Microelectronic Systems Final Project

Design and Development of DLX Microprocessors

Mariagrazia GRAZIANO
Giulia SANTORO
Luca GNOLI
Andrea COLUCCIO
Giovanna TURVANI

July 20, 2020

Contents

Chapter 1

DLX Structure: Hint and Recall

In this project you will refining your small DLX processor from RTL down to physical design. Your DLX version will be fully pipelined.

The project consists of completing the control unit partially given and based on the same structures analysed in lab4 for a subset of the instruction set and designing the whole data path. The design should start from the VHDL level and go down to the synthesis and physical design phase.

The specifications will be organized in two items. The first, which we will call **DLX-basic** is due. The second **DLX-pro** is optional. If you choose this type of project and decide to go besides the first step (DLXbasic), you can fulfill from one to all the specifications of the DLX-pro version.

Attention:
The DLX-basic VS DLX-pro choice is individual. You can split and recombine the lab groups in project teams if you have different ideas... The Project Team should have the same target (Basic or Pro) and can be composed of 1 or 2 students

The specifications from point 1 to point 6 **ARE MANDATORY** for every DLX (Basic and Pro) model.

From point 7 to 14 you can find a list of possible specifications for improving your **DLX-PRO** design. You can choose from one to all as you prefer.

DLX-basic Specifications:

1. **Pipeline (BASIC)** The architecture and control must be organized following the five stages pipeline. You have three ways: go further with microcontrolled CU, or change to a clever FSM or hardwired CU. Choose meaningful assembler programs so that pipeline and pipeline stall are underlined.
2. **Instructions subset (BASIC)** You should extend the control unit execution so that the following instructions can be executed:

   add
   addi
   and
   andi

beqz
bnez
j
jal
lw
nop
or
ori
sge
sgei
sle
slei
sll
slli
sne
snei
srl
srli
sub
subi
sw
xor
xori

Design modify and COMMENT one (or more) intelligent assembler programs so that these instructions can be meaningfully checked. Change the ALU OPCODE so that an enumeration type can be used (see appendices A and B).

3. **Data path (MANDATORY)** You should describe in VHDL (or Verilog) at RT level all the data path components necessary to fulfill the instruction subset defined at previous points. Of course you can reuse the blocks you already described in previous labs. Describe one or more intelligent assembler programs (COMMENTED) so that these instructions can be meaningfully checked.
4. **Synthesis (BASIC)** Both the data path and the control unit must be synthesized. Different synthesis results can be reported. A final optimization for frequency must be performed. The scripts you use for the synthesis step must be reported and commented (memory should not be synthesized).
5. **Physical design (BASIC)** The synthesized design must be placed and routed. Post physical design performance must be reported: delay, EM, thermal informations, power... before and after optimization steps. (Follow what you learn in the labs.)
6. **Report (MANDATORY)** A final report describing your design, your code, your scripts, results and reports is due together with all the code. Remember that a report is an important aspect of a project: it let one know how detailed was your work and how to use it. This will be evaluated as well.

DLX-pro Hints:

7. **Instructions subset (RECOMMENDED)** The following instructions (a subset or all of them) could be added in order to achieve pro functionality:

   addu, addui, jalr, jr, lb, lbu, lhi, lhu, sb, seq, seqi, sgeu, sgeui, sgt, sgti, sgtu, sgtui, slt, slti, sltu, sltui, sra, srai, subu, subui, mult (using integer registers: something should be modified).
8. **Data path (OPTIONAL)** The data path can be extended for each of the instruction you choose to add. Furthermore, independently from the instruction, try to optimize the microarchitecture so that the critical path is reduced (e.g. for the ALU you can choose one or more of the arithmetical structures analysed during lectures or suggested by the teacher.
9. **Windowing (OPTIONAL)** Use the windowed register file to support routine context switching and complete the control unit to support it.
10. **Control hazard (OPTIONAL)** Implement one or more of the techniques to prevent stall as mentioned during classes as instruction queue for jump (IQ), branch prediction using a small branch history table.
11. **Optimize power consumption from an architectural point of view (OPTIONAL)** Improve performance using: Parallelizing, pipelining, Feedback, Clock Gating, Isolation...
12. **Caching (OPTIONAL)** Add a small data cache (RTL) between the main memory and the CPU; define type of association, a coherency policy and a replacement strategy. (Of course the CU must be updated accordingly)
13. **Advanced synthesis (OPTIONAL)** Force further optimization to the design: try to reduce power consumption, perform a post synthesis VHDL simulation, so that realistic timing and power simulation with a real test bench can be performed (you will ask suggestions on this points).
14. **Physical design (OPTIONAL)** Post physical design simulations, clock tree synthesis and even crosstalk analysis would be appreciated in the final report.
15. **Whatever you want not mentioned before... (FUNDAMENTAL)** Feel you **FREE** to add features to your personal DLX!

The final grade for the **DLX-basic** project will be in the best case stopped at **27/30**, Instead for the **DLX-pro** the final grade can saturate up to **33/30 (30 with Honor)**! Of course, depending on the quality and quantity of your work.

Anyway the course final grade will be a combination of the lab works, of the project and of the oral exam.

Chapter 2

Microelectronics Systems final project specifications

In this chapter you will:

1. First recall the DLX behavior
2. Analyse the DLX test bench given
3. Debug the preliminary hardwired version of the control unit with some control signals generation capapility.
4. Learn how to add other instructions (example the addition of an immediate). This will be one of the possible implementation of the control unit for your final project.
5. You also start from a partially developed FSM control unit, that you should complete for one instruction of your choice. This will be one of the possible implementation of the control unit for your final project.
6. Debug and understand a microprogrammed control unit version already prepared for a jump instruction and add another instruction of your choice.
7. Using details on DLX instruction set (given in appendix B).
8. A few details on how to implement a FSM are given in the last section (2.2).

The given structure is the starting point for the control unit of your final DLX project. You are given a set of starting programs and examples. Copy them from the usual directory: **/home/-mariagrazia.graziano/ms/teamB/cap4/.**.

2.1 DLX instruction set and basic operations

Remember that the DLX processor (see cap.2 and 3 of Hennessy-Patterson) is a simple *load-store* architecture with 32 32-bit general-purpose registers (GPRs), named *R0, R1,...,R31.* A sketch of the datapath is in figure 2.1. The addressing modes are *immediate* and *displacement* with 16-bit fields. Register deferred is accomplished by placing 0 in the 16-bit displacement field, and absolute addressing with a 16-bit field is accomplished by using register *R0* (its value is always zero): four effective modes are available.

DLX memory is byte addressable in Big Endian mode with a 32-bit address; all memory references are through loads or stores between memory and the GPRs (RF).

