

Laboratory 7

7. Single-Cycle MIPS CPU Design (4): 16-bits version – One clock cycle per instruction

7.1. Objectives

Study, design, implement and test

- **Instruction Execute Unit for the 16-bit Single-Cycle MIPS CPU**
- **Testing of the Arithmetical-Logical Instructions**

Familiarize the students with

- Single-Cycle CPU design: Defining the instructions / writing the test program (MIPS assembly language, machine code)
- Xilinx® ISE WebPack
- Digilent Development Boards (**DDB**)
 - [Digilent Basys Board – Reference Manual](#)
 - [Digilent Basys 2 Board – Reference Manual](#)
 - [Digilent Basys 3 Board – Reference Manual](#)

7.2. Reduced Size MIPS Processor Description

(!) Read Lectures 3 and 4 in order to understand the works needed in this laboratory.

Remember that an instruction execution cycle (lecture 4) has the following phases:

- | | | |
|---------|---|------------------------------------|
| • IF | – | Instruction Fetch |
| • ID/OF | – | Instruction Decode / Operand Fetch |
| • EX | – | Execute |
| • MEM | – | Memory |
| • WB | – | Write Back |

Your own Single-Cycle MIPS 16 processor implementation (that you will continue and hopefully finish in this laboratory) will be partitioned in 5 (five) components (new entities). These components will be declared and instantiated in the “test_env” project.

The utility of this implementation will be understood in the future laboratories, when you will implement the pipeline version of your 16-bit MIPS processor!

In this laboratory you will design, VHDL description, implement and test the Instruction Execute Unit of your own single cycle MIPS 16 processor.

The data-path of the processor (32-bit version), including the control unit and the necessary control signals, is presented in the next figure. In order to reduce the complexity of the data-path, the control signals were not explicitly connected, but rather they can be easily identified by their names.

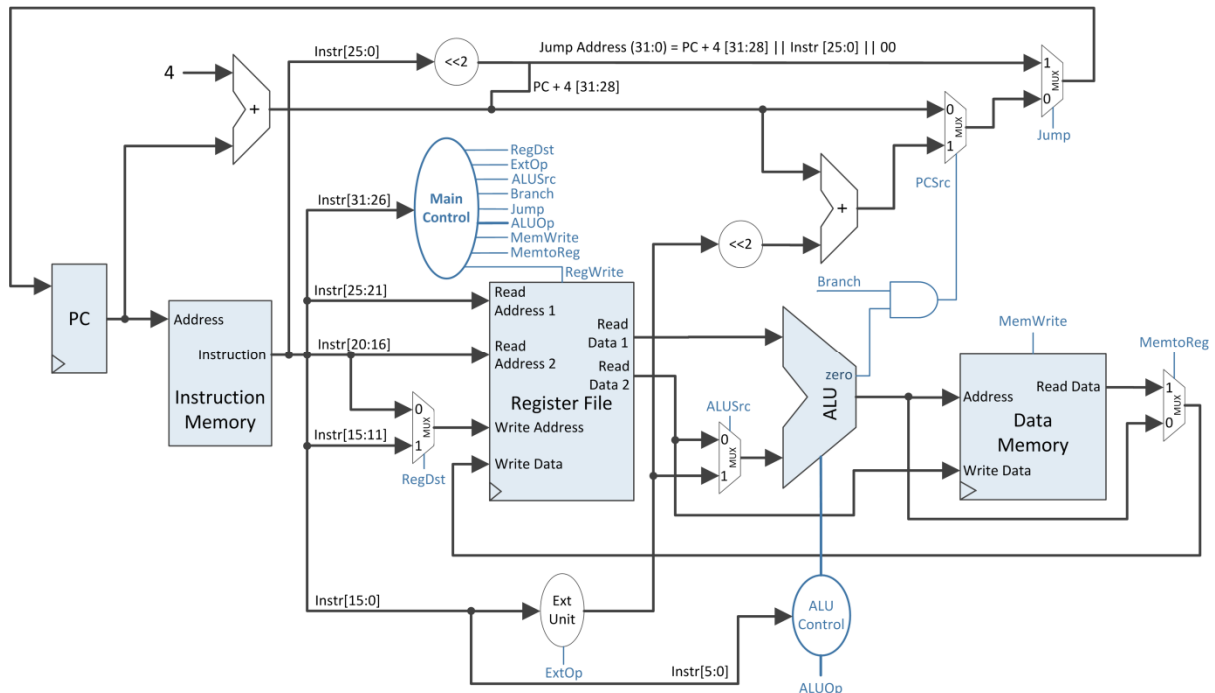


Figure 7-1: MIPS 32 Single-Cycle Data-Path + Control

As a reminder, the instruction formats for your MIPS 16 processor are presented below:

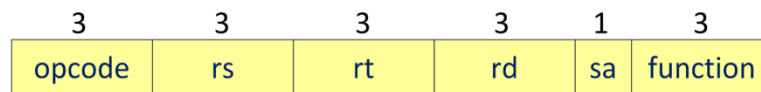


Figure 7-2: R-type Instruction format



Figure 7-3: I-type Instruction format



Figure 7-4: J-type Instruction format

The Execute Unit (Ex) consists in the following components:

- Arithmetic Logic Unit (ALU)
- ALU Control
- Multiplexer
- Shift Left 2 and adder for branch target address computation

Please refer to laboratories 2 and 4 for the characteristics of these components for your Single-Cycle MIPS 16 processor.

