 DRAM module. A DRAM cell is represented in figure 10.7, as said earlier it is made of a capacitor in series with a transistor.

A DRAM cell is about 20 time smaller than SRAM cell, and who say denser say cheaper. There is however a trouble to notice it's that capacitor leaks charge, and therefore must be refreshed periodically (\sim milliseconds).

10.3.1 DRAM read.

Reading data inside a DRAM module is perform in several steps listed above.

1. Precharge bitline (vertical lines) to $\frac{V_{dd}}{2}$;
2. Activate wordline (horizontal lines);
3. Capacitor and bitline then share charge
 - If capacitor was discharged, bitline voltage decreases slightly;
 - If capacitor was charged, bitline voltage increases slightly;
4. Sense bitline to determine if the bitline voltage variation is 0 or 1
 - Issue: reads are destructive (charge is gone), so, data must be rewritten to cell at end of the operation;

10.3.2 DRAM write.

Writing operation inside a DRAM module is perform simply by driving bitline to V_{dd} or GND , activate a given wordline, and then charge or discharge capacitor in order to represent data.

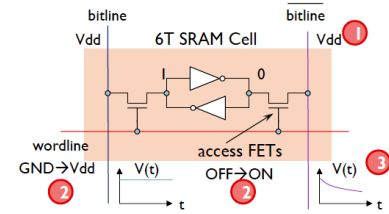


Figure 10.3: View of a read operation in a SRAM cell.

Figure 10.4: 2D-SRAM organization.

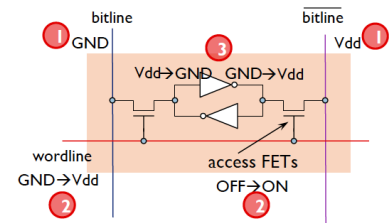


Figure 10.5: View of a write operation in a SRAM cell.

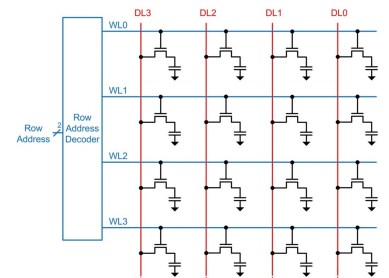


Figure 10.6: Simple 4x4 DRAM array topology.

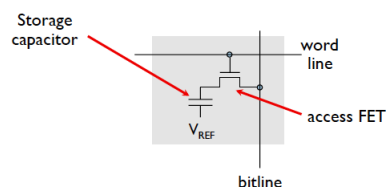


Figure 10.7: DRAM cell.

10.3.3 Scaling DRAM.

Big DRAM modules need many address bits. The row decoder and column multiplexer share the same address pins, which assert that

- **Row Address Strobe (RAS):** address is row address (slow);
- **Column Address Strobe (CAS):** address is column address (fast);

In order to improve the DRAM performances we can add clocks to DRAM (Synchronous DRAM (SDRAM)). This permit to specify the burst with a single address + burst length. Transfers can be also perform on both rising and falling edge of clock (Double Data Rate (DDR) SDRAM). In order to construct large DRAM modules we can also try to perform read and write operations from multiple banks, each with own row buffer as illustrate in figure 10.8. With this configuration, addresses are send to all banks (simultaneous read) and accesses are rotate between banks (address interleaving).

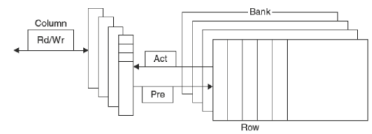


Figure 10.8: Multiple banks DRAM.

10.4 Non-volatile storage.

The non-volatile computer memory, or flash memory is an electronic (solid-state) storage medium that can be electrically erased and reprogrammed. The principle behind flash memory is that it stores information in an array of memory cells made from floating-gate transistors. In single-level cell (SLC) devices, each cell stores only one bit of information while in multi-level cell (MLC) devices, including triple-level cell (TLC) devices, more than one bit can be store per cell.

In flash memory, each memory cell resembles to a standard metal-oxide-semiconductor field-effect transistor (MOSFET), such as in SRAM cells, except that the transistor has two gates instead of one. Electrons in the float gate part diminish strength of field from control gate, thus there is no inversion and N-FET (N-Channel Field Effect Transistor) stays off even when word line is in high voltage.

Flash Memory use "floating gate" transistors to store charge. They are very dense, assigning multiple bits by transistors, performing read and written in blocks. Anyway, compared to the two memory types presented above, flash modules are slow, especially for write operations ($\sim 10\text{-}100\ \mu\text{s}$). Also, the number of writes is limited, meaning that charging/discharging the floating gate (writes) requires large voltages that damage transistor over time.

REMARK

A memory is said **non-volatile** when you can retrieve stored information in it even after having been power cycled. In contrast, it is said **volatile** memory needs constant power in order to retain data.

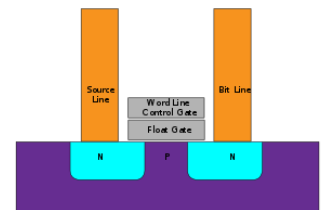


Figure 10.9: Cell structure of flash modules.

10.5 Memory elements classification.

The ideal memory element is large, fast, and cheap. But large memories are slow (even if built with fast components) and fast memories are expensive. The idea in order to get memory efficiency inside computers was to introduce a hierarchical system of memories (illustrate in figure 10.10), with different tradeoffs to emulate a large, fast and cheap memory.

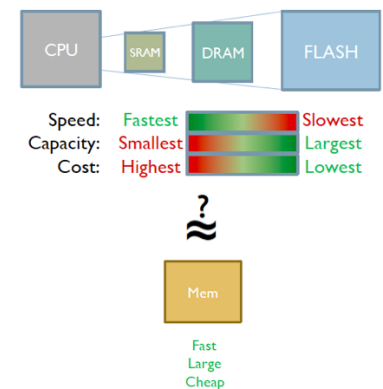


Figure 10.10: Is it possible to find a memory hierarchy which give the ideal memory component?

10.5.1 Approach 1: expose hierarchy.

With the model of hierarchy exposed in figure 10.11 we consider the following memory elements: Registers, SRAM, DRAM, Flash, Hard Disk. Each of them is available as storage alternatives. If this model is adopted, programmers need to know the memory available on the machine and use each type cleverly.

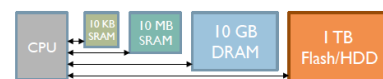


Figure 10.11: Expose hierarchy approach's model.

10.5.2 Approach 2: hide hierarchy.

With the model of hierarchy exposed in figure 10.12 we still consider the same memory component as in section 10.5.1. Each of them is remain available as storage alternatives but here it is a programming model, representing a single memory with single address space. The machine transparently stores data in fast or slow memory, depending on usage patterns.

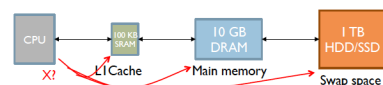


Figure 10.12: Hide hierarchy approach's model.

10.6 The locality principle.

In computer science, locality of reference, also known as the principle of locality, is the most famous principle in memory theory. It represent the tendency of a processor to access the same set of memory locations repetitively over a short period of time. There are two basic types of reference, as always in computer science, temporal and spatial locality. This principle is stated in definition 10.1 and illustrate in figure10.13.

Definition 10.1 - Locality principle

Locality of Reference: access to address X at time t implies that access to address $X + \Delta X$ at time $t + \Delta t$ becomes more probable as ΔX and Δt approach zero.

It is based on this principle that computers keep the most often-used data in a small but fast SRAM (often local to CPU chip). It also refer to main memory only rarely, for remaining data.

10.7 Memory caches.

Memory caches are some element (hardware or software) that stores data so that future requests for that data can be served faster. When their are hardware, these elements are place between the CPU and the memory (see figure 10.14). Because they are faster than the memory itself, if data request by the CPU is in the cache, the information needed is provided quickly. If the cache doesn't contain the requested value then it's the main memory element that serve the CPU. Since the locality principle is generally respected, these element improve substantially the performance of the machine in terms of memory accesses. Computer systems often use multiple levels of caches. Caching is widely applied beyond hardware (*e.g.* web caches: a routers will cache a requested URL in order to serve the same request provided by different hosts).

As shown in figure 10.15, everything in memory component is a cache for something else. Between registers that basically execute a code in the lower computation level and main memory that contain the code variables, we found out several caches

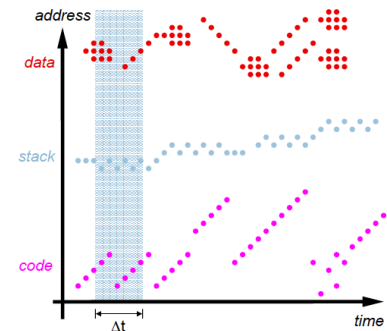


Figure 10.13: Plot of memory addresses accessed by the computer by time. For a fixed time interval (Δt), addresses are close to each other, as the locality principle states it.

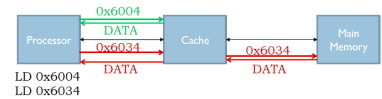


Figure 10.14: Memory caches and example of how it's work in practice. Here there is a **cache hit** in green for data in address 0x6004 and a **cache miss** in red for data in address 0x6034.

	Access time	Capacity	Managed By		Access time	Capacity	Managed By
On the datapath					1 cycle	1 KB	Software/Compiler
Registers	1 cycle	1 KB	Software/Compiler				
Level 1 Cache	2-4 cycles	32 KB	Hardware				
Level 2 Cache	10 cycles	256 KB	Hardware				
On chip							
Level 3 Cache	40 cycles	10 MB	Hardware				
Other chips							
Main Memory	200 cycles	10 GB	Software/OS				
Flash Drive	10-100us	100 GB	Software/OS				
Mechanical devices							
Hard Disk	10ms	1 TB	Software/OS				

Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
TODAY: Hardware Caches		
LATER: Software Caches (Virtual Memory)		

HW vs SW caches:

Same objective: fake large, fast, cheap mem

Conceptually similar

Different implementations (very different tradeoffs!)

When a processor sends address to cache in order to get the data contained inside it, there are two possible options which are

- **Cache hit:** data for this address is in the cache, then it can be returned quickly;
- **Cache miss:** data is not in cache so the cache need to
 - Fetch data from memory, send it back to processor;
 - Update the cache in order to retain this data (replacing some other data);

Therefore, processor must deal with variable memory access time. We can thus define easily 2 cache metrics which are

$$\text{Hit Ratio (HR)} = \frac{\text{hits}}{\text{hits} + \text{misses}} = 1 - \text{MR} \quad (10.1)$$

$$\text{Miss Ratio (MR)} = \frac{\text{misses}}{\text{hits} + \text{misses}} = 1 - \text{HR} \quad (10.2)$$

Also, the average memory access time (AMAT) is given by

$$\text{AMAT} = \text{HitTime} + \text{Miss Ratio} \cdot \text{Miss Penalty} \quad (10.3)$$

It's obvious that the goal of caching is to improve (10.3). This formula can be applied recursively in

multi-level hierarchies, which will give for 2 step

$$\begin{aligned} \text{AMAT} &= \text{HitTime}_{L1} + \text{MissRatio}_{L1} \cdot \text{AMAT}_{L2} \\ &= \text{HitTime}_{L1} + \text{MissRatio}_{L1} \cdot (\text{HitTime}_{L2} + \text{MissRatio}_{L2} \cdot \text{AMAT}_{L3}) \\ &= \dots \end{aligned}$$

The next sections explore the main characteristics of different caches types.