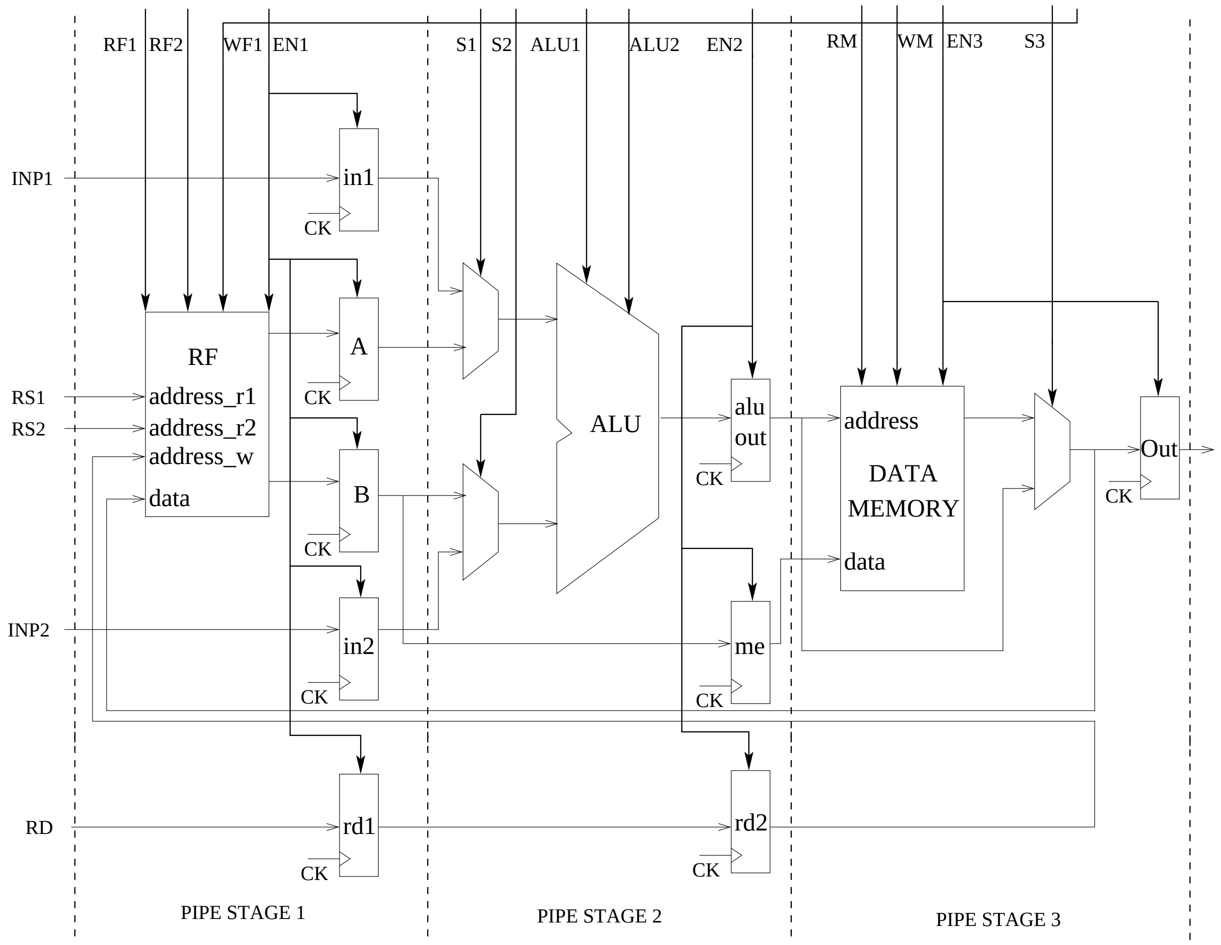


Figure 2.1:

DLX Instructions are grouped into 3 main types :

- *I-type*: Loads/Stores and conditional branch instructions.
- *R-type*: Register-register ALU operations. ALU operation is defined in the extra 11-bit field *func*.
- *J-types*: Jump and jump link instructions

In all these three types, the 6-bit *opcode* field is always present. Depending on the instruction type, the remaining 26 bits assume different meanings (fig. 2.3).

Examples of load and store instructions:

```
Load Word: LW R1, 30(R2) (Meaning: R1<- Mem[30+R2])
Store Word: SW R3, 500(R4) (Meaning: Mem[500+R4]<-R3)
```

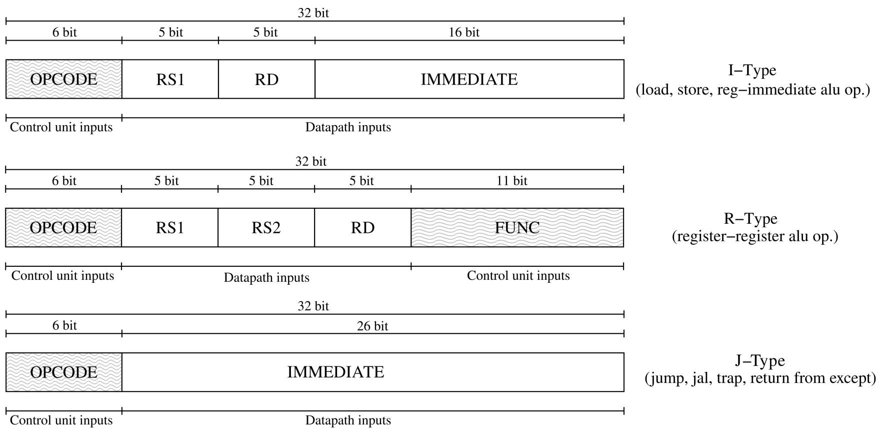


Figure 2.2:

Example of ALU instructions:

```
Add: ADD R1,R2,R3 (Meaning: R1<-R2+R3)
Add Immediate: ADDI R1,R2,\#3 (Meaning: R1<-R2+3)
```

Example of control-flow instructions:

```
Jump: J name (Meaning: PC<-name)
Branch equal zero: BEQZ R4,name (Meaning: if(R4==0) PC<-name)
```

Every DLX instruction can be implemented in at most five clock cycles:

- *Instruction fetch(IF)* cycle: Send out the PC and fetch the instruction from memory into the instruction register (IR) and increment the PC by 4 to address the next sequential instruction. The NPC register is used to hold the next sequential PC.

$$IR < - MEM[PC]$$
$$NPC < - PC + 4$$

- *Instruction decode/register fetch(ID)* cycle: Decode the instruction and access the register file (RF) to read the registers. The outputs of the general-purpose registers are read into two temporary registers (A and B).

$$\begin{gathered} A < - regF[IR6...IR10] \\ B < - regF[IR11...IR15] \\ C < - regF[IR16...IR31] \end{gathered} \tag{2.1}$$

- *Execution/effective address cycle(EX)*: The ALU operates on the operands (A and B) prepared in the previous cycle and the result is stored in the ALUOutput register.

$$\begin{array}{ll} ALU-OUT <- A + IMM & \text{Memory reference} \\ ALU-OUT <- A\ func\ B & \text{Alu Operation} \\ ALU-OUT <- A\ op\ IMM & \text{Register-Immediate ALU} \\ ALU-OUT <- NPC + IMM & \text{Address del Branch} \end{array} \tag{2.2}$$

- *Memory access/branch completion(MEM)* cycle : Access memory if needed. If instruction is a load, data return from memory and is placed in the LMD (Load Memory Data) register. If it is a store, the data from the B register is written into memory. In either case the address used is the one computed in the prior cycle and stored in the ALUOutput register.

$$\begin{array}{c} LMD <- MEM[ALU - OUT] \\ MEM <- [ALU - OUT]\ <- B \end{array} \tag{2.3}$$

- *Write-back(WB)* cycle: Write the result into the register file, whether it comes from the memory system or from ALU; the register destination field is also in one of the two positions depending on the function code.

$$\begin{array}{c} regF[IR6...IR10] < -ALU - OUT \\ regF[IR11...IR15] < -ALU - OUT \\ regF[IR16...IR31] < -ALU - OUT \end{array} \tag{2.4}$$

This data path is composed of simple blocks which you have already described in previous labs. Collecting them will be your final project job. But now, let's concentrate on how control signals can be generated and delivered to the data path.

2.1.1 Instruction encoding

As our aim is implementing the control unit, we must be aware of the instruction coding. This is typically a designer choice. Here we maintain the Hennessy-Patterson coding reported in table A (appendix A).

It is now important that you understand the meaning of the assembler output (see section 3.3 hexadecimal numbers in order to proceed with the work. In the following a graphical example for the first two instructions in our test.asm.mem program is given.

2.2 DLX Control Unit

Recalling section 3.1 we provide you three preliminary versions of the control unit: the hardwired (CU_HW.vhd), the FSM (CU_FSM.vhd) and the microprogrammed (CU_UP.vhd).

In all the cases the control unit assumes that each stage in the DLX has a register (as in figure 2.1). Anyway still it does not really manage the pipeline: this is a step that you are going to fulfill for the final project.

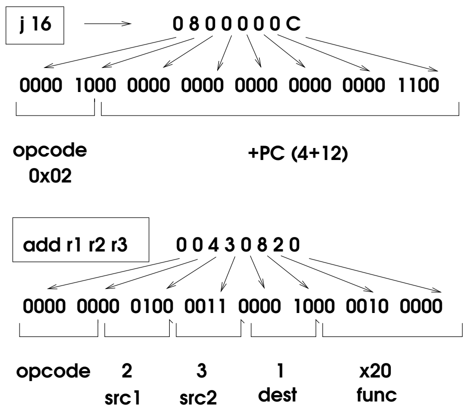


Figure 2.3:

In all the cases the CU organization is as in figure 2.4, where the essential control signals (control word) are defined and ordered according to the activity in the five stages (clearly, as soon as your project becomes a PRO one, these signals can increase in number).