The data-path of the Instruction Execute Unit is presented in Figure 7-5.

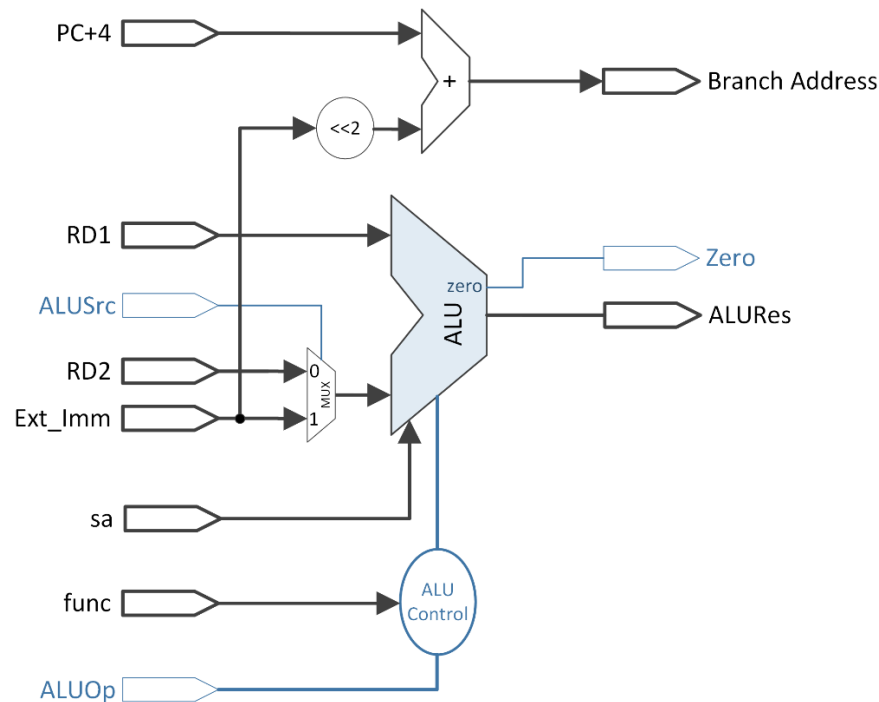


Figure 7-5: Instruction Execute Data-Path for MIPS 32

The EX unit provides, as output, the ALU Result used for writing the result of arithmetical/ logical instructions in the Register File or used as the address for the Data Memory in the case of lw and sw instructions. In addition, the ALU provides another output, the Zero Signal, which indicates whether the result of the ALU is equal to zero or not (if the result is equal to zero the signal will have as value 1, otherwise 0). For simplicity of your future, first version, pipeline implementation, the EX unit also includes the branch target address computation.

The inputs of the Ex unit are:

- Next Sequential Instruction Address (PC+4)
- 32-bit Read Data 1 (RD1)
- 32-bit Read Data 2 (RD2)
- 32-bit Extended Immediate (Ext_Imm)

- 6-bit function field (func)
- 5-bit shift amount (sa)
- Control Signals:
 - ALUSrc – selects between Read Data 2 and Extended Immediate as input to the second port of the ALU
 - ALUOp – ALU operation code provided by the Main Control Unit

The outputs of the Ex unit are:

- 32-bit Branch target address
- 32-bit ALU result (ALURes)
- 1-bit Zero signal

The meaning of the control signals:

- ALUSrc = 0 → the Read Data 2 signal is the second input for the ALU
- ALUSrc = 1 → the Extended Immediate signal is the second input for the ALU
- ALUOp → is defined by the Main Control Unit according to the operations implemented in the ALU.

The branch target address is computed with the following formula:

Branch Address \leftarrow PC + 4 + S_Ext(Imm) \ll 2;

The Zero signal together with the Branch Control Signal is used in order to select between the normal sequential execution of the program (PC + 4) or the branch target address.

ALU Control Unit defines the ALU operations encoded in the ALUCtrl control signal. For I-type instructions, the encoding of the ALUCtrl is simply defined by the ALUOp signal. For R-type instructions, the value of ALUCtrl is defined by the fixed value ALUOp and the function field.

7.3. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- All the assignments from the laboratories 4, 5, 6 completed
- The instruction fetch unit implemented and tested on the Digilent Development Board.
- The instruction decode unit implemented and tested on the Digilent Development Board.
- Xilinx project with “test_env” including the IF and ID units (Laboratory 6 Assignment 6.3.3)

Attention: If the homework from the previous laboratories is not completed, you will receive a 1 for this and all future laboratories until the homework is done without the possibility of any corrections to the mark!

7.3.1. Instruction Execute Unit design

Taking into account the instruction execution data-path from Figure 7-5 design a new component (new entity) in the “test_env” project for your own single-cycle MIPS 16. All the data fields are 16-bits wide.