10.7.1 Direct-mapped caches.

In direct-mapped caches, each word in memory maps into a single cache line. In order to access data (for cache with 2^W lines), several steps are performed

1. The address has an index into cache with W address bits (the index bits);
2. Read out the valid bit, tag, and data is performed inside the cache at the index;
3. If the valid bit is 1 and tag matches upper address bits, this is a cache hit;

This type of cache in action is illustrated in figure 10.16, where the information requested by the 32-bit address are read in index $W = 3$ inside the cache. The last bits inside the address field allow to find the right column in caches while the W other bits find the right row. As shown in figure 10.16 and 10.17 the address is composed of $32 - W - A$ bits of tag, where W is the index value and A is used as input of the multiplexer that gives the right output of the cache array (see figure 10.17)

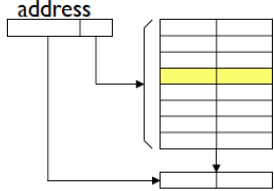
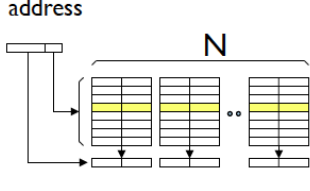
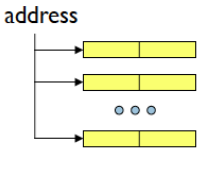
10.7.2 Fully-Associative caches.

Fully-associative caches are the extreme opposite of direct-mapped ones. In this type of caches, any address can be in any location, there is therefore no cache index. This type of cache is more flexible (because there are no conflict misses) but remains more expensive. Indeed, they must compare tags of all entries in parallel to find one matching but this takes space and costs in components (can do this in hardware, this is called a CAM). The structure of such cache elements and their associated address structure are given in figure 10.18.

10.7.3 N-way set associative caches.

This solution is a compromise between direct-mapped and fully associative that use N direct mapped caches placed in parallel (see figure 10.19). The selection of the right data between all these caches is performed such as with a fully associative cache, which map an address to each memory word. In these components, the number of rows is always equal to the number of sets and the number of columns always worth the number of ways. The set size is therefore given by the number of ways, that's why it is called "set associativity" (e.g. 4-way gives 4 entries/set). Inside these components, comparisons between all tags from all ways in parallel are performed in order to retrieve data. Direct-mapped and fully-associative caches are just special cases of N -way set-associative.

10.7.4 Cache types summary.

Direct-mapped	N-way set-associative	Fully-associative
		
<ul style="list-style-type: none"> • Compare addresses with only one tag; • Location A can be stored in exactly one cache line; 	<ul style="list-style-type: none"> • Compare addresses with N tags simultaneously; • Location A can be stored in exactly one set, but in any of the N cache lines belonging to that set; 	<ul style="list-style-type: none"> • Compare addresses with each tag simultaneously; • Location A can be stored in any cache line;

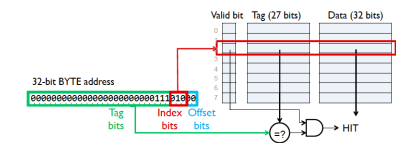


Figure 10.16: Example of a direct-mapped cache for 8-locations, meaning that $W = 3$

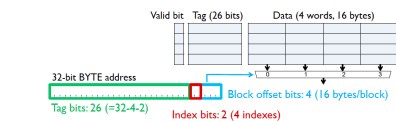


Figure 10.17: Complete example of how data are read inside a cache memory for a 4-block, 16-word direct mapped cache.

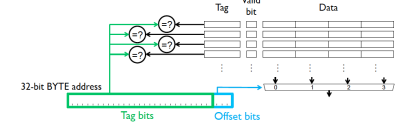


Figure 10.18: Fully-associative caches' structure.

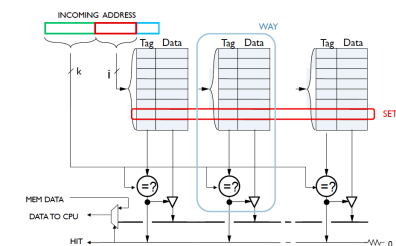


Figure 10.19: N -way associative caches for 3-way and 8-set.

Table 10.2: Summary of caches memory types.

10.7.5 Caches' replacement policies.

Replacing data inside the cache memory can be governed by some replacement policy. The optimal policy (Bélády's algorithm) is to replace the block that is accessed furthest in the future. But this method requires acknowledgement of the future which seems impossible in a deterministic way. The idea will thus be to try to predict the future from looking at the past. If a block has not been used recently, it's often less likely to be accessed in the near future (a locality argument).

Another policy is the replacement of the least recently used (LRU). With this, caches replace the block that was accessed furthest in the past. This works pretty well in practice but caches need to keep ordered list of N items, which represent $N!$ orderings. Thus the complexity in space will be $O(\log(2N!)) = O(N \log(2N))$. Caches also often implement cheaper approximations of LRU. It exists other policies such as the famous FIFO, First-In, First-Out (least recently replaced) or the simpler which is the random one, which choose a replacement candidate randomly. This last is not very good, but does not have adversarial access patterns, it works in practice with good performances and is really simple to implement. With the replacement policy comes the concept of writing policy inside caches. We meet in it different processes

- **Write-through:** the CPU writes are cached but also written to main memory immediately (stalling the CPU until write is completed), memory thus always holds current contents. This method is simple, slow, and wastes bandwidth;
- **Write-behind:** the CPU writes are cached and writes to main memory may be buffered. CPU keeps executing while writes are completed in the background. This method is faster than the previous but still uses lots of bandwidth;
- **Write-back:** the CPU writes are cached, but not written to main memory until we replace the block. Memory contents can be tricked. This method is the fastest, uses low bandwidth but is much more complex. In modern systems, it is the process commonly implemented;

SUMMARY

- A memory is said **volatile** when it needs **constant power** in order to **retain** data, otherwise it is said **non-volatile**.
- The **static random access memory** is a **volatile** memory using **semiconductor** random-access memory (RAM) that uses **bi-stable** latching circuitry to store **binary** data.
- The **dynamic random access memory** is a **volatile** memory using a tiny **capacitor** and a **transistor** to store **binary** data.
- The **non-volatile memory** (flash) is an electronic device that can be electrically **erased** and **reprogrammed**.
- **Large** memories are **slow** and **fast** memories are **expensive** so a **hierarchical system of memories** has been created to emulate a **large, fast** and **cheap** memory element.
- The **locality principle** represent the tendency of memory access by the processor. It is formulated as "If a particular **storage location** is referenced at a particular **time**, then it is likely that **nearby memory locations** will be referenced in the **near future**".
- A **memory cache** stores **data** so that **future requests** for that data can be **served faster**.
- A **cache hit** occurs when a requested data is **found** in the cache, **otherwise** we face a **cache miss**. The occurency ratio of these two are given by

$$\text{Hit Ratio (HR)} = \frac{\text{hits}}{\text{hits} + \text{misses}} = 1 - \text{MRMiss Ratio (MR)} = \frac{\text{misses}}{\text{hits} + \text{misses}} = 1 - \text{HR}$$

- ⇒ **Direct-mapped caches:** compare addresses with only **one tag**. Location **A** can be **stored** in exactly **one** cache line;
- ⇒ **N-way set-associative caches:** compare addresses with **N** tags simultaneously. Location **A** can be stored in exactly **one set**, but in any of the **N** cache lines belonging to **that set**;
- ⇒ **Fully-associative caches:** compare addresses with **each tag** simultaneously. Location **A** can be stored in **any** cache line;
- We can found out several caches' replacement policies
 - ⇒ **Bélády's algorithm:** replace the block that is accessed furthest in the future;
 - ⇒ **FIFO:** least recently accessed data is replaced;
 - ⇒ **Random:** pick up data randomly and replace. This approach is the easiest and give pretty good results;

11 Thread-level parallelism.

Thread-level parallelism (TLP) is the parallelism inherent in an application that runs multiple threads at once. Note that a thread of execution here is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the

operating system. This type of parallelism is found largely in applications written for commercial servers such as databases. By running many threads at once, these applications are able to tolerate the high amounts of I/O and memory system latency their workloads can incur. While one thread is delayed waiting for a memory or disk access, other threads can do useful work.

The instruction-level parallelism (ILP) has its limits inside processor. Deeper pipelines have more instructions "in-flight" which induce more opportunities for dependencies. Sometimes the processor speculates on several branches simultaneously and it generates more opportunities to get things wrong but also to face more wastage when the speculation results to bad guesses. Anyway, with some useful mechanisms we can go past these ILP limitations.

11.1 Hyper-threading.

So far we only considered a single program in execution on the CPU but also in the machine. A big part of the operating system's job is to support execution of several programs simultaneously, given to each program part of the RAM and keeping a copy of the "state" of the program execution. The second part of this job consist to save contents of all CPU registers (copied from registers when program "swapped out" of CPU, restored these saved values into registers when program "swapped in"). This require a logic to deal with unretired instructions. A memory and a state design a thread of execution and several threads of execution can exist at the same time.

Up to now in this courses, CPU was designed to execute a single thread at a time. With *hyper-threading*, we want to allow the CPU to support and execute several threads at the same time in order to avoids constant thread "swapping", these being expensive in terms of efficiency. In order to implement hyper-threading, we will simply duplicate architecture state. This means duplicate duplicate registers, which implies register file, PC, pipeline registers, reservation stations, re-order buffer, exception logic,... Note that the data path (*i.e.* all the electronic functional units, such as the ALU) will not be duplicated.

11.1.1 Architectural state.

As said earlier, registers will be duplicate inside the processor. Several architectural states will implies virtual CPU that will be identify each internally by extra bit, as shown in figure 11.1.

The principle is first to start fetching from other thread while predicting branch/stalling in this same thread. After what we retire instructions from other thread when the head of the re-order buffer (ROB) for this thread is not read. Note that we switch thread regularly too.

With hyper-threading, the control logic is also duplicated. The branch prediction logic is often shared between hyper-threads but this is source of security issues, namely because a thread can "steer" where other thread will speculate.

11.2 Multi-core CPUs.

In todays computer, CPU are composed of several processing units, called cores, each of which reads and executes program instructions, as if the computer had several processors. This conception responds to the performance race: for today's applications, we need faster CPU. As shown earlier, the clock rate cannot increase indefinitely because it's limited by components characteristics. The rate difference between CPU clock and memory access rate increases and the greater the clock rate is, the smaller the clock period will be. With this comes the size of die concept (*e.g.* size of "wires" on chip) that becomes (much) bigger than clock period and thus several signals come on the wires at the same time. A processor also face power dissipation, given by (11.1).

$$P \propto V^2 f \quad (11.1)$$

where V is the CPU voltage and f is the CPU frequency (*i.e.* the CPU's clock rate). Because the clock rate cannot be increased after a certain level, the replication of CPU itself seems to be a good way to get more computing power. Figure 11.2 and 11.3 illustrates this duplication in order to implement a quad-core CPU from a theoretical point of view and a practical one respectively.

For multi-core CPUs, memory elements are physically distributed, but controllers cooperate to give an impression of forming single memory space. The memory latency depends on where the request is from and where the data is stored. This is so called a Non Uniform Memory Access (NUMA).