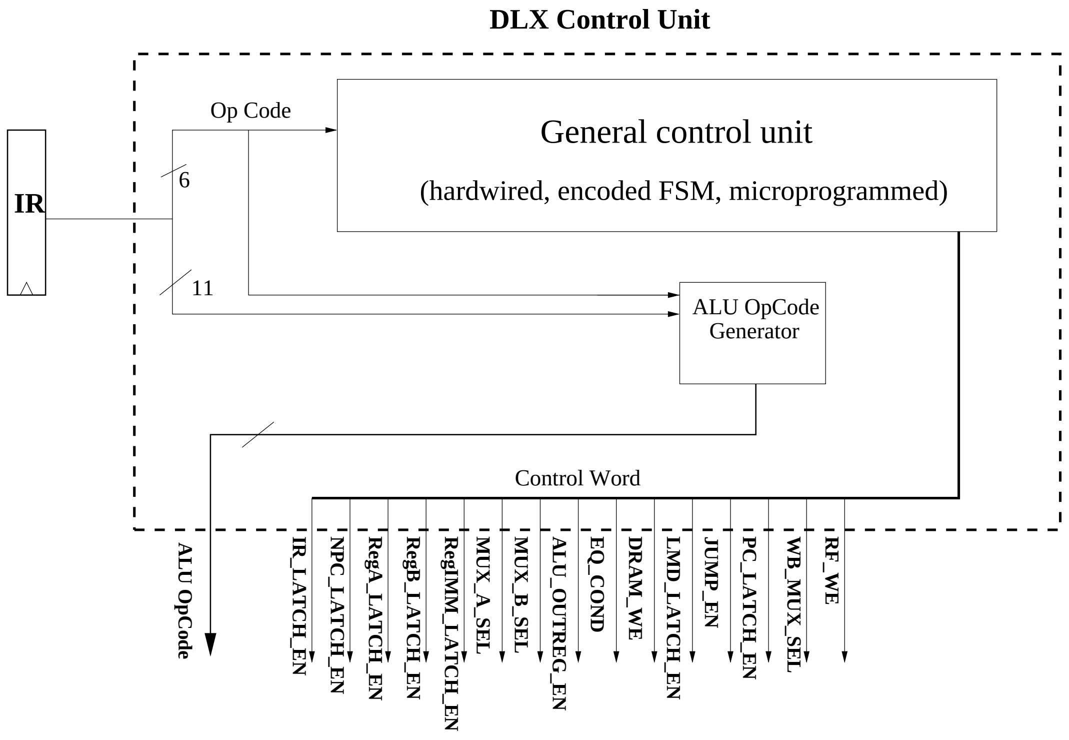


Figure 2.4:

It is important that you now associate these signals to the datapath previously recalled in figure 2.1. Notice that the control unit has a separate block for the ALU opcode generation (distributed control).

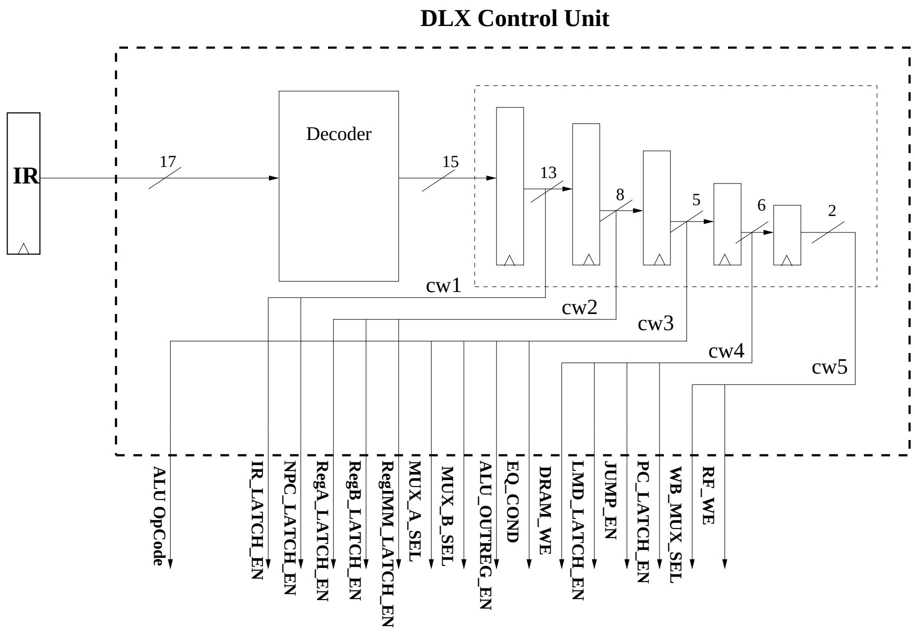


Figure 2.5:

2.2.1 HARDWIRED control unit

Start from the hardwired version: CU_HW.vhd. It is not complete, but can be simulated at least in an initial condition. As you see in the code the control word is delayed to correctly synchronize control signals and stages in which they must be active. An idea is in figure 2.5.

NOW YOU ARE READY TO START:

1. Set up the simulation:
    - Set-up the modelsim environment variable **setmentor**
    - create the work library (vlib work)
    - compile the five vhdl files *mytypes.vhd, IRAM.vhd, CU_HW.vhd, DLX.vhd, TB_DLX.vhd.*
2. Simulate for a few clock cycles (around 40ns).

Now it's important that you debug the whole behavior, in order to be ready to:

1. analyze if control signals are correctly set to manage the first instruction in the example (a jump)
2. implement what is necessary in order to generate the control signals for the other instructions in the example (only the control signals! Assume you have already the data path).

in Summary:

Hardwired control unit executing the JUMP, the ADD and the ADDI instructions: VHDL netlist (well commented!) and meaningful waveforms.

2.2.2 FSM control unit

Using the same environment you can now work with the FSM version. You are given a preliminary version in file CU_FSM.vhd. Clearly, to complete it, we assume you know how to describe a FSM in vhdl. If not, a small example of a two state FSM is given (fsm1.vhd and test bench tb_fsm.vhd to simulate it, a few details are in section **??**). As soon as you have clear how to describe it in vhdl, complete the given FSM version in order to manage the jump and the two add instructions.

in Summary:

FSM control unit executing the JUMP, the ADD and the ADDI instructions: VHDL netlist (well commented!) and meaningful waveforms.

2.2.3 Micro programmed control unit

The Control Unit (CU) is supposed to generate all the required output control signals (latch enables/muxes selections/ALU OpCode) for the datapath on the basis of the operation type.

An example of *microprogrammed* control unit is provided for this lab (fig. 2.6). The **CU_UP.vhd** reflects the diagram in figure.