The Ex entity will contain the hardware components described in Figure 7-5 that will not be implemented with other components (use behavioral VHDL description)

Use one line of code for the branch target address computation (no shift required only addition).

Use a process (with a case statement) for the ALU implementation as in laboratory 2. The 1-bit shift amount is used only for the logical and / or arithmetic shift operations.

Use a process (with a case statement) for the implementation of the ALU Control. The encoding of the ALU Ctrl control signal is dependent on your own 15 instructions and defines the arithmetic-logical operations implemented by the ALU.

Attention! Be careful when transforming the data fields from Figure 7-5 (MIPS 32) into your own single-cycle MIPS 16 implementation.

7.3.2. Testing of the Instruction Execute Unit

Instantiate the Execution Unit in the “test_env” project. At this moment, you will connect the output of the Execution Unit (AluRes) to the Write Data port of the ID Unit (WD input)

You have to test all your arithmetical / logical instructions on the Digilent Development Board: Add, Sub, Shift left, Shift Right, And, Or, Addi, etc. Make sure that all the instructions perform correctly.

All the signals present on the data-path must be connecting to the SSD, i.e. the outputs of the IF, ID and EX units. Use switches in order to control the display on the SSD (multiplexor):

- $sw(7:5) = 000 \rightarrow$ display the instruction on the SSD
- $sw(7:5) = 001 \rightarrow$ display the next sequential PC ($PC + 1$ output) on the SSD
- $sw(7:5) = 010 \rightarrow$ display the RD1 signal on the SSD
- $sw(7:5) = 011 \rightarrow$ display the RD2 signal on the SSD
- $sw(7:5) = 100 \rightarrow$ display the Ext_Imm signal on the SSD

- $sw(7:5) = 101 \rightarrow$ display the ALURes signal on the SSD
- $sw(7:5) = 111 \rightarrow$ display the WD signal on the SSD

On the LEDs, you will display the control signals from the Main Control Unit. You have 8 x 1-bit control signals and ALUOp. Use another switch to control the display on the LEDs:

- $sw(0) = 0 \rightarrow$ Display the 1-bit control signals on the LEDs. The order of the control signals is your own choice.
- $sw(0) = 1 \rightarrow$ Display the n-bit ALUOp signal on the LEDs (the rest of LEDs will have the value '0' for now)

Trace your program (without any memory operation or conditional /unconditional jump) on the Digilent Development Board instruction by instruction. Be sure that all the control signals / data fields are correct.

7.4. References

- [1] Computer Architecture Lectures 3 & 4 slides.
- [2] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", 5th edition, ed. Morgan-Kaufmann, 2013.
- [3] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design: A Quantitative Approach", 5th edition, ed. Morgan-Kaufmann, 2011.
- [4] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
- [5] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
- [6] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
 - Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratory 8

8. Single-Cycle MIPS CPU Design (5): 16-bits version – One clock cycle per instruction

8.1. Objectives

Study, design, implement and test

- **Memory Unit for the 16-bit Single-Cycle MIPS CPU**
- **Write Back Unit for the 16-bit Single-Cycle MIPS CPU**
- **Other necessary connections for branch / jump address computation**
- **Test the Single-Cycle MIPS CPU**

Familiarize the students with

- Single-Cycle CPU design: Defining the instructions / writing the test program (MIPS assembly language, machine code)
- Xilinx® ISE WebPack
- Digilent Development Boards (**DDB**)
 - [Digilent Basys Board – Reference Manual](#)
 - [Digilent Basys 2 Board – Reference Manual](#)
 - [Digilent Basys 3 Board – Reference Manual](#)

8.2. Reduced Size MIPS Processor Description

(!) Read Lectures 3 and 4 in order to understand the works needed in this laboratory.

Remember that an instruction execution cycle (lecture 4) has the following phases:

- | | | |
|---------|---|------------------------------------|
| • IF | – | Instruction Fetch |
| • ID/OF | – | Instruction Decode / Operand Fetch |
| • EX | – | Execute |
| • MEM | – | Memory |
| • WB | – | Write Back |

Your own Single-Cycle MIPS 16 processor implementation (that you will continue and hopefully finish in this laboratory) will be partitioned in 5 (five) components (new entities). These components will be declared and instantiated in the “test_env” project.

The utility of this implementation will be understood in the future laboratories, when you will implement the pipeline version of your 16-bit MIPS processor!

In this laboratory you will design, VHDL description, implement and test the Memory Unit, Write Back Unit and the rest of the connections of your own single cycle MIPS 16 processor.

The data-path of the processor (32-bit version), including the control unit and the necessary control signals, is presented in the next figure. In order to reduce the complexity of the data-path, the control signals were not explicitly connected, but rather they can be easily identified by their names.