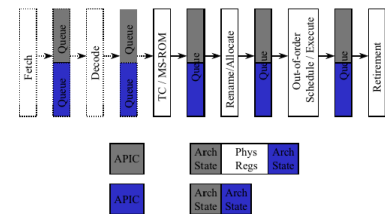


Figure 11.1: Representation of virtual CPU for architectural states and their bit identifiers. The value 0 is in blue while 1 is red/grey.

REMARK

A re-order buffer (ROB) is used for manage out-of-order instruction execution, it allows instructions to be committed in-order. Normally, there are three stages of instructions: "Issue", "Execute", "Write Result".

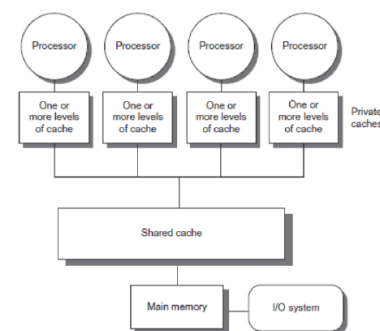


Figure 11.2: Processor duplication for a quad-core CPU for a theoretical view.

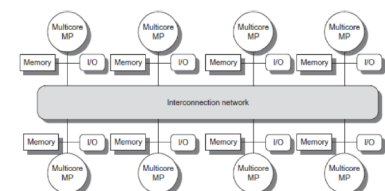


Figure 11.3: Processor duplication for a quad-core CPU for a practical view.

11.2.1 Trouble with multi-core CPUs.

A big problem that multi-core CPUs need to face is the cache coherence issue. A shared resource data can end up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data as illustrate in example 11.1.

Example 11.1: Illustration of caches incoherent storage for shared memory resources.

Consider here that 2 cores can read inside a shared memory components but that each of them has its own cache memory.

Time slot	Event	Processor A cache's content	Processor B cache's content	Location X memory content
t_0				1
t_1	Processor A reads X	1		1
t_2	Processor B reads X		1	1
t_3	Processor A store 0 at X	0	1	0

We can see from the execution illustrate in the table above that caches can contain erroneous value of memory content.

We speak about cache coherence when all reads by any core must return the most recently written value and writes to the same location by any cores are seen in the same order by all cores. We designed by consistency the moment when a written value will be returned by a read. This means that if a core writes location A , then location B , any core that sees new value of B must also see new value of A . These show that we need protocol between private caches.

11.2.2 Cache coherence protocol.

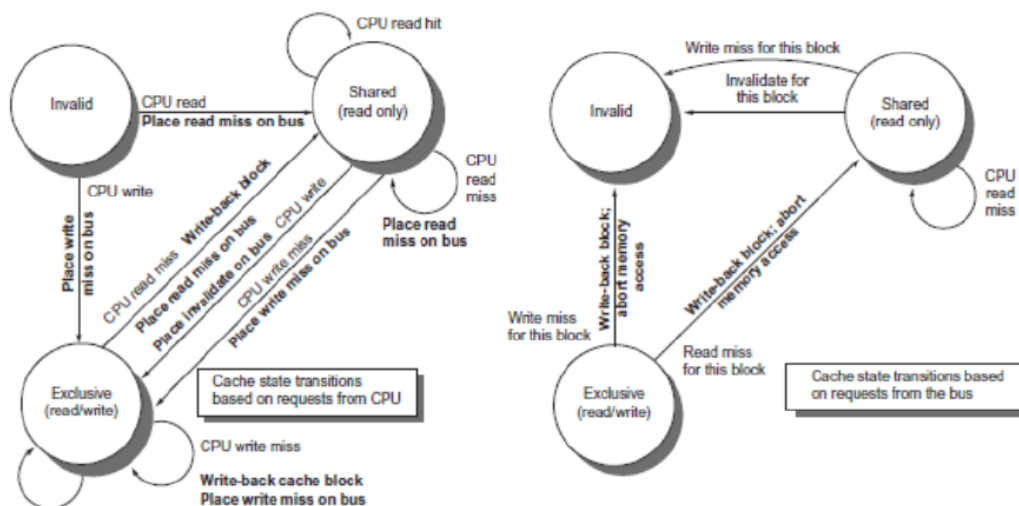


Figure 11.4: Finite state machine of the cache coherence protocol.

In figure 11.4 can be seen the finite state machine of the cache coherence protocol with transition based on cache's CPU actions for the left side or based on other CPUs' from the bus actions for the right side. This protocol solve the previously studied caching issue. A practical example of this protocol execution is provided in example 11.2.

Example 11.2: Illustration of caches coherent protocol between 2 CPUs.

- CPU 1 starts in invalid state, places read miss on the bus, reads block in location X and then goes to shared state;

- CPU 1 re-reads block X because of the previous read miss and these are this time read hits;
- CPU 2 reads blocks X, places read miss on the bus, reads block X and also goes to shared state;
- CPU A writes block X, always place write miss on the bus and then move towards the exclusive state of the FSM;
 - CPU 2 using right side FSM this time will read write-miss and thus move to the invalid state;
- CPU 1 writes or reads block X but remains in the exclusive state this time;
- CPU 2 reads block X, places read miss in the bus;
 - CPU 1 using now the right side of the figure 11.4 gets read miss and moves to shared mode, supplies correct memory block to CPU 2; CPU 2 finally move to shared mode;

SUMMARY

- **Thread-level parallelism (TLP)** is the **parallelism** inherent in an application that runs **multiple threads** at once.
- Big parts of the **operating system's** job are to support execution of several programs simultaneously **providing** them **RAM** and copying their **states** and **saving** contents of all **CPU registers**.
- **Hyper-threading** allows the CPU to **support** and **execute several threads** at the **same time**.
- Nowadays **CPU** are composed of several **processing units**, each of which **reads** and **executes program** instructions, as if the computer had several processors.
- Because the **clock rate** cannot be increase after a **certain level** (because of **race conditions** appearances), the **replication** of **CPU** itself is a good way to get **more computing power**.
- A **cache coherence** issue appears when **erroneous data** are still stored in a **private processor cache** but has been **updated** by another **processor**.
- The **cache coherence protocol** solve the **cache coherence** issue.

12 Data-level parallelism.

Data parallelism is a form of parallel computing for multiple processors using a technique for distributing the data across different parallel processor nodes. It contrasts to thread or instruction parallelism as another form of parallelisms. In a multiprocessor system where each one is executing a single set of instructions, data parallelism is achieved when each processor performs the same task on different pieces of distributed data. In some situations, a single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code.

12.1 The 4 computer architecture classes.

We can designated 4 big classes of computer architecture from now thanks to the Flynn's taxonomy which is a classification system, they are

1. **Single Instruction Stream, Single Data stream (SISD):** these family is composed of basic uni-processors (*e.g.* the Beta). These are sequential computers which exploit no parallelism in either the instruction or data streams. Single control unit (CU) fetches single instruction stream (IS) from memory. The CU then generates appropriate control signals to direct single processing element (PE) to operate on single data stream (DS) (*i.e.* one operation at a time);
2. **Multiple Instruction Streams, Multiple Data streams (MIMD):** multiple autonomous processors simultaneously executing different instructions on different data. In this type of architecture we found out multicore processors such as presented in section 11;
3. **Multiple Instruction Streams, Single Data stream (MISD):** multiple instructions operate on one data stream. This is an uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result (*e.g.* the Space Shuttle flight control computer).
4. **Single Instruction Stream, Multiple Data streams (SIMD):** a single instruction operates on multiple different data streams. Instructions can be executed sequentially, such as by pipelining, or in parallel by multiple functional units. This implement data-level parallelism and is the most used for today's architectures;

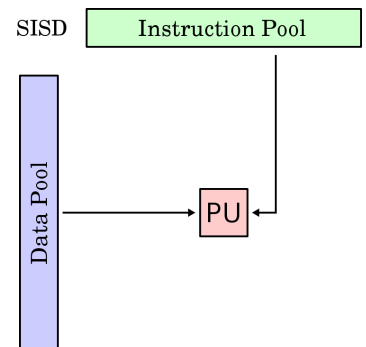


Figure 12.1: Single Instruction Stream, Single Data stream (SISD) model from the Flynn's taxonomy.

The classification system has stuck, and has been used as a tool in design of modern processors and their functionalities. Illustrations of the different Flynn models can be seen from figures 12.1-12.5.

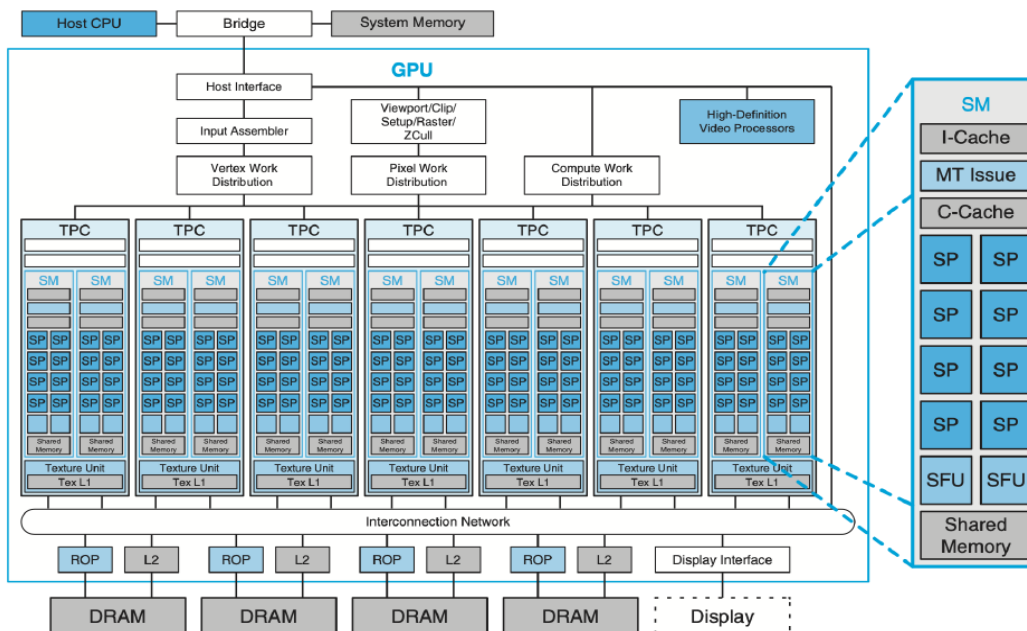
12.2 GPU architecture.

Inside modern computers, a graphics processing unit (GPU) is a specialized electronic circuit designed to quickly manipulate and alter memory in order to accelerate the creation of images in a frame buffer intended for output to a display device. Their highly parallel structure makes them more efficient than central processing units (CPUs) for algorithms that process large blocks of data in parallel. In a personal computer, a GPU can be present on a video card or embedded on the motherboard. Anyway, inside modern systems this is mandatory to have both CPU and GPU, you will never find a system with only a GPU running.

These structure are then highly optimized for massive data parallelism, meaning that they are efficient for repeating the "same" thing on many different data (*e.g.* encryption, scientific computing, image processing). It exists several programming frameworks in order to implement GPU such as the compute unified device architecture (CUDA) created by Nvidia or the open computing language (OpenCL) originally provided by Apple.

GPU are classified in an other Flynn's families different from the 4 fundamentals which is the Single Program, Multiple Data streams (SPMD) architecture class. These type of architectures exploit multiple autonomous processors simultaneously executing the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data. Also termed single process, multiple data - the use of this terminology for SPMD is technically incorrect, as SPMD is a parallel execution model and assumes multiple cooperating processors executing a program. Here, programs are written as single thread, but run as multiple threads and these threads are allowed to diverge in execution path.