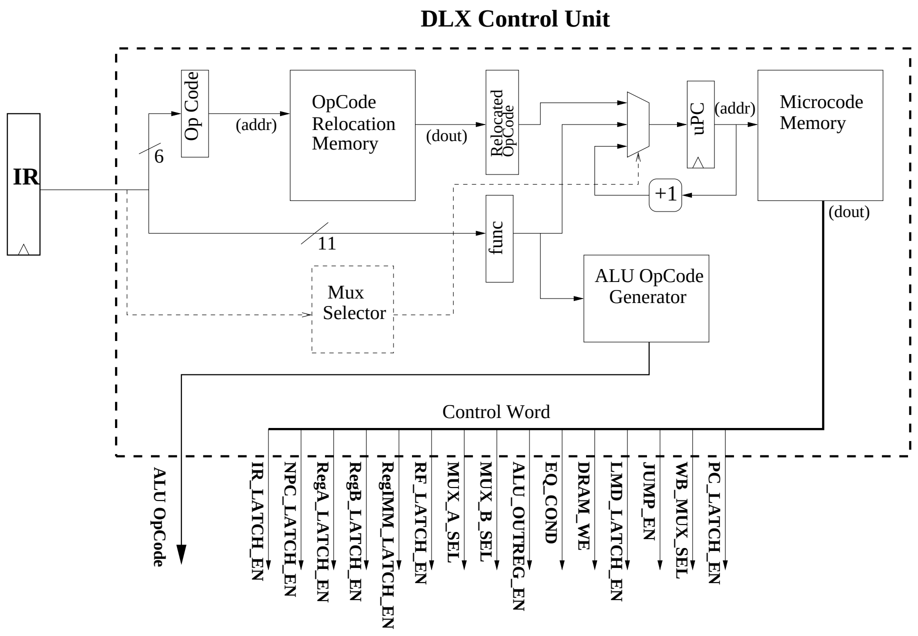


Figure 2.6:

The microcode memory stores the appropriate control signals depending on the instruction and on the stage(IF,ID,EX,MEM,WB) of the instruction, thus implementing the FSM for the given instruction. The OpCode of the instruction is used as a *starting-address* for the microcode memory and it is stored in the uPC register. In a normal five clock-cycle instruction, the basic address stored in the uPC is incremented five times in order to generate the subsequent control signals for all the datapath stages. In case of *R-type* instructions, an extra opcode is used to specify instruction behaviour using the 11-bit filed *func*. This means that in this case the value of the func field is stored in the uPC instead of the normal opcode field.

Since the OpCode is used as an address to map the control in the five different stages, the value of each OpCode must differ of at least five values. In the DLX architecture some OpCodes (reported at the end of this chapter) differ by less then five values, as this is thought for a simpler FSM control. Thus, in this exercise, a *OpCode Relocation Memory* has been used. The relocation memory simply remaps a given value to another value. In this case, OpCodes which are too close will be remapped with a new address giving the possibility to implement the required five stages.

In the **dlxasm.pl** assembler, so as at paragraph **??** opcodes (end of section) for all the instructions can be find.

Now it's up to you

Once you have clearly understood the CU behavior in the JUMP case (from assembler to hexadecimal code, from instruction memory to relocation memory, from microcodes to control world), try to program the CU so that the ADD instruction can be executed by the control unit as well, simulate it and check the correct CU output behavior (**attention you should not include the ALU here, but just check that the ALU control signals are managed correctly by the control unit**).

in Summary:

Microprogrammed Control unit executing the ADD instruction: VHDL netlist and meaningful waveform.

Chapter 3

How To Manage and Submit the Project

In order to simplify the project start we provide you:

- Top Levels Implementation
- Files organization structures: the files very important in order to work efficiently, without wasting time to search and rename files at the end of the project (when the number bursts).
- Testing Tools and BenchMark Methods

3.1 Project Set Up

We provide you the top level component interface and some other example of Control Unit and Test Bench to modify and complete. Let's have a look with a top-bottom order. The test bench, sketched in figure 3.1, just instances the DLX, sets clock, reset and a few parameters. The file is DLX_Project_Files/vhd/test_benc/TB_DLX.vhd.

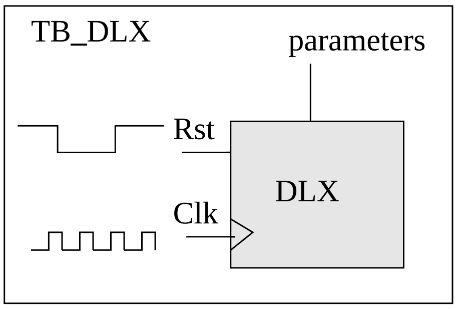


Figure 3.1: this structure is NOT REALISTIC, but is easy to use, if you want improve, migrating towards more realistic, CONSULT THE REFERENTS.

The inner block is then the DLX (file a-DLX.vhd) in figure 3.2, where some blocks are already defined, as the program counter PC, the instruction register IR and the instruction RAM (THAT

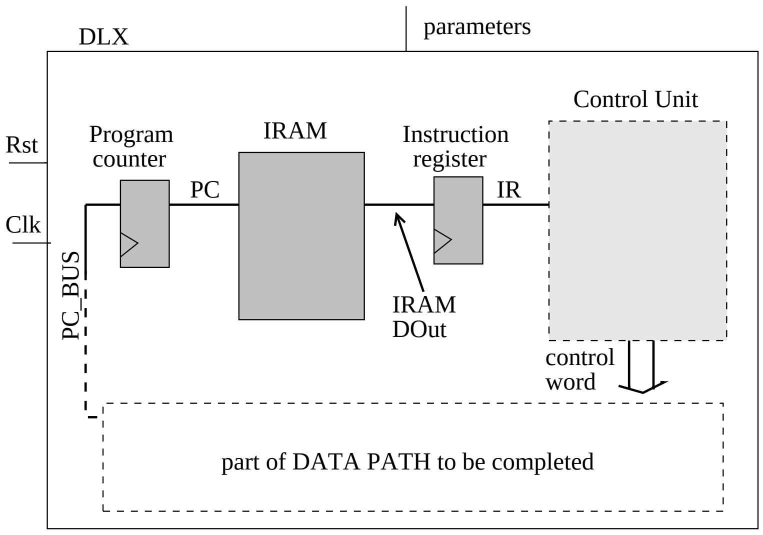


Figure 3.2: DLX_Project_Files/vhd/a-DLX.vhd

IS PLACED HERE FOR SIMPLICITY), while the data path is still missing (TO BE IMPLEMENTED).
You can read the instruction memory file (DLX_Project_Files/vhd/a.c-IRAM.vhd): the important actions derive from a process which reads (at the beginning) file test.asm.mem. What happens after this? Try to figure out by accurately reading file IRAMs.vhd.
For as regard the Control Unit we propose you three different prototype of the control unit (to complete): the hardwired (CU_HW.vhd), the FSM (CU_FSM.vhd) and the microprogrammed (CU_UP.vhd), you can find these files in DLX_Project_Files/vhd/ with thw prefix a.a-, they will be well explained in section 2.2.

3.2 Files organization structures

The files organization is very important in order to work efficiently, without wasting time to search and rename files at the end of the project (when the number bursts).
In order to help us to correct and **EVALUATE** your project YOU MUST name the file with this format:

orderingPrefixLevel0.subLevel.subSubLevel-nameOfComponent_v#OfVersion.vhd

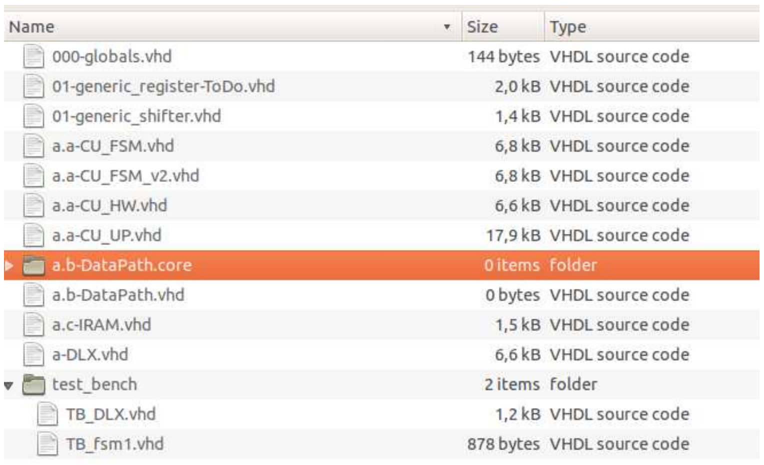
example: **a.b.c-booth_multiplier_v2.vhd**

If the components of a sub part are too much create a folder in order to group all sub components! but follow the same rules that we use for the name, for example:

a.b-DataPath.core

Take all the Test Bench in the folder test_bench with the prefix *TB-*, and the common component at the begin of the folder with numeric prefix, define all the custom type in the file *000-globals.vhd*.