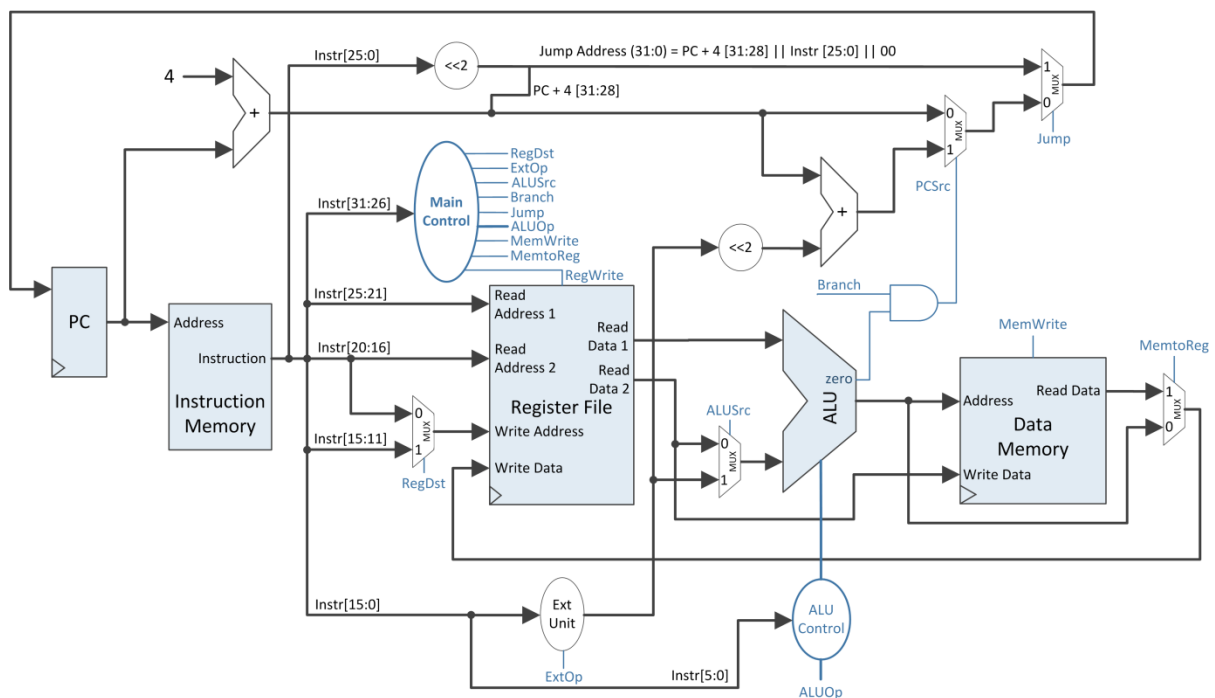


Figure 8-1: MIPS 32 Single-Cycle Data-Path + Control

As a reminder, the instruction formats for your MIPS 16 processor are presented below:

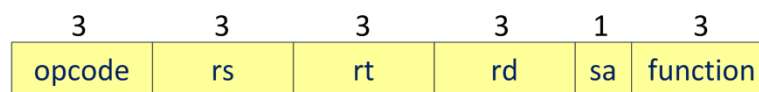


Figure 8-2: R-type Instruction format

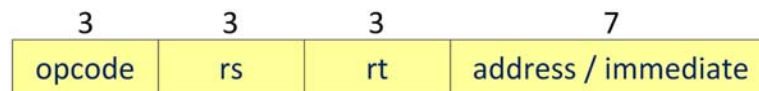


Figure 8-3: I-type Instruction format

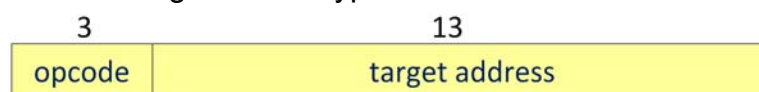


Figure 8-4: J-type Instruction format

The Memory Unit consist in the following component

- Data Memory

The data-path of the Memory Unit is presented in Figure 8-5.

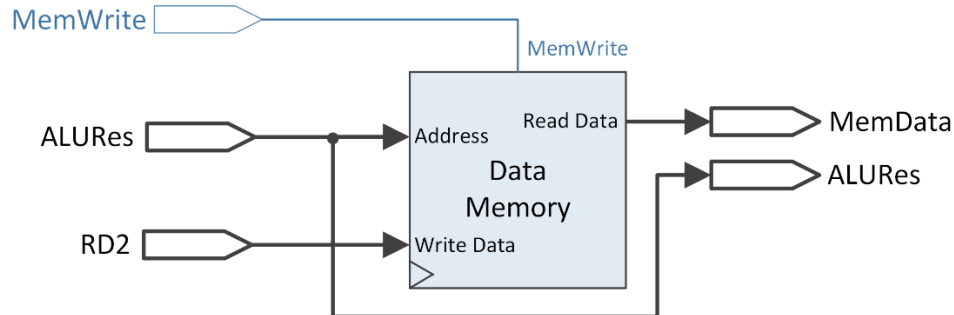


Figure 8-5: Memory Unit Data-Path for MIPS 32

The Data Memory is a RAM with asynchronous read and synchronous write operations. A similar RAM implementation (with synchronous read) was done in laboratory 3.