A typical GPU architecture scheme is provided in figure 12.4. We can see that GPUs are composed of hundreds (even thousands) of streaming processor (SP) cores, organised as multiple (tens/hundreds) multi-threaded streaming multiprocessors (SM). They also possess banked memory organization for high bandwidth (multiple simultaneous accesses).



All cores in an streaming multiprocessors execute the exact same instruction simultaneously. Thus this require a single controller but a shared instruction cache, a shared constant cache, multiple issue units and a shared memory either for thread local data but also for thread communication.

Inside streaming processors, an highly multi-threaded policy is implemented. We found out

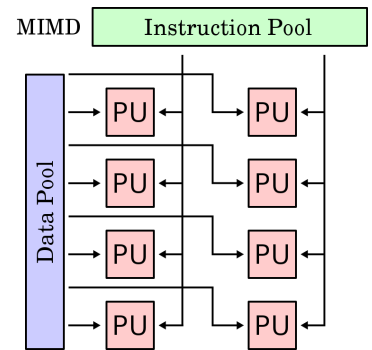


Figure 12.2: Multiple Instruction Streams, Multiple Data streams (MIMD) model from the Flynn's taxonomy.

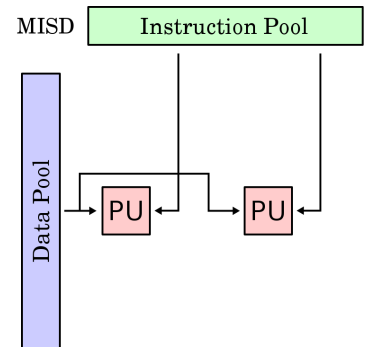


Figure 12.3: Multiple Instruction Streams, Single Data stream (MISD) model from the Flynn's taxonomy.

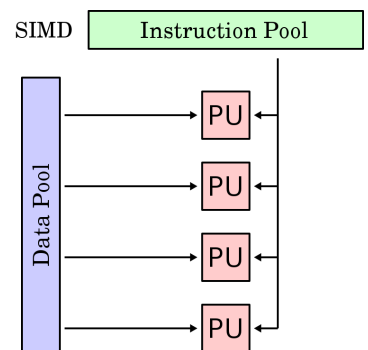


Figure 12.5: Single Instruction Stream, Multiple Data streams (SIMD) model from the Flynn's taxonomy.

multiple (duplicated) pipelines with many hyperthreads per pipeline and each core is very simple for computing, they do not perform speculation or other high level mechanisms. Inside processors we also find a huge register file which contain thousands of registers in multi-port SRAM-based register file. These registers are allocated dynamically by the compiler to cores. The SFU is a specialized unit, either for graphics or machine learning operations. ROP and texture units are graphics operations specialization.

12.3 Predication.

Inside a GPU architecture, all cores must execute the same instruction, but different threads may take different sides of branches. For instance, a core can enter inside an `if` statement with a longer body of instruction than the `else` statement executed by a neighbour core. That's why inside GPUs, branch execution are based on predication. This is an architectural feature that provides an alternative to conditional transfer of control, implemented by machine instructions such as conditional branch, conditional call, conditional return, and branch tables. Predication works by executing instructions from both paths of the branch and only permitting those instructions from the taken path to modify architectural state. The example 12.1 show how to translate a simple branching code in its predication style equivalent.

Example 12.1: Illustration of caches coherent protocol between 2 CPUs.

Imagine the code given in 12.1

```
if (cond)
    instr1;
else
    instr2;
```

Listing 12.1: Simple branching statement for a GPU.

This can be rewritten in a prediction style in order for each core to perform the same exact operation as described in listing 12.2.

```
[cond] instr1;
[!cond] instr2;
```

Listing 12.2: Simple branching statement for a GPU in predication style.

Inside listing 12.2, `[cond]` is a predicate used by core, if this predicate is true, core commit results and if it's false the core do not commit (equivalent of performing a NOP).

Inside GNUs, each side of a branch is executed sequentially which can lead to loss of efficiency because not every core is doing useful work all the time. Also, nested branches require (in hardware) per-core branch synchronization stack for management of predicates.

12.4 CUDA execution model.

The Compute Unified Device Architecture (CUDA) platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. This platform is designed to work with programming languages such as C, C++, and FORTRAN. The execution of CUDA's software is perform following a certain model, but first let's introduce some vocabulary.

- **Kernel:** a program or function for one thread, executed by many hardware threads;
- **Thread block:** A set of concurrent threads that execute the same program and may cooperate to compute result;
- **Grid:** A set of thread blocks that execute the same kernel program;
- **Warp:** the set of parallel threads that execute the same instruction together in a Single Instruction, Multiple Thread (SIMT) architecture. Thread blocks can contain one or several warps;

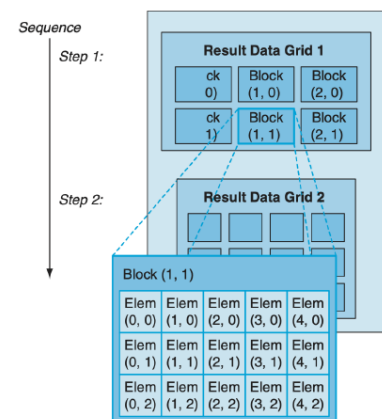


Figure 12.6: Example of data partition for CUDA execution model.

The trick for CUDA execution is to partition the data and assign each partition to different threads, that's why a warp scheduler is needed to transparently schedules ready warps (see figure 12.7). A ready warp is basically a warp where each thread has all the data necessary to execute next instruction. Warp scheduling hides memory latency inside the core. Remember here that this is all done per SM and we have multiple independent SMs, so we can run several different kernels simultaneously (on different SMs).

12.5 Memory issue.

It is clear that we can have thousands of threads trying to access memory at the same time inside GPUs, but the main RAM is not designed for that. A cheaper solution (in terms of cycles) is to transfer big block of data from CPU memory to GPU memory and have thread realize concurrent access on GPU memory. Then to transfer results back to CPU memory after what it can already start next transfer while current block is being processed on GPU. Threads should then access CPU memory directly in order to treat data efficiently.

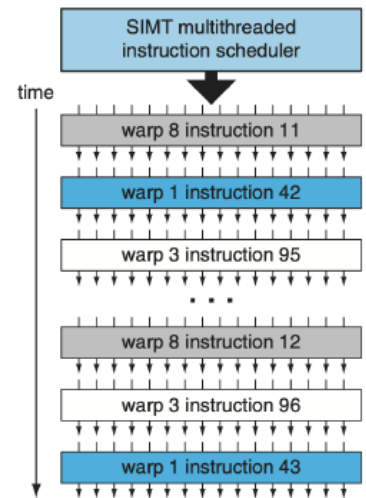


Figure 12.7: Example of warp scheduling for CUDA execution model.

SUMMARY

- **Data parallelism** is a form of **parallel computing** for multiple **CPU** using a technique for **distributing** the **data** across different **parallel processor** nodes.
- The **Flynn's taxonomy** classified systems in several classes, the most famous are
 - **Single Instruction Stream, Single Data stream (SISD)**: uni-processor;
 - **Multiple Instruction Streams, Multiple Data streams (MIMD)**: multiple autonomous processors;
 - **Multiple Instruction Streams, Single Data stream (MISD)**: multiple instructions operate on one data stream;
 - **Single Instruction Stream, Multiple Data streams (SIMD)**: single instruction operates on multiple different data streams;
- A **graphics processing unit (GPU)** has a particular **design** to quickly **manipulate** and **alter memory** to acceleration **image** processing. Their **highly parallel** structure makes them more efficient for **computing** algorithms that process **large blocks of data** in **parallel**.
- **GPU** are **classified** in the **Single Program, Multiple Data streams (SPMD)** family as defined by the **Flynn's taxonomy**, exploiting **multiple autonomous processors** simultaneously **executing** the **same** program on **different data**.
- The **prediction** is a technique where **all branches** instruction are **read** by the program but only **executed** if a the **predicate** relative the the **instruction** is **respected**. This technique **allows** the GPU thread to **run** the **exact same instruction** at any **time**.
- The **Compute Unified Device Architecture (CUDA)** platform is a **software** layer that gives **direct access** to the **GPU's** virtual instruction set and **parallel** computational **elements**, for the execution of compute kernels.

13 Instruction-level parallelism.

The instruction-level parallelism (ILP) is a measure of how many instructions in a computer program can be executed simultaneously. ILP must not be confused with concurrency. Instruction-level parallelism is about parallel execution of a sequence of instructions belonging to a specific thread of execution of a process (that is a running program with its set of resources, *e.g.* its address space, a set of registers, its identifiers, its state, program counter, and more). Conversely, concurrency regards with the threads of one or different processes being assigned to a CPU's core in a strict alternation or in true parallelism if there are enough CPU's cores, ideally one core for each runnable thread.

There are two approaches to instruction level parallelism: Hardware and Software. Hardware level works upon dynamic parallelism (*i.e.* the processor decides at run time which instructions to execute in parallel), whereas the software level works on static parallelism (*i.e.* the compiler decides which instructions to execute in parallel). We have already meet a form of ILP with the pipelining of the β -machine

13.1 Pipelining.

The main idea of pipelining is to overlaps execution of several instructions such as already presented in section 9. With these models, each stage requires shorter clock period (than for a non-pipelined architecture). In an ideal steady-state performance, 1 instruction is completed every clock cycle.

Anyway with pipelining hazards are a problem and we denote several types of them

- **Structural hazards**: not enough resources are available;
- **Data hazards**: an instruction is still in pipeline and produces result for later instruction already in pipeline. These hazards are mostly solved by forwarding, but load followed by use always

triggers stall;

- **Control hazards:** generate by branches (and some indirect jumps). A solution is to try to guess what comes next;

ILP can be easily improve namely by building deeper pipelines in order to allow more instructions to overlap, but increasing hazard opportunities. Another improvement would be to improve the dynamic branch predictions. We could also allow out-of-order execution, because the program correctness depends on data and control flow preservation but the order of independent instructions can be changed. A final solution could be to replicate internal components of pipeline but this could introduce multiple issues, because we physically execute multiple instructions at the same time.

13.2 Control hazards.

Control hazard occurs when the pipeline makes wrong decisions on branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded. The term branch hazard also refers to a control hazard.

To avoid control hazards microarchitectures can insert a pipeline bubble (*i.e.* a method to preclude data, structural, and branch hazards - as instructions are fetched, control logic determines whether a hazard could/will occur), guaranteed to increase latency, or use branch prediction and essentially make educated guesses about which instructions to insert, in which case a pipeline bubble will only be needed in the case of an incorrect prediction. If a branch causes a pipeline bubble after incorrect instructions have entered the pipeline, care must be taken to prevent any of the wrongly-loaded instructions from having any effect on the processor state excluding energy wasted processing them before they were discovered to be loaded incorrectly.

We have seen static solution to deal with control hazards, always assume branch instructions as not taken. On wrong guess, unroll execution back to guess point, and start again with correct guess. Static guesses are always arbitrary, and thus often wrong. Dynamic guesses use dynamic branch prediction, which means, as already explain above, that prediction of branches are made at running time using running time information. Components for these method can easily be integrated to instruction fetch stage.