The final result would be something like the figure below (fig. 3.3):



Name	Size	Type
000-globals.vhd	144 bytes	VHDL source code
01-generic_register-ToDo.vhd	2,0 kB	VHDL source code
01-generic_shifter.vhd	1,4 kB	VHDL source code
a.a-CU_FSM.vhd	6,8 kB	VHDL source code
a.a-CU_FSM_v2.vhd	6,8 kB	VHDL source code
a.a-CU_HW.vhd	6,6 kB	VHDL source code
a.a-CU_UP.vhd	17,9 kB	VHDL source code
a.b-DataPath.core	0 items	folder
a.b-DataPath.vhd	0 bytes	VHDL source code
a.c-IRAM.vhd	1,5 kB	VHDL source code
a-DLX.vhd	6,6 kB	VHDL source code
test_bench	2 items	folder
TB_DLX.vhd	1,2 kB	VHDL source code
TB_fsm1.vhd	878 bytes	VHDL source code

Figure 3.3: File organization structure

that's Why?

Because keep ordered the file ordered is useful in phase of simulation and synthesis, and HELP US to APPRECIATE YOUR WORK.

3.3 Testing Tools and BenchMark Methods

From the methodology point of view we are NOT going to write each instruction code MANUALLY in the instruction memory, starting from our assembler program. On the contrary we:

1. Chose the assembler program to be executed:
   - MANY examples are given **DLX_Project_files/asm_example/** in order to start we reccommend you to use the simplest one: **test.asm**.
   - a list with asm mnemonics and their syntax and coding is given in table A and in appendix B
2. Compile it and convert to a readable format:
   - a compiler written in Perl is given; in a terminal type:

     prompt> ./assembler.bin/dlxasm.pl test.asm
   - the compiler generates a binary file with "exe" extension (test.asm.exe)
   - (you can give a look at it using the **hexedit** editor if installed)
   - convert the file format so that it can be easily read by the VHDL control unit:

   prompt> ./assembler.bin/conv2memory test.asm.exe > test.asm.mem
   which can be now read using **more** or **emacs** or any text editor.

Or if you prefer you can use and/or improve the **assembler.sh** script that fuse all previous step.

ACHTUNG ACHTUNG!!!

MANY asm code examples are given in **DLX_Project_files/asm_example/** *folder BUT may be that they doesn't works!!! In order to benchmark your code you have to:*

- *Select a subset of those examples.*
- *Verify and adjust them (make it compatible with your architecture).*
- *Comment each .asm file that you want to use for bench marking.*
- *Report It accurately: explain witch operation use, what is the proven property, comment the output result, time output or optionally document it with a memory dump COMMENTED.*

Chapter 4

How To Document the Project

In this chapter you will find some guidelines on how to write a well documented project report. The advices that we want to give you are of two types:

1. how to organize the report in such a way that it is clear, neither too synthetic nor too long and wordy;
2. some tips on how to use LaTeX to generate the report.

4.1 How the documentation should be like

It is not easy to write a good report that documents in a clear and concise way the work that you have done. Sometimes you are too synthetic, you don't justify the choices that you have made or your documentation is messy. In other words the reader doesn't understand what you have done, how and why.
In other cases you describe too many particulars or you include not relevant part or descriptions. That's a lot of 'bla bla bla' that makes the reader sleep.
So, here is a short list of advices:

- Outline always, as first thing, the **overall situation** giving the main points but not the details; only after having described the general picture delineate the **relevant details**. As an example, let's suppose that you have to describe the ALU of your microprocessor. You create a section called ALU and in it you give the overall description of this unit:
  The Arithmetic Logic Unit designed consists of the following components:

  1. *an adder;*
  2. *a multiplier;*
  3. *a logical unit;*
  4. *. . . .*

  Then you add a subsection for each component of the ALU to describe them more in details;
- **Justify every choice** you made because it is the only way to let the reader understand your design;
- Make use of **lists** since they improve the text readability;

- Use **explanatory images/graphs/schemes** since they are an effective way to explain something that can be complex to describe using only words. Use **"hierarchical" images** that means: draw first a generic block scheme and then depict the details of what's inside each block (see as example figure 4.1). Choose the image size in such a way that every

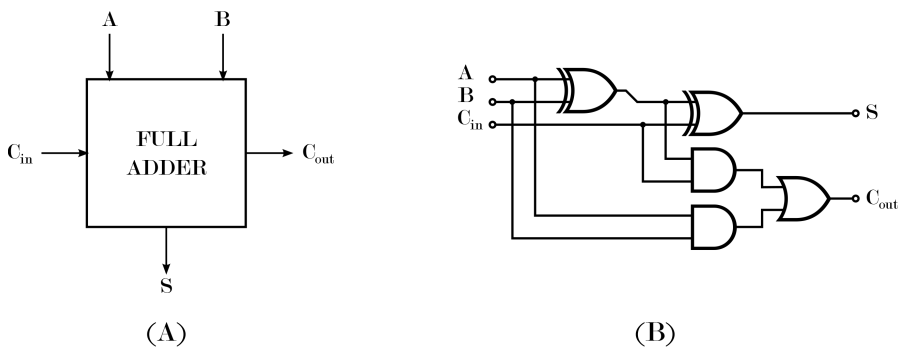


Figure 4.1: (A) Full Adder block representation. (B) Implementation details of the full adder.

detail can be seen clearly and indicate every **signal name** in a readable way.
Please, try to **avoid hand-drawn images** (there are a lot of simple tools, e.g. **Inkscape**, that you can use to draw beautiful images)!!!

- Use **tables** to organize data.

Are you a LaTeX beginner? No problem! In the next section you can find some useful tips that will help you to compose the document. In addition, you have at your disposal a **template to generate the report**: download and unzip the file **DLX_report_template.zip** that you find within the DLX material on Portale della Didattica.

4.2 LaTeX tips

Here you will find some guidelines on how to do basic things with LaTeX. For more information we suggest to see the **on-line documentation** (e.g., LaTeX Wikibooks).
The official "place" where you can find everything about LaTeX is CTAN (Comprehensive TeX Archive Network).

First of all, you need an **editor** to write LaTeX documents. There exist a lot of editors, for example:

- Texmaker which is a multi-platform program;
- TeXShop which is Mac OS X native;
- Kile which is Linux native.

Download and install the editor that you prefer.

Now, take a look at what's inside the folder DLX_report_template. There are:

- A file homebook.tex. You don't have to modify it except for two things: your group ID and your names. Open this file and you will find indications about where to make these changes.

- A subfolder named *chapters* in which you can put the tex files of the chapters that will make up your report; put all the images that you will use in the folder *figures* and all the external files that you may need to include in the text in the folder *files*.
- A subfolder named *appendices* in which you can put the tex files of the appendices; you may need an appendix if, for example, you want to include a piece of code. This subfolder is organized in the same way as the folder *chapters*.

You can see that in the folders *chapters* and *appendices* there are two sample files called chap1.tex and appendix1.tex, respectively.
Now open the file homebook.tex with the editor you chose, change the group ID and your names and then go through it.
At the end you will find a part that says HERE IS WHERE YOU INCLUDE YOUR CHAPTERS. In that point, using the command `\input{./path-to-file/file_name}` you can include your chapter files.
In the point that says HERE IS WHERE YOU INCLUDE YOUR APPENDICES (IF ANY), using the same command shown above you can include the appendix files.
Compile the file and see what happens... If there are no errors in compile time you should see the resulting pdf file.

Now open the file chap1.tex. Here you can find some examples of how to insert images, create tables, write equations, insert code and so on.

Images If you want to insert an image in your document you have to use the following piece of code:

```
\begin{figure}[position_options]
\centering
\includegraphics[size_options]{path-to-figure/figure_name}
\caption{figure_caption}
\label{fig:figure_label}
\end{figure}
```

Editable parameters:

- `position_options`[1]: modify these options to change the position of the image in the document;
- `size_options`[1]: modify these options to change the size of the image;
- `path-to-figure/figure_name`: insert the correct path to the figure and its name (it is not necessary to add the file extension);
- `figure_caption`: write here the caption of the figure;
- `figure_label`: insert here the label of the figure (useful if you want to refer to the figure within the tex document).