The inputs of the Memory Unit:

- The clock signal (for the Data Memory Writes)
- 32-bit ALURes signal – consists in the address for the Data Memory
- 32-bit RD2 signal – the second output of the Register File (used only for store word instructions) is the Write Data field for the Data Memory
- MemWrite control signal

The outputs of the Memory Unit:

- 32-bit MemData, the Read Data from the Data Memory (used only for load word instructions).
- 32-bit ALURes, this signal is also the result of the arithmetic-logical instructions that must be stored in Register File, so it is also fed as output for the Memory Unit and input to the Write Back Unit.

The only control signal present in this stage is the MemWrite Control Signal.

- MemWrite = 0 → Nothing is written in the Data Memory.
- MemWrite = 1 → The RD2 signal is written in the Data Memory at the address indicated by the ALURes signal.

The Write Back unit is simply the last multiplexor from Figure 8-1. The rest of the components are the AND gate for generating the PCSrc control signal, the jump address computation.

The control signal for the Write Back multiplexor is MemtoReg and identifies what value is fed to the Write Data port of the Register File in the ID state:

- MemtoReg = 0 → the ALURes signal is the input to the Write Data port of the Register File.
- MemtoReg = 1 → the MemData is the input to the Write Data port of the Register File.

The PCSrc signal identifies if the current instruction is a Branch instruction and if the content of the rs and rt registers are the same; i.e. $RF[rs] == RF[rt]$.

$PCSrc \leftarrow \text{Branch and Zero}$;

8.3. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- All the assignments from the laboratories 4, 5, 6, 7 completed
- The instruction fetch unit implemented and tested on the Digilent Development Board.
- The instruction decode unit implemented and tested on the Digilent Development Board.
- The instruction execute unit implemented and tested on the Digilent Development Board
- The memory unit implemented and tested on the Digilent Development Board
- Xilinx project with “test_env” including the IF, ID, EX, MEM units (Laboratory 7 Assignment 7.3.3)

Attention: If the homework from the previous laboratories is not completed, you will receive a 1 for this and all future laboratories until the homework is done without the possibility of any corrections to the mark!

8.3.1. Memory Unit Design

Describe a new component (new entity) for the Memory Unit. Use the RAM implementation from laboratory 3 and change the read operation to an asynchronous one. Write only with processes inside the Memory Unit.

Instantiate the Memory Unit in the “test_env” project. Connect all the signals from the Memory Unit in the data-path. The MemWrite signal should be validated with an output of the MPG component as it was previously implemented for the writing in the Register File (RegWrite signal).

8.3.2. Adding the Write Back Unit and the jump address computation

Add the write back multiplexor for your own MIPS processor. Use only one line of code to implement this multiplexor.

Complete your own MIPS processor implementation with the jump address computation and the PCSrc signal computation. Complete all the necessary connections for the data-path as in Figure 8-1 (jump address, branch target address, write back in the register file, etc.).

Test the LW, SW, BEQ and Jump Instructions for your MIPS processor.

8.3.3. Testing it ALL: You own Single-Cycle MIPS 16 processor

At this moment, you should have all the components from the data-path implemented in the “test_env” project.

All the signals present on the data-path must be connecting to the SSD, i.e. the outputs of the IF, ID, EX, M and WB units. Use switches in order to control the display on the SSD (multiplexor):

- $sw(7:5) = 000 \rightarrow$ display the instruction on the SSD
- $sw(7:5) = 001 \rightarrow$ display the next sequential PC ($PC + 1$ output) on the SSD
- $sw(7:5) = 010 \rightarrow$ display the RD1 signal on the SSD
- $sw(7:5) = 011 \rightarrow$ display the RD2 signal on the SSD
- $sw(7:5) = 100 \rightarrow$ display the Ext_Imm signal on the SSD
- $sw(7:5) = 101 \rightarrow$ display the ALURes signal on the SSD
- $sw(7:5) = 110 \rightarrow$ display the MemData signal on the SSD
- $sw(7:5) = 111 \rightarrow$ display the WD signal on the SSD

On the LEDs, you will display the control signals from the Main Control Unit. You have 8 x 1-bit control signals and ALUOp. Use another switch to control the display on the LEDs:

- $sw(0) = 0 \rightarrow$ Display the 1-bit control signals on the LEDs. The order of the control signals is your own choice.
- $sw(0) = 1 \rightarrow$ Display the n-bit ALUOp signal on the LEDs (the rest of LEDs will have the value '0' for now)

If needed for debugging the processor on the Digilent Development Board you can additionally display other signals on the SSD / LEDs: branch target address, jump address, ALUCtrl, etc.

Now trace your program on the Digilent Development Board instruction by instruction. Be sure that all the control signals / data fields are correct.

Present your Single-Cycle MIPS implementation to your TA.

8.4. References

[1] Computer Architecture Lectures 3 & 4 slides.