13.2.1 Dealing with control hazards: branch prediction.

Two-way branchings are usually implemented with a conditional JUMP instruction. A conditional JUMP can either be "not taken" and continue execution with the first branch of code which follows immediately after the conditional JUMP (when condition is not encountered), or it can be "taken" and jump to a different place in program memory where the second branch of code is stored (when condition is encountered). It is not known for certain whether a conditional JUMP will be taken or not taken until the condition has been calculated and the conditional jump has passed the execution stage in the instruction pipeline.

Without branch prediction, the processor would have to wait until the conditional jump instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline. The branch predictor attempts to avoid this waste of time by trying to guess whether the conditional JUMP is most likely to be *taken* or *not taken*. The branch that is guessed to be the most likely is then fetched and speculatively executed. If it is later detected that the guess was wrong then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay.

13.2.2 2-bits saturating counter.

A 2-bits saturating counter is a finite state machine of 4 states: strongly not taken, weakly not taken, weakly taken and strongly taken. When a branch instruction is evaluated, the corresponding state machine is updated. Branches often evaluated as not taken change the state toward strongly not taken, and branches often evaluated as taken change the state toward strongly taken. The advantage of the two-bit counter scheme over a one-bit scheme is that a conditional JUMP has to deviate twice from what it has done most in the past before the prediction changes. The finite state machine of this

REMARK

The typical 5-stages β -machine presented in section 9 is a very idealized version of pipelining, namely because more realistic pipelines must deal with varying instruction latencies.

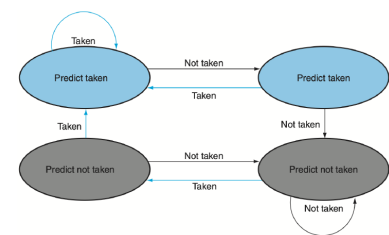


Figure 13.1: State diagram of 2-bit saturating counter.

branch predictor is provided in figure 13.1.

13.2.3 Branch prediction buffer.

Imagine an `if` statement executed three times by a pipelined machine. The decision made on the third execution might depend upon whether the previous two were taken or not. In such scenarios, adaptive predictors remember the history of the last k occurrences of the branch and uses one saturating counter for each of the possible 2^k history patterns. Inside branch prediction buffer, or branch history table such as the one represented in figure 13.2, we found out branch instruction address indexed by lower bits where each of it is associate to a 1-bit predictor (1 is Taken, 0 is Not taken). If the prediction made by reading this table is wrong, the process roll back, flip the prediction and fetch proper sequence.

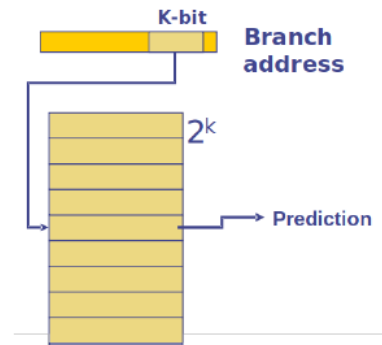
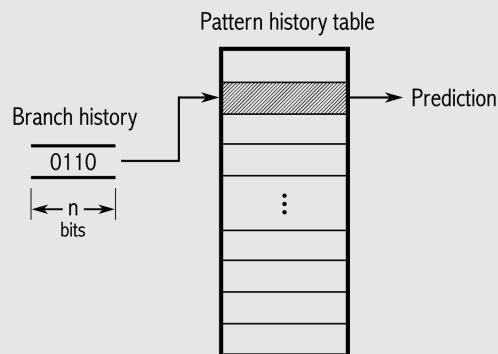


Figure 13.2: Branch history table or branch prediction buffer.

Example 13.1: Two-level adaptive predictor example of execution.

Consider the example of $k = n = 2$. This means that the last two occurrences of the branch are stored in a two-bit shift register. This branch history register can have $2^{k=n=2} = 4$ different binary values, 00, 01, 10, and 11, where zero means "not taken" and one means "taken". A pattern history table contains four entries per branch, one for each of the $2^2 = 4$ possible branch histories, and each entry in the table contains a two-bit saturating counter of the same type as in figure 13.1 for each branch. The branch history register is used for choosing which of the four saturating counters to use. If the history is 00, then the first counter is used. If the history is 11, then the last of the four counters is used.



Assume, for example, that a conditional JUMP is taken every third time. The branch sequence is 001001001... In this case, entry number 00 in the pattern history table will go to state "strongly taken", indicating that after two zeroes comes a one. Entry number 01 will go to state "strongly not taken", indicating that after 01 comes a zero. The same is the case with entry number 10, while entry number 11 is never used because there are never two consecutive ones.

13.2.4 Indirect branch.

Address to jump to for a JUMP instruction is contain in registers, or requires computation (1 cycle). This may be also loaded from memory (*e.g.* from stack for RTN processes) and this is really hard to predict. There are many callers for one single callee or jumps can be performed to multiple address from single address. A branch target buffer (BTB) associated with local hardware stack (for a better support function calls/return) can handle such type of prediction. BTB is almost the same principle as branch predictor or cache. It predicts the target of a taken conditional branch or an unconditional branch instruction before the target of the branch instruction is computed by the execution unit of the processor. BTB thus differs from branch predictor which attempts to guess whether a conditional branch will be taken or not-taken (*i.e.* , binary).

13.3 Hardware based speculations.

Hardware based speculations (HBS) are based on three key concepts

1. **Dynamic Branch prediction:** to choose which instruction to execute;

2. **Speculation:** to allow the execution of instructions before the control dependencies are resolved;

3. **Dynamic scheduling:** to deal with the scheduling of different combinations OS basic blocks; The important thing about program execution using speculation is that, instructions can use results of other speculated instructions but cannot write back to the registers until the speculated instructions are in harmony with the outcome of the control instruction and have already been committed to the registers. If the outcome of the control instruction is against the speculated instructions then all speculated instructions as well as the dependent instructions must be abandoned (flushed out).

The key idea is then to allow the out-of-order execution but to commit results in order to prevent any irrevocable action. In order to keep the results of the completed but incommitted instructions, a hardware buffer is provided. This reorder buffer provides additional virtual registers to extend the register set. It's a source of operands for instructions until the instruction is committed back. The entry in a reorder buffer contains following information

- **Instruction type:** the opcode of instruction, *i.e.* whether the instruction is a branch, store or ALU op;
- **Destination Field:** register number or the memory address where this result should go;
- **Value Field:** result of that instruction which is to sent to registers/memory;

Instructions are issued into reservation stations and allocated next reorder buffer (ROB) slot. Reservation station has several fields that are

- **Op:** the opcode of the instruction;
- **Q_j, Q_k:** ROB slot number that should produce result for the corresponding operand;
- **V_j, V_k:** known value of the corresponding operand;
- **A:** the address information for memory operations;
- **Busy:** a flag given whether this reservation station is busy or not;

In figure 13.4 is provided the architecture of the hardware based speculation. The execution steps of this architecture are

1. **Issue:** instruction is passed to the available reservation station from the instruction queue, and entry is made in the reorder buffer to ensure proper commitment of results;
2. **Execute:** as soon as the operands are available, the functional unit's start execution;
3. **Write Result:** results are written to the reorder buffer as soon as the common data bus (CDB) is available and also to the reservation stations which are awaiting this result. The reservation station is freed;
4. **Commit:** as the instruction reaches the head of the reorder buffer, it written back to the registers if the outcome of the control instruction is not against the speculated instructions;

13.3.1 Issue.

In the issue execution step, the architecture get instruction from instruction queue. If there is no reservation station or no ROB slot available, it stall the execution. The issue instruction is pass to the reservation station and to the reordered buffer, with operand if it's available in registers or ROB. After what it rewrite register operand with ROB slot of not yet available operand. The architecture then indicate target ROB slot to reservation station and finally do housekeeping (*i.e.* buffer management, indicate target ROB slot to target register,...)

13.3.2 Execute.

At this execution stage, if there is a missing operand, the architecture monitor the common data bus (CDB). When both operand are available, the functional unit execute its instructions which can take several cycles.

13.3.3 Write result.

This step simply broadcast result on the common data bus of the architecture, tagged with target reordered buffer slot.

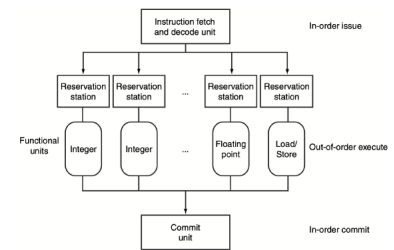


Figure 13.3: Hardware based speculation main principle.

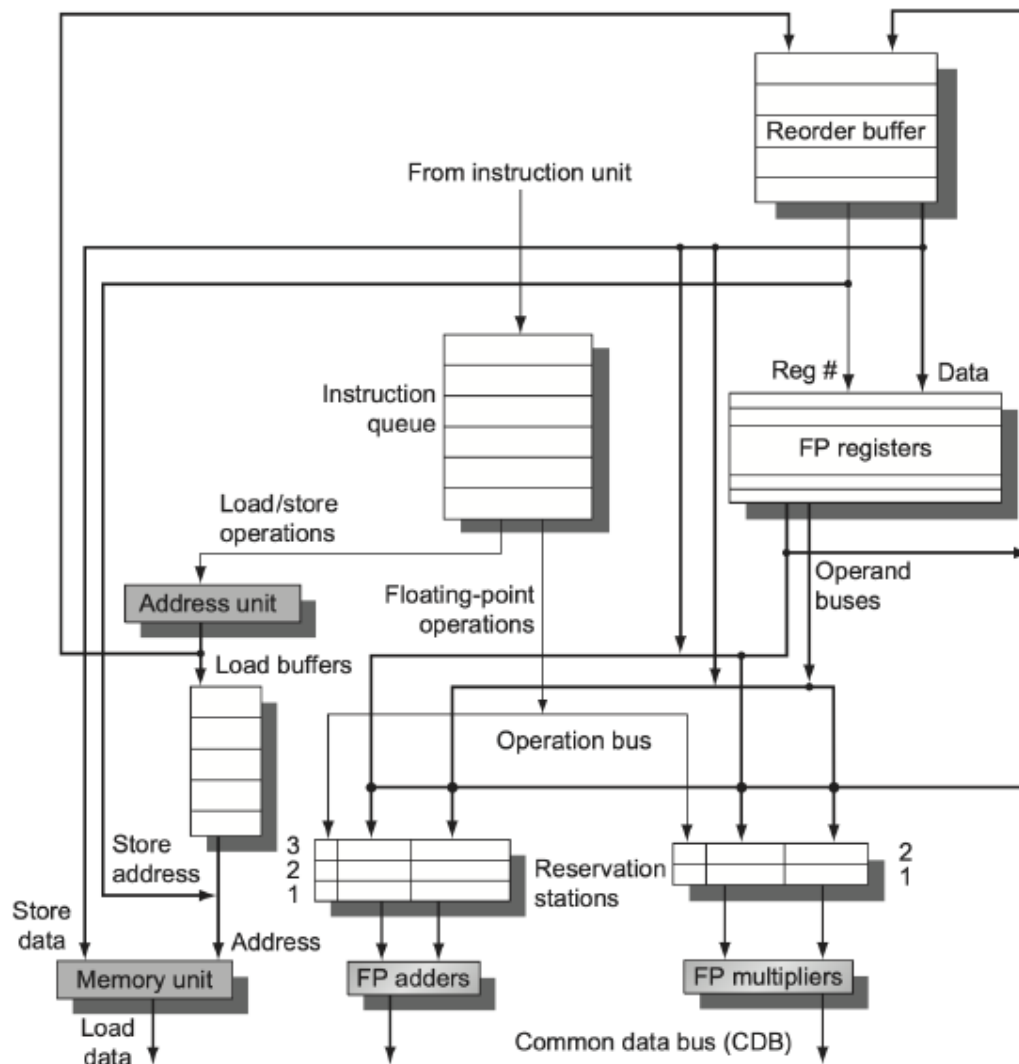


Figure 13.4: Hardware-based speculation architecture.