[1]Please search on the Internet all the option possibilities and the supported image formats.

Tables Below you can find an example of table.

```
\begin{table}[position_options]
\centering
\begin{tabular}{column_text_options}
\toprule
A B Cin & S & Cout\\
\midrule
0 0 0 & 0 & 0\\
1 0 0 & 1 & 0\\
0 1 0 & 1 & 0\\
1 1 0 & 0 & 1\\
0 0 1 & 1 & 0\\
1 0 1 & 0 & 1\\
0 1 1 & 0 & 1\\
1 1 1 & 1 & 1\\
\bottomrule
\end{tabular}
\caption{table_caption Truth table of a 1-bit full adder.}
\label{tab:table_label}
\end{table}
```

Editable parameters:

- `&`: column separator;
- \\: start a new row;
- \`toprule`: uppermost horizontal line of the table;
- \`midrule`: middle horizontal line of the table (you can add different midrules if you want lines that separate each row of the table);
- \`bottomrule`: lowermost horizontal line of the table;
- `position_options`[2]: modify these options to change the position of the table in the document;
- `column_text_options`[2]: modify these options to change the text alignment (centred, left justified, right justified) in the cells;
- `table_caption`: write here the caption of the table;
- `table_label`: insert here the label of the table (useful if you want to refer to the table within the tex document).

[2]Please search on the Internet all the option possibilities.

Lists Here is how you write a bulleted list:

```
\begin{itemize}
\item first element;
\item second element;
...
\item last element.
\end{itemize}
```

Here is how you write a numbered list:

```
\begin{enumerate}
\item first element;
\item second element;
...
\item last element.
\end{enumerate}
```

Equations Here is how you write an equation that appears to be separate from the main text:

```
Text text text text ...
\begin{equation}
- write the equation here -
\label{eq:eq_label}
\end{equation}
Text continues here ...
```

You can also write in-line equations which appear to be within the text where they are declared:

```
Text text ... $write the equation between dollar symbols$ text continues ...
```

NOTE: please find on-line how you can write mathematical symbols, operators, Greek letters and much more.

Source code inclusion If you want to include some source code in your document you can do it in two ways:

1.
   ```
   \begin{lstlisting}[style=MyC++Style]
   - put your code here -
   \end{lstlisting}
   ```

2. `\lstinputlisting[options]{/path-to-file/source_filename}`

In the second case you directly import the source file so you have to specify the path to the file. Regarding the options, here you can set, for example, the language (VHDL, Assembler, ...). If you specify the language the code keywords will appear highlighted on the output pdf.
Please, check on-line the languages supported and all the other useful options (i.e., line numbering, formatting options, ...) that you can set.
There is another way to set the options which is the use of the following command:

```
\lstset{options}
```

This command can be put anywhere in the tex file and the parameters set using `\lstset` will be valid globally (for all source codes included).

Appendix A

DLX Assembler - CODING and COMPILING

The project comes with an assembler compiler written in Perl language. The instructions to use the compiler are reported in section 3.3

The Opcodes and Func codes used by the compiler are reported in table A. A full explanation of the instruction meaning is at this file (appendix B). For Register-register and Floating-point instructions the Opcode is respectively 0x00 and 0x04, the coding reported in the table for this instructions is the function code (FUNC field). For the other instructions the reported code is the OPCODE of the instruction.

Table A.1: DLX instructions coding

Instruction set coding					
General instructions		Register-register instructions		Floating-point instruction	
Mnemonic	OP-CODE	Mnemonic	FUNC-CODE	Mnemonic	FUNC-CODE
j	0x02	sll	0x04	addf	0x00
jal	0x03	srl	0x06	subf	0x01
beqz	0x04	sra	0x07	multf	0x02
bnez	0x05	add	0x20	divf	0x03
bfpt	0x06	addu	0x21	addd	0x04
bfpf	0x07	sub	0x22	subd	0x05
addi	0x08	subu	0x23	multd	0x06
addui	0x09	and	0x24	divd	0x07
subi	0x0a	or	0x25	cvtf2d	0x08
subui	0x0b	xor	0x26	cvtf2i	0x09
andi	0x0c	seq	0x28	cvtd2f	0x0a
ori	0x0d	sne	0x29	cvtd2i	0x0b
xori	0x0e	slt	0x2a	cvti2f	0x0c
lhi	0x0f	sgt	0x2b	cvti2d	0x0d
rfe	0x10	sle	0x2c	mult	0x0e
trap	0x11	sge	0x2d	div	0x0f
jr	0x12	movi2s	0x30	eqf	0x10
jalr	0x13	movs2i	0x31	nef	0x11
slli	0x14	movf	0x32	ltf	0x12
nop	0x15	movd	0x33	gtf	0x13
srli	0x16	movfp2i	0x34	lef	0x14
srai	0x17	movi2fp	0x35	gef	0x15
seqi	0x18	movi2t	0x36	multu	0x16
snei	0x19	movt2i	0x37	divu	0x17
slti	0x1a	sltu	0x3a	eqd	0x18
sgti	0x1b	sgtu	0x3b	ned	0x19
slei	0x1c	sleu	0x3c	ltd	0x1a
sgei	0x1d	sgeu	0x3d	gtd	0x1b
lb	0x20			led	0x1c
lh	0x21			ged	0x1d
lw	0x23				
lbu	0x24				
lhu	0x25				
lf	0x26				
ld	0x27				
sb	0x28				
sh	0x29				
sw	0x2b				
sf	0x2e				
sd	0x2f				
itlb	0x38				
sltui	0x3a				
sgtui	0x3b				
sleui	0x3c				
sgeui	0x3d				

Appendix B

DLX Instruction Set: Hint and Recall

A few points:

- Bits are numbered from 0 (the most significant bit) to 31 (the least significant bit).
- All transfers are 32 bits unless otherwise specified, with the exception of double precision fp operations which are 64 bit transfers unless otherwise noted.
- All integer operations are on 32-bit integers.
- All assignments to integer register[x] are conditional on x not being zero. Register 0 has a hardwired *zero* value and cannot be modified.
- Single precision floating point is 32 bits and double precision floating point is 64 bits..
- Double register[x] is a 64 bit quantity that represents the same storage as fp register[x] and fp register[x+1]. Only even values of x are allowed (double register addresses are aligned).
- Memory will be stored in big endian format and all effective addresses must be aligned with the data type.