- [2] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", 5th edition, ed. Morgan–Kaufmann, 2013.
- [3] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design: A Quantitative Approach", 5th edition, ed. Morgan-Kaufmann, 2011.
- [4] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
- [5] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
- [6] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
 - Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratory 9

9. Pipeline MIPS CPU Design: 16-bits version

9.1. Objectives

Study, design, implement and test

- **MIPS 16 CPU, pipeline version**

Familiarize the students with

- Pipeline CPU design
- Xilinx® ISE WebPack
- Digilent Development Boards (**DDB**)
 - [Digilent Basys Board – Reference Manual](#)
 - [Digilent Basys 2 Board – Reference Manual](#)
 - [Digilent Basys 3 Board – Reference Manual](#)

9.2. Transforming the MIPS 16 Single-Cycle CPU to a Pipeline CPU

! You must attend/read lecture 8 in order to fully understand the Pipeline CPU

Remember that an instruction execution cycle (lecture 4) has the following phases:

- | | | |
|---------|---|------------------------------------|
| • IF | – | Instruction Fetch |
| • ID/OF | – | Instruction Decode / Operand Fetch |
| • EX | – | Execute |
| • MEM | – | Memory |
| • WB | – | Write Back |

The data-path of the single-cycle processor (32-bit version), including the control unit and the necessary control signals, is presented in the next figure. In order to reduce the complexity of the data-path the control signals were not explicitly connected, but rather they can be easily identified by their names.

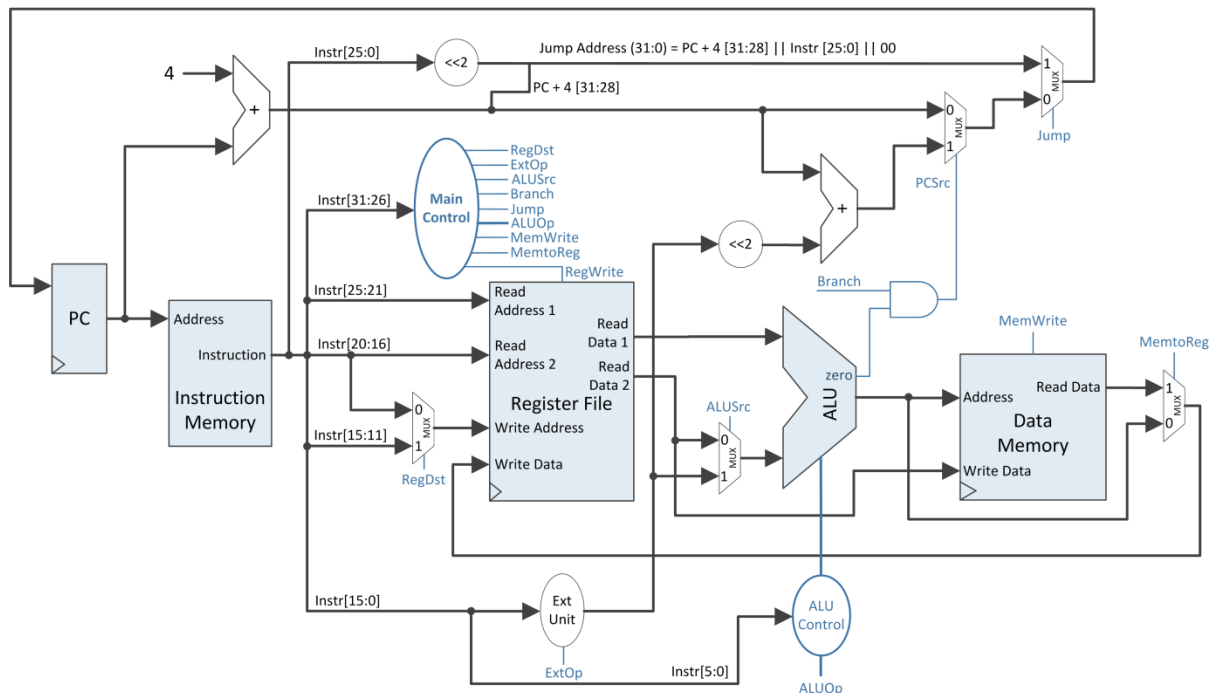


Figure 9-1: MIPS 32 Single-Cycle Data-Path + Control

As a reminder, the instruction formats for your MIPS 16 processor are presented below:

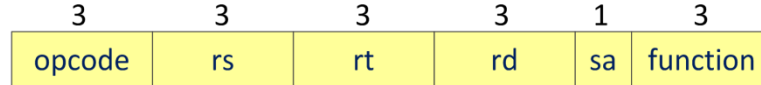


Figure 9-2: R-type Instruction format

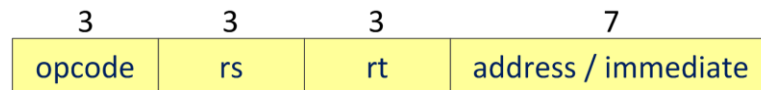


Figure 9-3: I-type Instruction format

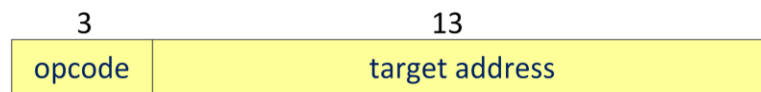


Figure 9-4: J-type Instruction format

The main issue with the single-cycle MIPS CPU is the length of the critical path, for the load word instruction (see lecture 04). The necessary time for transmitting the data along the critical path must be covered by the clock cycle time. This results in a long cycle time (slow clock).