13.3.4 Commit.

With final stage of this architecture, when the instruction at the head of the reordered buffer has its results, the process update register/memory and remove instruction from ROB. If the instruction at the head of the ROB is a branch, then the prediction was right and the execution continue, else the prediction was wrong and the architecture flush the ROB, load correct branch successor and update the branch predictor.

SUMMARY

- The **instruction-level parallelism (ILP)** is a **measure** of how many **instructions** in a computer program **can be executed simultaneously**.
- We denote several types of **hazards** for the **pipelining** method
 - **Structural hazards**: there are **not enough** available **resources**;
 - **Data hazards**: an instruction is **still** in pipeline and produces result for **prior** instruction **already** in pipeline;
 - **Control hazards**: generate by **branches**;
- A **conditional JUMP** can either be "**not taken**" (**continue** execution), or "**taken**" (**jump** to a different place in program execution).
- The **branch prediction** mechanism attempt to **guess** which **state** of the **conditional jump** will be take in order to **reduce** the **latency** between the time when the instruction **enter** in the **pipeline** and the time where it reach the **execute stage**. The **branch** that is **guessed** is then **fetched** and **speculatively executed**.
- A **2-bits saturating counter** is a finite state machine of 4 states: **strongly not taken**, **weakly not taken**, **weakly taken** and **strongly taken**.

- An **adaptive predictor** remembers the **history** of the last **k occurrences** of the **branch** and uses one **saturating counter** for each of the possible **2^k history patterns** in a **prediction buffer**.
- **Hardware based speculations (HBS)** are based on three key concepts
 - **Dynamic Branch prediction:** choose which **instruction to execute**;
 - **Speculation:** allow the execution of **instructions before** the control **dependencies** are **resolved**;
 - **Dynamic scheduling:** deal with the **scheduling** of different **combinations** OS basic blocks;
- For programs using **speculation**, **instructions** can use **results** of other **speculated instructions** but cannot **write back** to the registers until the **speculated instructions** are in **harmony** with the **outcome** of the **control instruction** and have already been **committed** to the registers
- A **reorder buffer** provides additional **virtual registers** to extend the **register set** and save inside the **results** of the **completed** but **incommitted** instructions.

14 Interrupts.

Interrupts have already been briefly presented in section 6.2.4, where in our last version of the β -machine we add several new signals in order to manage code execution issues. In this section we go more into details about the treatment of these problem but also on how to use level of parallelism introduce in sections 11-13 in order to trigger traps and interrupts on the pipelined β -machine.

14.1 Control hazards.

Example 14.1: Control hazards example for a loop execution in the pipelined β -machine.

Imagine that the piece of code given in listing 14.1 is executed in the 4-stages pipelined β -machine.

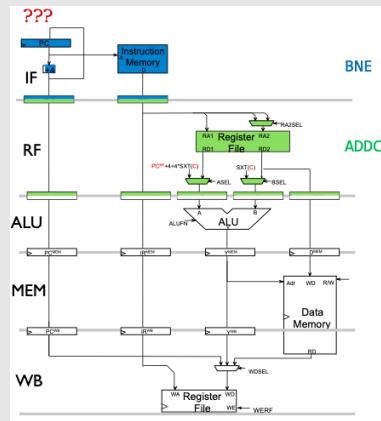
```

loop: ADDC(R1,4,R2)
      BNE(R3, loop)
      SUB(R6, R7, R8)
      :

```

Listing 14.1: Piece of executed code on the pipelined β -machine.

This execution is shown in the following diagram of the β -machine. When the BNE will enter in the IF stage, we have absolutely no idea of how to update PC value because R3 value is read one stage further.



From listing 5.6 it can be seen that in order to update the PC value while a branching instruction is computed, the **OPCODE**, **offset**, **PC+4**, **Reg[Ra]** values need to be known for simple condition branchings while just the **OPCODE** and **Reg[Ra]** need to be known for jumps. The problem here illustrate in example 14.1 on the pipelined β -machine is that all of these are only computed and known at the register file (RF) stage of the machine.

14.1.1 Solving control hazards.

On figure 14.1 is presented an hardware solution in order to solve the problem presented by the example 14.1. The idea here is to add a new control signal, **IRSrc^{IF}**, which will permit to choose

NOP as next executed instruction on the pipeline. If the OPCODE entering in the register file stage is the one of a BEQ, BNE or JMP instruction, then the machine set PCSEL to load branch or jump target and IRSrc^{IF} to 1 in order to inject a NOP. The new signals are shown in figure 14.1. This will permit to resolve hazards issues with speculation.

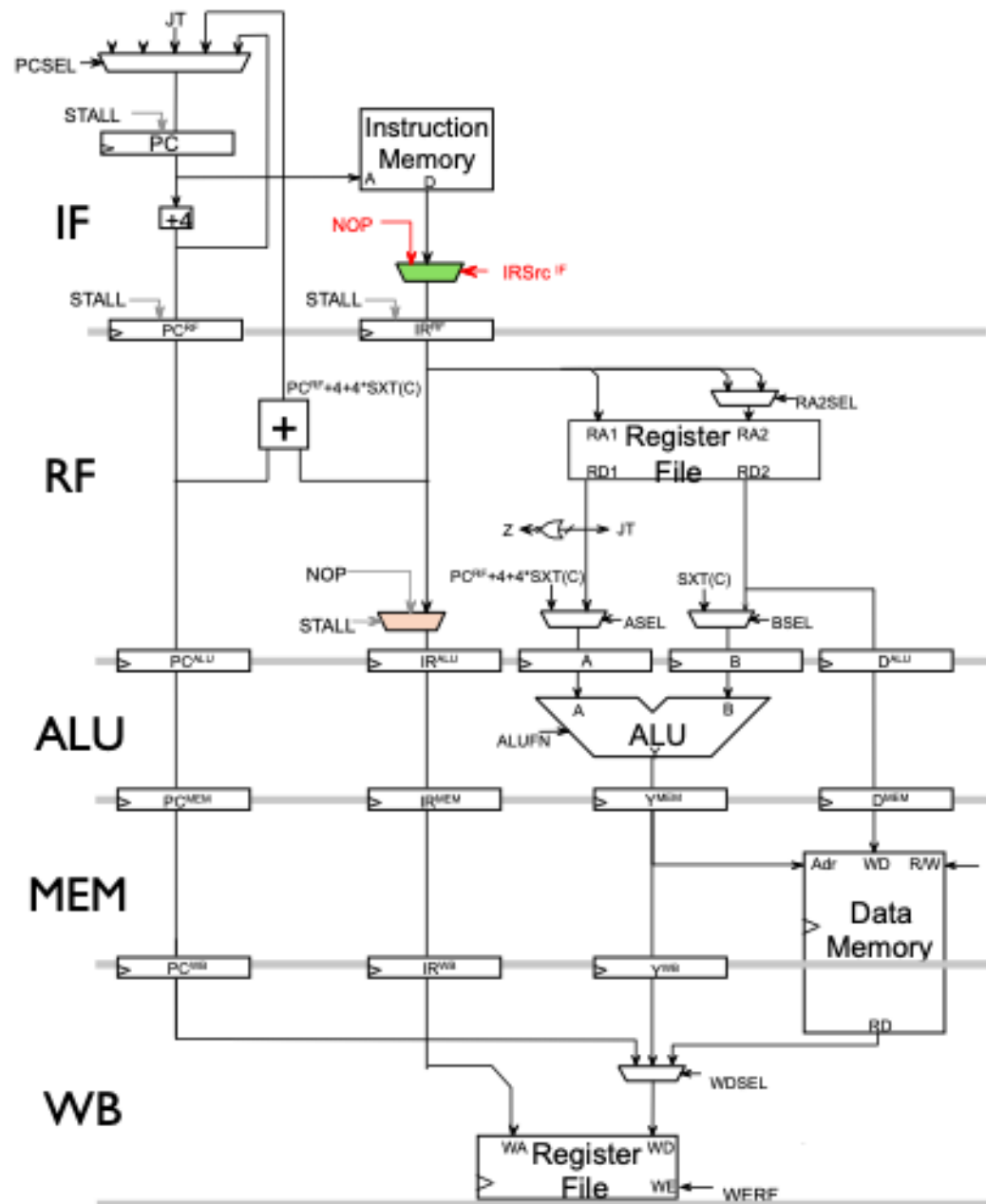


Figure 14.1: Stall logic for control hazards in the pipelined β -machine.

Example 14.2: A simple example of resolving hazards with speculation.

Imagine the execution of the code in listing 14.2 in the pipelined β -machine.

```

loop: ADDC(R1, -1, R3) MUL(R4, R5, R6)
      BNE(R3, loop)
      SUB(R6, R7, R8)
      XOR(R9, R10, R11)
      :

```

Listing 14.2: Basic code executed in the pipelined β -machine as illustration of speculation solution to control hazard.

Assume that the BNE is not taken in this example, we then have the following stage execution

Time slot	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
IF	ADDC	MUL	BNE	SUB	XOR
RF	...	ADDC	MUL	BNE	SUB	XOR
ALU	ADDC	MUL	BNE	SUB	XOR	...
MEM	ADDC	MUL	BNE	SUB	...
WB	ADDC	MUL	BNE	...

At stage executing time t_4 , the machine start fetching at PC+4 (SUB) but BNE is not resolved yet. Anyway here, the guess is right because the BNE is not taken so the execution keep going this way.

Example 14.3: A simple example of resolving hazards with speculation.

Imagine the same execution of the code in listing 14.2 still in the pipelined β -machine. Assume that the BNE is this time taken for this example, we now have the following stage execution

Time slot	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
IF	ADDC	MUL	BNE	SUB	ADDC	MUL	BNE	SUB	ADDC
RF	...	ADDC	MUL	BNE	NOP	ADDC	MUL	BNE	NOP
ALU	ADDC	MUL	BNE	NOP	ADDC	MUL	BNE
MEM	ADDC	MUL	BNE	NOP	ADDC	MUL
WB	ADDC	MUL	BNE	NOP	ADDC

At stage executing time t_4 , the machine still start fetching at PC+4 (SUB) thinking it has the right guess for PC value, even if BNE is not resolved yet. But this time, the guess value is erroneous because the BNE is taken. What the machine do in order to correct this error is that it cancel the execution of the SUB instruction at time t_5 for the RF stage by simply setting $IRSrc^{IF}$ to 1, which result to the choice of NOP instead of the next instruction as register file input.

14.2 Exceptions.

When a the machine meet an exception in a program execution, it need to

1. Save the current PC+4 inside XP, the exception point already define as R30 in section 6.2.4;
2. Load PC with exception vector (either `Illop` or `XAdr` value);
3. Execute the loaded exception manager;
4. Came back to the saved PC value in XP;

The exceptions obviously cause control flow hazard, because time execution will depend of triggered executions. They are implicit branches instructions, completely invisible from the user point of view. Also, exceptions need to be precised in order to be able to complete all preceding instructions but also to not executed instruction causing exception and future instructions (there are therefore no updates to register or memory). As we have seen it in section 6.2.4, it is pretty simple to implement in single-cycle machines, but it gets more complex with pipelining.