B.0.1 Mnemonics

```
Notation
  Symbol        Meaning

 x_y      bit y of x
 x_y..z   bits y to z of x (right justified)
 x^y       xx....x (x repeated y times)
 x\#\#y       xy (x concatenated with y)
 IR      instruction register
 IAR      interrupt address register
 PC       program counter
 R[rega] integer register[IR_6..10]
 R[regb] integer register[IR_11..15]
 R[regc] integer register[IR_16..20]
 F[frega] fp register[IR_6..10]
 F[fregb] fp register[IR_11..15]
 F[fregc] fp register[IR_16..20]
 D[drega] double register[IR_6..10]
 D[dregb] double register[IR_11..15]
 D[dregc] double register[IR_16..20]
 imm16       value of (IR_16)^16 ## IR_16..31
 uimm16       value of 0^16 ## IR_16..31
 imm26       value of (IR_6)^6 ## IR_6..0
 fps     floating point status bit
 <--      a 32-bit transfer
 <--n      an n-bit transfer
```

```
add
Ex: add r1,r2,r3
R[regc] <-- R[rega] + R[regb]
All are signed integers.


addd
Ex: addd f4,f4,f6
D[dregc] <-- D[drega] + D[dregb]
All are double precision floating point numbers.


addf
Ex: addf f3,f4,f5
F[fregc] <-- F[frega] + F[fregb]
All are single precision floating point numbers.


addi
Ex: addi r5,r2,#5
R[regb] <-- R[rega] + imm16
All are signed integers.


addu
Ex: addu r2,r3,r4
R[regc] <-- R[rega] + R[regb]
All are unsigned integers.


addui
Ex: addui r2,r3,#28
R[regb] <-- R[rega] + uimm16
All are unsigned integers.


and
Ex: and r2,r3,r4
R[regc] <-- R[rega] & R[regb]
All are unsigned integers. Logical ‘and’ is performed on a bitwise basis.


andi
Ex: andi r3,r4,#5
R[regb] <-- R[rega] & uimm16
All are unsigned integers. Logical ‘and’ is performed on a bitwise basis.


beqz
Ex: beqz r1,label
if (R[rega] == 0) PC <-- PC + imm16


bfpf
Ex: bfpf label
if (fps == 0) PC <-- PC + imm16
fps is the floating point status bit.


bfpt
Ex: bfpt label
if (fps == 1) PC <-- PC + imm16
fps is the floating point status bit.
```

```
bnez
Ex: bnez r1,label
if (R[rega] != 0) PC <-- PC + imm16


cvtd2f
Ex: cvtd2f f1,f4
F[fregc] <-- (float) D[drega]
Converts double precision floating point value to single precision floating point
     value.


cvtd2i
Ex: cvtd2i f1,f0
F[fregc] <-- (int) D[drega]
Converts double precision floating point value to integer.


cvtf2d
Ex: cvtf2d f4,f9
D[dregc] <-- (double) F[frega]
Converts single precision float to double.


cvtf2i
Ex: cvtf2i f3,f4
F[fregc] <-- (int) F[frega]
Converts single precision float to integer.


cvti2d
Ex: cvti2d f2,f9
D[dregc] <-- (double) F[frega]
Converts a signed integer to double precision float.


cvti2f
Ex: cvti2f f2,f5
F[fregc] <-- (float) F[frega]
Converts a signed integer to single precision float.


div
Ex: div f2,f2,f3
F[fregc] <-- F[frega] / F[fregb]
All are signed integers.


divd
Ex: divd f4,f4,f6
D[dregc] <-- D[drega] / D[dregb]
All are double precision floats.


divf
Ex: divf f2,f3,f6
F[fregc] <-- F[frega] / F[fregb]
All are single precision floats.


divu
Ex: divu f2,f3,f4
F[fregc] <-- F[frega] / F[fregb]
All are unsigned integers.


eqd
Ex: eqd f2,f4
if (D[drega] == D[dregb]) fps = 1 else fps = 0
Both are double precision floats.
```

```
eqf
Ex: eqf f3,f5
if (F[frega] == F[fregb]) fps = 1 else fps = 0
Both are single precision floats.


ged
Ex: ged f8,f6
if (D[drega] >= D[dregb]) fps = 1 else fps = 0
Both are double precision floats.


gef
Ex: gef f3,f6
if (F[frega] >= F[fregb]) fps = 1 else fps = 0
Both are single precision floats.


gtd
Ex: gtd f8,f6
if (D[drega] > D[dregb]) fps = 1 else fps = 0
Both are double precision floats.


gtf
Ex: gtf f3,f6
if (F[frega] > F[fregb]) fps = 1 else fps = 0
Both are single precision floats.


j
Ex: j label
PC <-- PC + imm26
Unconditionally jumps relative to the PC of the next instruction. imm26 is a
     26-bit signed integer.


jal
Ex: jal label
R31 <-- PC + 4; PC <-- PC + imm26
Saves a return address in register 31 and jumps relative to the PC of the next
     instruction. imm26 is a 26-bit signed integer.


jalr
Ex: jalr r2
R31 <-- PC + 4; PC <-- R[rega]
Saves a return address in register 31 and does an absolute jump to the target
     address contained in R[rega].


jr
Ex: jr r3
PC <-- R[rega]
R[rega] is treated as an unsigned integer. Does an absolute jump to the target
     address contained in R[rega].


lb
Ex: lb r1,40-4(r2)
R[regb] <-- (sign extended) M[imm16 + R[rega]]
One byte of data is read from the effective address computed by adding signed
     integer imm16 and signed integer R[rega]. The byte from memory is then sign
     extended to 32-bits and stored in register R[regb].


lbu
Ex: lbu r2,label-786+4(r3)
R[regb] <-- 0^24 ## M[imm16 + R[rega]]
One byte of data is read from the effective address computed by adding signed
     integer imm16 and signed integer R[rega]. The byte from memory is then zero
     extended to 32 bits and stored in register R[regb].
```

```
ld
Ex: ld f2,240(r1)
D[dregb] <--64 M[imm16 + R[rega]]
Two words of data are read from the effective address computed by adding signed
     integer imm16 and unsigned integer R[rega] and stored in double register
     D[dregb]. This is equivalent to two lf instructions:

F[fregb] <-- M[imm16 + R[rega]]
F[freg(b+1)] <-- M[imm16 + R[rega] + 4]
where F[freg(b+1)] is the next fp register after F[fregb] in sequence, and all
     values are simply copied and not converted.


led
Ex: led f8,f6
if (D[drega] <= D[dregb]) fps = 1 else fps = 0
Both are double precision floats.


lef
Ex: lef f3,f6
if (F[frega] <= F[fregb]) fps = 1 else fps = 0
Both are single precision floats.


lf
Ex: lf f6,76(r4)
F[fregb] <-- M[imm16 + R[rega]]
One word of data is read from the effective address computed by adding signed
     integer imm16 and signed integer R[rega] and stored in fp register F[fregb].


lh
Ex: lh r1,32(r3)
R[regb] <-- (sign extended) M[imm16 + R[rega]]
Two bytes of data are read from the effective address computed by adding signed
     integer imm16 and signed integer R[rega]. The address must be half-word
     aligned. The half-word from memory is then sign extended to 32 bits and stored
     in register R[regb].


lhi
Ex: lhi r3,#-40
R[regb] <-- imm16 ## 0^16
Loads the 16 bit immediate value imm16 into the most significant half of an
     integer register and clears the least significant half.


lhu
Ex: lhu r2,-40+4(r3)
R[regb] <-- 0^16 ## M[imm16 + R[rega]]
Two bytes of data are read from the effective address computed by adding signed
     integer imm16 and signed integer R[rega]. The address must be half-word
     aligned. The half-word from memory is then zero extended to 32 bits and stored
     in register R[regb].


ltd
Ex: ltd f8,f6
if (D[drega] < D[dregb]) fps = 1 else fps = 0
Both are double precision floats.


ltf
Ex: ltf f3,f6
if (F[frega] < F[fregb]) fps = 1 else fps = 0
Both are single precision floats.


lw
Ex: lw r19,label+63(r8)
R[regb] <-- M[imm16 + R[rega]]
```

```
One word is read from the effective address computed by adding signed integer
    imm16 and unsigned integer R[rega] and is stored in R[regb].


movd
Ex: movd f2,f4
D[dregc] <-- D[drega]
Copies two words from double register D[drega] to double register D[dregc].


movf
Ex: movf f1,f2
F[fregc] <-- F[frega]
Copies one word from fp register F[frega] to fp register F[fregc].


movfp2i
Ex: movfp2i r3,f0
R[regc] <-- F[frega]