In order to reduce the clock cycle time, the solution is to partition the data-path along the critical path with rising edge triggered registers (D flip-flops). These registers are inserted between the MIPS 32 functional units that coincide with the instruction execution phases: IF, ID, EX, MEM, WB. In this manner, one can simultaneously execute at most 5 instructions, each of them executing one of the five execution phases. The pipeline execution units are also referred to as **stages**.

The data-path together with the control unit for the pipelined MIPS 32 CPU is presented in Figure 9-5.

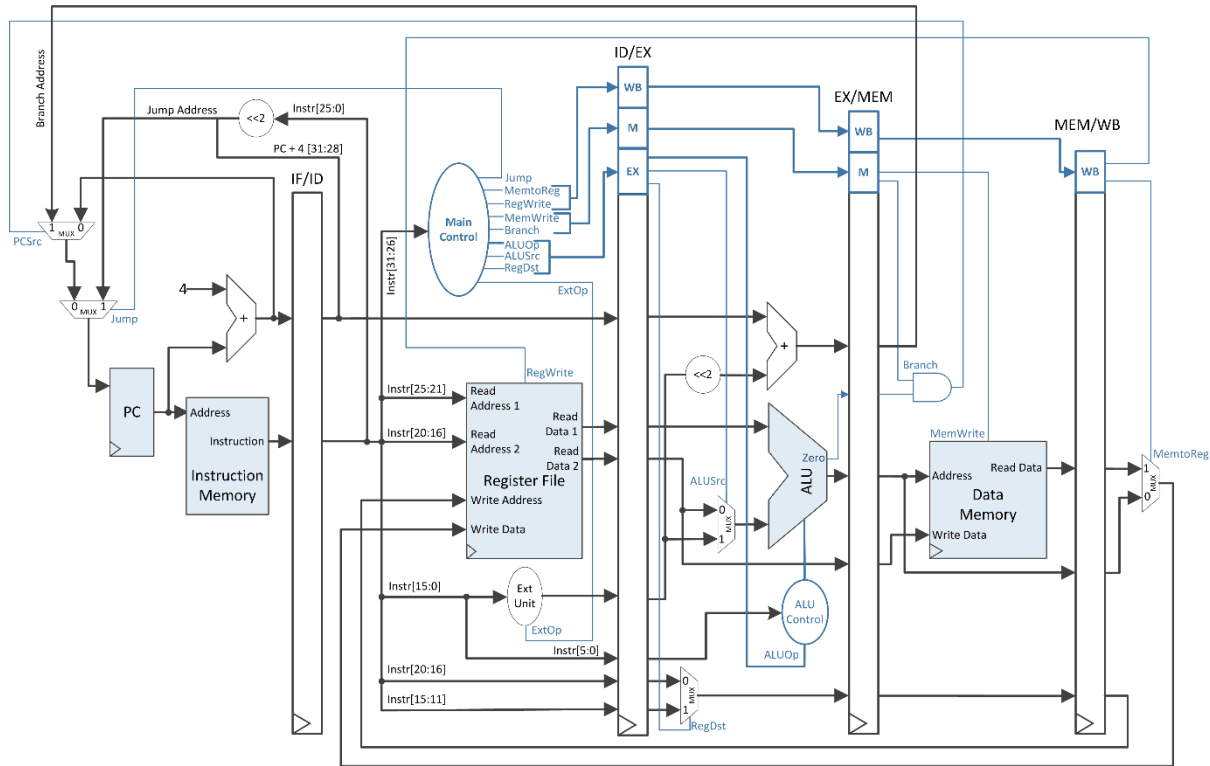


Figure 9-5: MIPS 32 Pipeline Data-Path + Control, obtained from the partitioning of the Single-Cycle Data-Path

Each intermediate register will be referred depending on its position between the pipeline stages. The register between the IF stage and the ID stage is IF/ID, the one between ID and EX is ID/EX, etc.

The role of these intermediate registers is to hold the intermediate results of the instruction execution in order to provide these results to the next stage, in the next clock cycle.

Furthermore, the execution on the data-path depends on the control signals values, which are specific for each instruction. So, through the intermediate registers (starting with the ID/EX register) the control signals will also be provided for the next stages. The control signals are symbolically grouped after the stage name where they belong.

The control signals are transmitted together with the intermediate results until the stages where they are needed.

Lecture 8 explains in more detail the design of the pipeline CPU; the details presented so far represent the necessary knowledge for transforming your own MIPS 16 single-cycle CPU into a pipeline one.

One notable difference between the two data-paths is that the multiplexer used for selecting the write address for the Register File is placed in EX, not in ID as in the single-cycle CPU case. There are 2 possible solutions:

- a) Leave it in the ID stage. In this case, the `RegDst` signal will not be transmitted through ID/EX and will be connected directly from the control unit.
- b) Move it to the EX stage, modifying the input / output ports of the ID and EX units, and transmit the `RegDst` signal according to the presented pipeline data-path.

Observation: The `MemRead` signal will be ignored, as in the single-cycle case.