14.2.1 Exceptions classification in the pipelined β -machine.

We knows that an exception is triggered when something wrong occurs in a program execution. But it's also important to know what type of exceptions can happen in a pipelined β -machine and in which stages of it.

- **Instruction fetch:** memory fault (*e.g.* illegal memory address);
- **Register file:** illegal instruction, given by an erroneous OPCODE;

- **ALU:** arithmetic exception (*e.g.* divide by zero);
- **Memory or write back:** again a memory fault;

Instructions following the one that causes the exception may already be in the pipeline but none has written registers or memory yet, meaning that none of them already impact the machine state.

14.2.2 Resolving an exception.

If an instruction has an exception at stage i , the machine could simply turn that instruction into $\text{BNE}(\text{R31}, 0, \text{XP})$ to save $\text{PC}+4$, then annul instructions in stages $i - 1, \dots, 1$ helping by NOP operation and thus completely flush the pipeline. Finally it can set PC to either I110p or XAdr in order to treat an illegal operation or a wrong memory location exception respectively.

Example 14.4: Dealing with a memory fault for a LD instruction.

Imagine the running code in listing 14.3 in the pipelined β -machine. Here a memory fault is triggered for the LD instruction execution.

```
LD(R1, 4, R2)
ST(R3, 0, R4)
MUL(R4, R5, R6)
SUB(R7, R8, R9)
```

Listing 14.3: Executing code with LD memory fault on the pipelined β -machine.

The memory fault manager here will simply perform an ADDC instruction following by a ST. The pipeline execution scheme is provided in the table below.

Time slot	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
IF	LD	ST	MUL	SUB	ADDC	ST
RF	...	LD	ST	MUL	NOP	ADDC	ST	...
ALU	LD	ST	NOP	NOP	ADDC	...
MEM	LD	NOP	NOP	NOP	...
WB	BNE	NOP	NOP	...

At stage executing time t_4 , the memory fault is triggered by the MEM stage. As a result, this is a BNE that is executed by the WB stage and the whole pipeline is fill with NOP operations, except for the instruction fetch that already launch the XAdr exception context managers.

In order to efficiently implement these exception handlers on the pipelined β -machine, new signals need to be added to it. These signals and their constructions are shown in figure 14.2. We see in it that now the apparition of $\text{IRSrc}_{\text{IF}, \text{RF}, \text{ALU}, \text{MEM}}$ signals that control multiplexers in order to inject NOP (if preceding instruction has an exception) or BNE (if instruction in current stage has an exception) in each stages of the pipeline.

14.2.3 Resolving multiple exceptions.

From now, we saw some techniques about how managing a single exception inside a program running on the pipelined β -machine. As example 14.5 present it, when several exceptions are triggered, this is not the same thing.

Example 14.5: Dealing with a memory fault followed by an illegal operation.

Imagine the running code in listing 14.4 in the pipelined β -machine. Here a memory fault is triggered for the LD instruction execution and is followed by $\text{ILLEGAL}(\text{R2})$ which is an instruction with an illegal OPCODE.

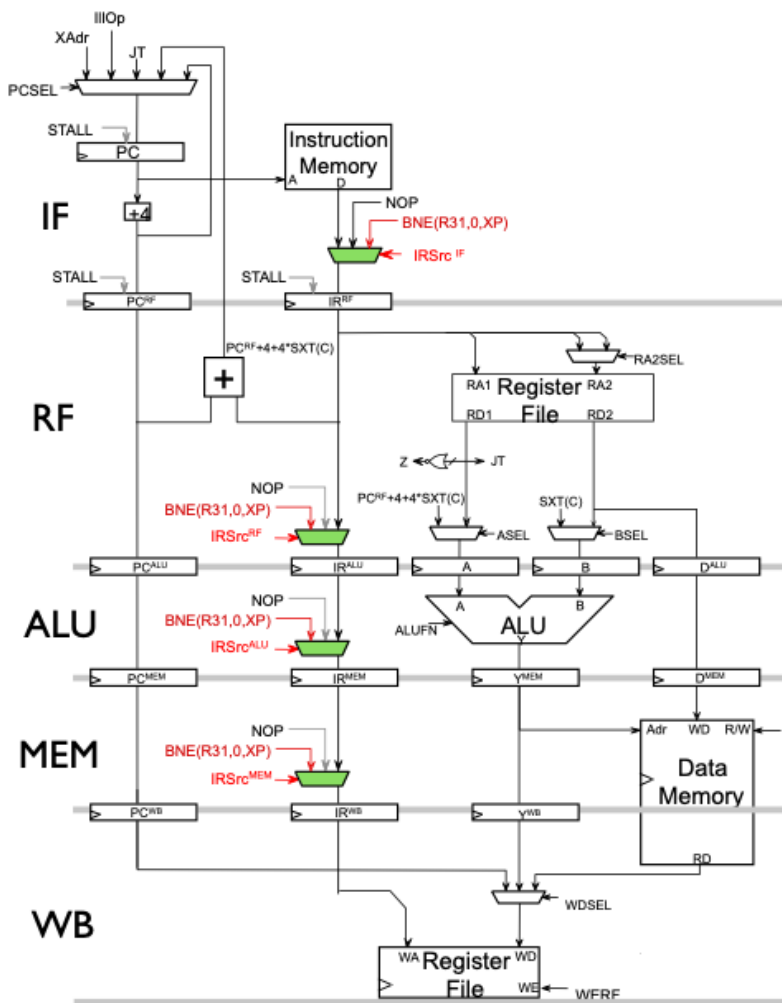
```
LD(R1, 4, R2)
ILLEGAL(R2)
MUL(R4, R5, R6)
SUB(R7, R8, R9)
```

Listing 14.4: Executing code with memory fault and illegal operation on the pipelined β -machine.

The memory fault manager here will still simply perform an ADDC instruction following by a ST and the illegal operation manager will executed a XORC followed by a SUBC. The pipeline execution scheme is provided in the table below.

Time slot	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
IF	LD	ILLEGAL	MUL	XORC	ADDC	ST
RF	...	LD	ILLEGAL	NOP	NOP	ADDC	ST	...
ALU	LD	BNE	NOP	NOP	ADDC	...
MEM	LD	NOP	NOP	NOP	...
WB	BNE	NOP	NOP	...

At stage executing time t_3 , an invalid OPCode is triggered by the RF stage, resulting by the launching of the illegal operation context described above at time t_4 . At this same time (t_4), the MEM stages detects a memory fault and thus launch the associate exception manager, also flushing the whole pipeline. With this structure, we see that this still work fine even if exceptions from latter instruction is detected first. Here also, the second triggered exception (memory fault) will be manage while the management of the ILLEGAL operation will not entirely be performed.

Figure 14.2: β -machine with new signals to deal with single exception.

The example 14.5 show that the pipelined β -machine as presented in figure 14.2 well manage exceptions, but could do even better by executing a whole exception context before launching another one. That's why the complete version of the pipelined β -machine (see figure 14.3) able to handle

exceptions will implement asynchronous interrupts, which save the current context while an exception is trigger, launch the exception manager and then came back to the initial saved context.

In this final version of the pipelined β -machine, if data hazards are triggered, the machine stall IF and RF stages (by setting STALL signal to 1 and $IRSrc^{RF}$ has the value that inject NOP). We also notice the apparition of bypasses for each stages' outputs in order to access not yet updated data in each of them.

For control hazards, the machine speculate PC+4's value and if its a JMP or taken branch in RF $IRSrc^{IF}$ is set in order to inject NOP again, and PCSEL is set to reach JT/branch target.

If an exception at stage X is trigger, $IRSrc^X$ is set in order to inject BNE operation. For all previous stages Y, $IRSrc^Y$ select the NOP operation. PCSEL is set to XAdr or I11Op depending of the triggered exception's nature.

If the machine face an interrupt, $IRSrc^{IF}$ is set the the value selecting the BNE instruction and PCSEL is simply set to the value that select XAdr's value.

14.2.4 Exception handler of the pipelined β -machine with hardware based speculator.

While the machine presented in figure 14.3 treat an interrupt, it could use an hardware speculator. With this, it will record interrupt in reordered buffer (see figure 13.4 for a reminder of the speculator architecture). When an external interrupt will reach the head of the ROB. This would permit to act on interrupts when interrupted instruction will be at the head of the ROB (*i.e.* will "jump" to interrupt handler).

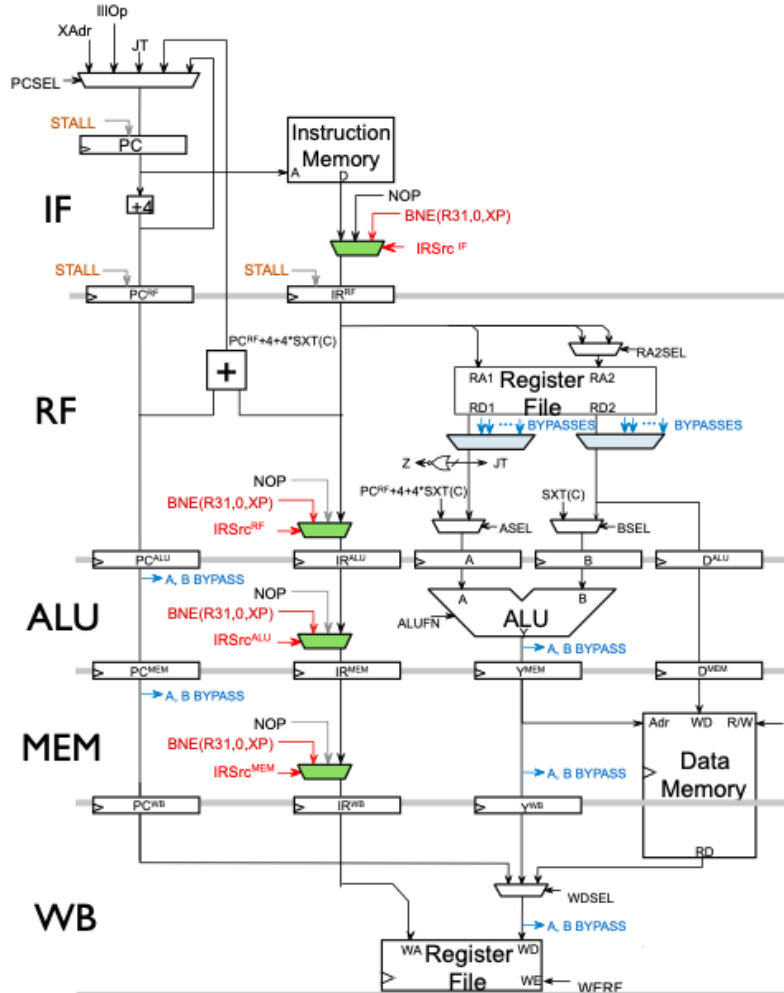


Figure 14.3: Final version of the pipelined β -machine implementing exception handler.