Copies one word from fp register F[frega] to integer register R[regc].


movi2fp
Ex: movi2fp f0,r3
F[fregc] <-- R[rega]
Copies one word from integer register R[rega] to fp register F[fregc].


movi2s
Ex: movi2s r1
Unspecified
Copies one word from integer register R[rega] to a special register.


movs2i
Ex: movs2i r2
Unspecified
Copies one word from a special register to integer register R[rega].


mult
Ex: mult f2,f3,f4
F[fregc] <-- F[frega] * F[fregb]
All are signed integers.


multd
Ex: multd f2,f4,f6
D[dregc] <-- D[drega] * D[dregb]
All are double precision floats.


multf
Ex: multf f3,f4,f5
F[fregc] <-- F[frega] * F[fregb]
All are single precision floats.


multu
Ex: multu f2,f3,f4
F[fregc] <-- F[frega] * F[fregb]
All are unsigned integers.


ned
Ex: ned f8,f6
if (D[drega] != D[dregb]) fps = 1 else fps = 0
Both are double precision floats.


nef
Ex: nef f3,f6
if (F[frega] != F[fregb]) fps = 1 else fps = 0
```

```
Both are single precision floats.


nop
Ex: nop
Idles one cycle.


or
Ex: or r2,r3,r4
R[regc] <-- R[rega] | R[regb]
All are unsigned integers. Logical ‘or’ is performed on a bitwise basis.


ori
Ex: ori r3,r4,#5
R[regb] <-- R[rega] | uimm16
All are unsigned integers. Logical ‘or’ is performed on a bitwise basis.


rfe
Ex: rfe
Unspecified
Return from exception.


sb
Ex: sb label-41(r3),r2
M[imm16 + R[rega]] <--8 R[regb]_24..31
One byte of data from the least significant byte of register R[regb] is written to
     the effective address computed by adding signed integer imm16 and signed
     integer R[rega].


sd
Ex: sd 200(r4),f6
M[imm16 + R[rega]] <--64 D[dregb]
Two words from double register D[dregb] are written to the effective address
     computed by adding signed integer imm16 and signed integer R[rega].


seq
Ex: seq r1,r2,r3
if (R[rega] == R[regb]) R[regc] <-- 1 else R[regc] <-- 0
All are signed integers.


seqi
Ex: seqi r14,r3,#3
if (R[rega] == imm16) R[regb] <-- 1 else R[regb] <-- 0
All are signed integers.


sf
Ex: sf 121(r3),f1
M[imm16 + R[rega]] <-- F[fregb]
One word from fp register F[fregb] is written to the effective address computed by
     adding signed integer imm16 and signed integer R[rega].


sge
Ex: sge r1,r3,r4
if (R[rega] >= R[regb]) R[regc] <-- 1 else R[regc] <-- 0
All are signed integers.


sgei
Ex: sgei r2,r1,#6
if (R[rega] >= imm16) R[regb] <-- 1 else R[regb] <-- 0
All are signed integers.


sgeu
```

```
Ex: sgeu r1,r3,r4
if (R[rega] >= R[regb]) R[regc] <-- 1 else R[regc] <-- 0
All are unsigned integers.


sgeui
Ex: sgeui r2,r1,#6
if (R[rega] >= uimm16) R[regb] <-- 1 else R[regb] <-- 0
All are unsigned integers.


sgt
Ex: sgt r4,r5,r6
if (R[rega] > R[regb]) R[regc] <-- 1 else R[regc] <-- 0
All are signed integers.


sgti
Ex: sgti r1,r2,#-3000
if (R[rega] > imm16) R[regb] <-- 1 else R[regb] <-- 0
All are signed integers.


sgtu
Ex: sgtu r4,r5,r6
if (R[rega] > R[regb]) R[regc] <-- 1 else R[regc] <-- 0
All are unsigned integers.


sgtui
Ex: sgtui r1,r2,#3000
if (R[rega] > uimm16) R[regb] <-- 1 else R[regb] <-- 0
All are unsigned integers.


sh
Ex: sh 421(r3),r5
M[imm16 + R[rega]] <--16 R[regb]_16..31
Two bytes of data from the least significant half of register R[regb] are written
     to the effective address computed by adding signed integer imm16 and unsigned
     integer R[rega]. The effective address must be halfword aligned.


sle
Ex: sle r1,r2,r3
if (R[rega] <= R[regb]) R[regc] <-- 1 else R[regc] <-- 0
All are signed integers.


slei
Ex: slei r8,r5,#345
if (R[rega] <= imm16) R[regb] <-- 1 else R[regb] <-- 0
All are signed integers.


sleu
Ex: sleu r1,r2,r3
if (R[rega] <= R[regb]) R[regc] <-- 1 else R[regc] <-- 0
All are unsigned integers.


sleui
Ex: sleui r8,r5,#345
if (R[rega] <= uimm16) R[regb] <-- 1 else R[regb] <-- 0
All are unsigned integers.


sll
Ex: sll r6,r7,r11
R[regc] <-- R[rega] << R[regb]_27..31
All are unsigned integers. R[rega] is logically shifted left by the low five bits
     of R[regb]. Zeros are shifted into the least-significant bit.
```

```
slli
Ex: slli r1,r2,#3
R[regb] <-- R[rega] << uimm16_27..31
All are unsigned integers. R[rega] is logically shifted left by the low five bits
     of uimm16. Zeros are shifted into the least-significant bit. (Actually only
     the bottom five bits of R[regb] are used.)


slt
Ex: slt r3,r4,r5
 if (R[rega] < R[regb]) R[regc] <-- 1 else R[regc] <-- 0
 All are signed integers.


 slti
 Ex: slti r1,r2,#22
 if (R[rega] < imm16) R[regb] <-- 1 else R[regb] <-- 0
 All are signed integers.


 sltu
 Ex: sltu r3,r4,r5
 if (R[rega] < R[regb]) R[regc] <-- 1 else R[regc] <-- 0
 All are unsigned integers.


 sltui
 Ex: sltui r1,r2,#22
 if (R[rega] < uimm16) R[regb] <-- 1 else R[regb] <-- 0
 All are unsigned integers.


 sne
 Ex: sne r1,r2,r3
 if (R[rega] != R[regb]) R[regc] <-- 1 else R[regc] <-- 0
 All are signed integers.


 snei
 Ex: snei r4,r5,#89
 if (R[rega] != imm16) R[regb] <-- 1 else R[regb] <-- 0
 All are signed integers.


 sra
 Ex: sra r1,r2,r3
 R[regc] <-- (R[rega]_0)^R[regb] ## (R[rega]>>R[regb])_R[regb]..31
 R[rega] and R[regc] are signed integers. R[regb] is an unsigned integer. R[rega]
      is arithmetically shifted right by R[regb]. The sign bit is shifted into the
      most-significant bit. (Actually uses only the five low order bits of R[regb].)


 srai
 Ex: srai r2,r3,#5
 R[regb] <-- (R[rega]_31)^uimm16 ## (R[rega]>>uimm16)_uimm16..31
 R[rega] and R[regc] are signed integers. uimm16 is an unsigned integer. R[rega]
      is arithmetically shifted right by R[regb]. The sign bit is shifted into the
      most-significant bit. (Actually uses only the five low order bits of uimm16.)


 srl
 Ex: srl r15,r2,r3
 R[regc] <-- R[rega] >> R[regb]_27..31
 All are unsigned integers. R[rega] is arithmetically shifted right by R[regb].
      Zeros are shifted into the most significant bit.


 srli
 Ex: srli r1,r2,#5
 R[regb] <-- R[rega] >> uimm16_27..31
 All are unsigned integers. R[rega] is arithmetically shifted right by uimm16.
      Zeros are shifted into the most significant bit.
```

```
sub
Ex: sub r3,r2,r1
Ex: R[regc] <-- R[rega] - R[regb]
All are signed integers.


subd
Ex: subd f2,f4,f6
D[dregc] <-- D[drega] - D[dregb]
All are double precision floats.


subf
Ex: subf f3,f4,f6
F[fregc] <-- F[frega] - F[fregb]
All are single precision floats.


subi
Ex: subi r15,r16,#964
R[regb] <-- R[rega] - imm16
All are signed integers.


subu
Ex: subu r3,r2,r1
R[regc] <-- R[rega] - R[regb]
All are unsigned integers.


subui
Ex: subui r1,r2,#53
R[regb] <-- R[rega] - uimm16
All are unsigned integers.


sw
Ex: sw 21(r13),r6
M[imm16 + R[rega]] <-- R[regb]
One word from integer register R[regb] is written to the effective address
     computed by adding signed integer imm16 and unsigned integer R[rega].


trap
Ex: trap #3
Execute trap with number in immediate field.
Saves state and jumps to an operating system procedure located at an address in
     the interrupt vector table. In our systems, this is simulated by calling the
     procedure corresponding to the trap number.


xor
Ex: xor r2,r3,r4
R[regc] <-- F[rega] XOR R[regb]
All are unsigned integers. Logical 'xor' is performed on a bitwise basis.


xori
Ex: xori r3,r4,#5
R[regb] <-- R[rega] XOR uimm16
All are unsigned integers. Logical 'xor' is performed on a bitwise basis.
```