SUMMARY

- An **exceptions** is an **anomalous** or **exceptional conditions** requiring special processing during a program execution.
- An **interrupt** is an input **signal** to the **processor** indicating an **event** that needs **immediate attention**.
- When an exception is encountered the machine
 - **Save** the current **PC+4** inside **XP**;
 - **Load PC** with **exception vector** (either **ILLOp** or **XAdr** value);
 - **Execute** the loaded **exception manager**;
 - **Came back** to the **saved PC** value in **XP**;
- Each pipeline's stage has its own type of exception
 - **Instruction fetch**: **memory** fault;
 - **Register file**: illegal instruction, given by an **erroneous OP CODE**;
 - **ALU**: **arithmetic** exception;
 - **Memory or write back**: again a **memory** fault;
- **Exceptions** can be **manage** with **software** solutions, **hardware** ones or helping by a **speculator**.

15 Modern computer architectures.

This section is the icing on the cake of the course, it present few current computer architectures used nowadays. With some examples you will see that what we have done, even if it was made with a lot of simplifications, is not far away from the reality. Even if the β -machine could be seen as a really simple pre-historic computer model, the steps of its construction respect pretty well the way modern processors are design, even if they are more optimum.

15.1 RISC-V architecture.

RISC-V is an open-source hardware instruction set architecture (ISA), based on established reduced instruction set computer (RISC) principles, similar to the one presented in section 5.4 for the β -machine. Unlike other academic designs which are optimized only for simplicity of exposition, the designers state that the RISC-V instruction set is for practical computers. It is said to have features to increase computer speed, yet reduce cost and power use. This architecture is the latest iteration of an open source ISA. It has a Berkeley Software Distribution (BSD) license, which means that it's part of a family of permissive free software licenses, imposing minimal restrictions on the use and distribution of covered software. Derivatives of the RISC-V architecture can thus be either open or closed, depending of the developers interests. Only a few paid members of the RISC-V foundation can vote to approve changes or utilize the trademarked compatibility logo.

This ISA began in 2010 at the University of California, Berkeley, getting momentum and more and more volunteers contributors since. The project is however only hardware available since 2018. It is supported by most of compilers such as Clang, GCC,... (even CompCert).

Since it's creation, many big companies including Nvidia, Alibaba, Western Digital,... was interested about involve themselves in the project.

15.1.1 RISC-V main characteristics.

Inside the RISC-V architecture we found out several main characteristics

- **Load-store instructions**: (such as for the β -machine), which are instructions with address only in registers, with load and store, instructions that still conveying to and from memory (note that unaligned addresses are allowed);
- **Little-endian order**: this means that the machine running RISC-V places the least significant byte first and the most significant byte last. Conversely the big-endian ordering does the opposite;
- **RV32I/RV32E**: these 2 RISC-V differents versions use 32-bits registers. We found 32 of them for the RV32I version while we have 16 registers only for the RV32E version (I stands for Integers and E for embedded);
- **RV64I2**: this RISC-V version use 32 registers of 64 bits each;
- **Instructions length**: on RISC-V machines, instructions are 32-bits long, or of variable length (C extension);

A new version of RISC-V compatible with 128-bits CPU is still in development. The whole documentation about the RISC-V architecture can be found on the company's website⁴. In table 15.1

Base Integer Instruction Sets	
RV32I	32-bit, 31 registers (R0 is 0)
RV32E	32-bit embedded, 15 registers (R0 is 0)
RV64I	64-bit, 31 registers (R0 is 0)
Extensions	
M	Integer Multiplication and Division
A	Atomic Instructions
F	Single-Precision Floating-Point
D	Double-Precision Floating-Point
G	Shorthand for the I base and above extensions
Q	Quad-Precision Floating-Point
C	Compressed Instructions

Table 15.1: RISC-V instruction set structure's parts and optional extensions.

⁴ <https://riscv.org/>

are provided some RISC-V architecture and some of their extensions. For instance, when a RV32EC run on a machine, it's the RISC-V 32-bit core in embedded system architecture that is designated.

15.2 ARM architecture.

ARM, previously designed Advanced RISC Machine but originally record as Acorn RISC Machine, is a family of reduced instruction set computing (RISC) architectures, such as the β -machine is one of them, for computer processors and configured for various environments. The ARM Holding business model is based on IP (intellectual property). Core licenses are sold to companies that will integrate them (possibly with other devices) on their chips. This company has manufactured not less than 10^{11} processors (≈ 20 per people) since 1983 and these are used in embedded systems, internet of things (IoT), phones, Raspberry Pi, laptops, desktops,... It exists several profile of ARM architecture, each having its own work area

- **A:** designed for application, high performances computing with full-fledged OS (*e.g.* GNU/Linux, Android);
- **R:** used for real-time systems, such as networking equipment and embedded systems;
- **M:** made for IoT, small and low power systems

Inside a single phone modern phone, besides the A-profile cores (advertised in the product sheet), we can found out many secondary A, M, and R cores responsible of task like network connection, encryption, imaging process,...

15.2.1 ARM main characteristics.

Inside the ARM architecture we found out several main characteristics

- **Load-store instructions:** (such as for the β -machine), which are instructions with address only in registers, with load and store, instructions that still conveying to and from memory (note that unaligned addresses are allowed);
- **Little-endian order:** this feature is the default one for this architecture but can be configured if warranted, unlike with RISC-V architecture.
- **ARM/A32:** this ARM version use 32-bits registers. We found 15 of them inside these architecture created in 1983;
- **ARM/A64:** this ARM version use 32 registers (31 registers + 0), these including reserved registers such as SP, of 64 bits each. This version has been release in 2011;
- **Instructions length:** on ARM machines, instructions are 32-bits long for the ARMv7 version, or 16-bits long for the Thumb32 extension;

Once again, the whole documentation about the ARM architectures can be found on the company's website⁵.

⁵ <https://developer.arm.com/>

15.3 Intel architecture.

Intel, for integrated electronics, is the world's second largest and second highest valued semiconductor chip manufacturer based on revenue (after being overtaken by Samsung Electronics) founded in 1969. It's also the inventor of the x86 series of microprocessors, the processors found in most personal computers. The Intel company is the direct competitor of the well Advanced Micro Devices (AMD) company, also specialized in microprocessors for PCs. Intel developed the x86 line, whose first components were the Intel 8086 (in 1978, successor of the 8-bits processor 8080), than came the updates

- **Intel 80186:** in 1982, still using 16-bits addresses but now permitting fast hardware address calculation;
- **Intel 80286:** in 1982, still using 16-bits addresses but with MMU (Memory Management Unit), to allow protected mode and a larger address space;
- **Intel 80386:** in 1985, using 32-bits addresses and thus now an instruction set on 32-bits still with a MMU integrated;

, , Intel agreed to license the x86 family to AMD in 1982, in order to get IBM market (who wanted at this time minimum two suppliers of processors). Since then, Intel and AMD (and VIA) shared

licensing. x86 means binary compatibility with the 32-bit instruction set of 80386. In 2001, AMD developed a 64 bits extension (x64) for processors, and licensed it to Intel. Anyway, Intel still dominates the server, desktop and laptop market today. The complex instruction set computer (CISC) for x86/x64 families added on average one instruction per month over 30+ year lifetime.

15.3.1 x86/x64 main characteristics.

Inside the x86/x64 architectures we found out several main characteristics

- **Register-Memory:** instructions can operate both on registers and memory and unaligned addresses are still allowed;
- **Little-endian order:** this is still the policy employed by these processors family. As a reminder, this means that the least significant byte is placed first and the most significant byte in last;
- **Registers:** this architecture use 16 registers, some of them for special purposes, of 64-bits or 32-bits;
- **Instructions length:** on x86/x64 machines, instructions are of variable size (1, 2, 3 up to 15 bytes);

Once again, the whole documentation about the Intel architectures can be found on the company's website⁶.

⁶ <https://software.intel.com/en-us/articles/intel-sdm>

15.4 Example of assembly code for each presented architectures.

Example 15.1: Illustration of assembly code for RISC-V, ARM and Intel architectures.

Imagine the code given in listing 15.1 executed on a RISC-V, an ARM and an Intel machine. Each of these machine has its own instruction set as explain previously, thus for each of them the compiler will translate into different assembly codes generating a processor-readable set of instruction. These assembly translations of the code in listing 15.1 are provide for RISC-V architecture in listing 15.2, for ARM architecture in listing 15.3 and for Intel architecture in listing 15.4.

```
int array_max(unsigned *A, unsigned n){
    unsigned i, max = 0, *p = A;
    for(i = n; i > 0; i--){
        int value = *p;
        if(value > max)
            max = value;
        p++;
    }
    return max;
}
```

Listing 15.1: C code for the maximum value of an array research.

```
array_max:
    beqz a1, .L5          # if a1==0 goto .L5
    addw a4,a1,-1         # a4 = a1 - 1
    sll a4,a4,32          # a4 = a4 << 32
    srl a4,a4,32          # a4 = a4 >> 32
    add a4,a4,1           # a4 = a4 + 1 // 32 to 64 bits
    sll a4,a4,2           # a4 = a4 << 2 // * 4
    add a4,a0,a4          # a4 = a4 + a0 = A + n
    li a3,0              # a3 = 0
.L4:
    lw a5,0(a0)          # a5 = *a0
```

```

    add a0,a0,4      # a0 = a0 + 4
    bgeu a5,a3,.L3   # if a5 > a3 goto .L3
    mv a5,a3        # a5 = a3
.L3:
    mv a3,a5        # a3 = a5
    bne a4,a0,.L4    # if a0 != a4 goto .L4
    sext.w a0,a5     # a0 = a5
    ret             # return
.L5:
    li a0,0         # a0 = 0
    ret

```

Listing 15.2: RISC-V assembly code of the maximum value of an array research C code.

```

array_max:
    cbz r1, .L4      @ if r1 == 0 goto .L4
    movs r3, #0      @ r3 = 0
.L3:
    ldr r2, [r0], #4  @ r2 = *(r0+4); r0 += 4
    cmp r3, r2        @ r3 < r2
    it cc             @ if yes then execute next
    movcc r3, r2      @ r3 = r2
    subs r1, r1, #1   @ r1--
    bne .L3           @ if r1 != 0 goto .L3
    mov r0, r3        @ r0 = r3
    bx lr            @ return
.L4:
    mov r0, r1        @ r0 = r1 (i.e. 0)
    bx lr            @ return

```

Listing 15.3: Arm assembly code of the maximum value of an array research C code.

```

.LFB0:
    mov eax, esi      ; eax = esi
    test esi, esi     ; esi == 0?
    je .L2            ; if esi == 0 goto .L2
    lea eax, -1[rsi]   ; eax = rsi - 1
    lea rcx, 4[rdi+rax*4] ; rcx = 4 + rdi + rax * 4
    mov eax, 0        ; eax = 0
.L3:
    mov edx, DWORD PTR [rdi]; edx = *rdi
    cmp eax, edx       ; edx > eax?
    cmovb eax, edx     ; If (edx > eax) eax = edx
    add rdi, 4         ; rdi += 4
    cmp rcx, rdi       ; rdi == rcx ?
    jne .L3           ; if rdi != rcx goto .L3
.L2:
    rep ret           ; return

```

Listing 15.4: x86-64 assembly code of the maximum value of an array research C code.

SUMMARY

- **RISC-V** is an **open-source** hardware **instruction set architecture** based on a **RISC**;
- **ARM** (Acorn RISC Machine) is an **reduced instruction set computing** (RISC) architectures based on an **intellectual property** business model.
- The **Intel x86** is a **complex instruction set computer** CISC architecture based on **revenue**.