9.3. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- Xilinx project with “test_env” including the complete and correct implementation of the single-cycle MIPS 16 CPU.

9.3.1. Verify the MIPS 16 CPU design

Before you begin transforming your single-cycle processor into a pipeline one, generate the *.bit file and investigate the clock frequency of your processor: **Processes → Design Summary/Reports**. Open **Detailed Reports → Synthesis Report**. In the synthesis report, locate the section about the clock frequency, similarly with the following text:

TIMING REPORT

...

Timing Summary:

...

Minimum period: 13.284ns (Maximum Frequency: **75.280MHz**)

...

Write down the frequency of your single-cycle processor, so that you can compare it with the frequency of the pipeline version.

If you have not completed the testing of the single-cycle processor, you must complete it now, before you begin the pipeline implementation.

The pipeline implementation must start from a fully functional single-cycle MIPS 16 version.

9.3.2. Design of the intermediate registers (paper and pencil)

For each intermediate register identify the fields that it must store, taking into account your own MIPS 16 implementation.

Use the data-path from Figure 9-5 (!) but keep in mind your MIPS 16 processor's features: 16 bits, not 32; based on your own chosen instructions the data-path may contain additional elements.

For example, for the first intermediate register, IF/ID, one must memorize the following two fields (generic name according to the stage they belong):

- IF.PC+1 – 16 bits
- IF.Instruction – 16 bits

It results that the IF/ID register should contain 32-bits.

Similarly describe the ID/EX, EX/MEM, MEM/WB registers.

When describing the fields of the next intermediate registers, take a look at the associated functional units (the inputs unit and the destination unit respectively: example for IF/ID the IF and ID units respectively) in the laboratories 5, 6, 7 and 8. Identify the fields from the input/output ports of the functional units. This step will be used in the next assignment.

9.3.3. Describe the intermediate registers in VHDL

For this assignment, you will work in the “test_env” entity (where the components of the processor are instantiated and connected together).

Attention: When introducing the new pipeline registers, some small modifications in your functional units may appear. You will easily identify these modifications by carefully studying Figure 9-5 and the particular data-paths for each functional unit of the MIPS 16 processor – laboratories 5, 6, 7, 8. For example, the ID unit needs an additional input port for the write address of the Register File, address that will come from the last pipeline register MEM/WB.

You will not declare new entities for the pipeline registers. For each register, you have to declare a signal of appropriate length (according to the previous assignment) and the behavior of the pipeline register will be described with one process (synchronous data transfer on the rising edge of the clock signal). In order to ease the testing of the processor each register will be controlled with an enable signal (the same MPG output used for validating the writing in the PC register).

You will realize the partitioning of the single-cycle data-path with the pipeline registers and make the necessary correct connections in the mapping of the functional units.

For example, the value of the `RegWrite` control signal will be transmitted through the MEM/WB register and will be mapped at the input of the ID Unit.

Pipeline register example (one can also use concatenation): for IF/ID you must declare a 32-bit signal `RegIF_ID` and describe the behavior of the register in one process:

On rising edge of the clock

`RegIF_ID(31..16) <= PC+1;`

`RegIF_ID(15..0) <= Instruction;`

Where `PC+1` and `Instruction` are the outputs of the IF unit.

Alternatively, you can declare new signals and concatenate the stage name to the signal name like:

On rising edge of the clock

`PC_ID <= PC + 1;`

`Instruction_ID <= Instruction;`

9.3.4. Homework – Paper and Pencil

Draw your own MIPS 16 pipeline CPU data-path together with the control unit and control signals.

9.4. References

- [1] Computer Architecture Lectures 3, 4 & 8 slides.
- [2] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", 5th edition, ed. Morgan-Kaufmann, 2013.
- [3] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design: A Quantitative Approach", 5th edition, ed. Morgan-Kaufmann, 2011.
- [4] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
- [5] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
- [6] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
 - Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratory 10

10. Pipeline MIPS CPU Design (2): 16-bits version

10.1. Objectives

Study, design, implement and test

- **MIPS 16 CPU, pipeline version with the modified program without hazards**

Familiarize the students with

- Pipeline CPU design
- Xilinx® ISE WebPack
- Digilent Development Boards (**DDB**)
 - [Digilent Basys Board – Reference Manual](#)
 - [Digilent Basys 2 Board – Reference Manual](#)
 - [Digilent Basys 3 Board – Reference Manual](#)

10.2. Transforming the MIPS 16 Single-Cycle CPU to a Pipeline CPU

! You must attend/read lecture 8 in order to fully understand the Pipeline CPU

Remember that an instruction execution cycle (lecture 4) has the following phases:

- | | | |
|---------|---|------------------------------------|
| • IF | – | Instruction Fetch |
| • ID/OF | – | Instruction Decode / Operand Fetch |
| • EX | – | Execute |
| • MEM | – | Memory |
| • WB | – | Write Back |

The data-path of the single-cycle processor (32-bit version), including the control unit and the necessary control signals, is presented in the next figure. In order to reduce the complexity of the data-path the control signals were not explicitly connected, but rather they can be easily identified by their names.

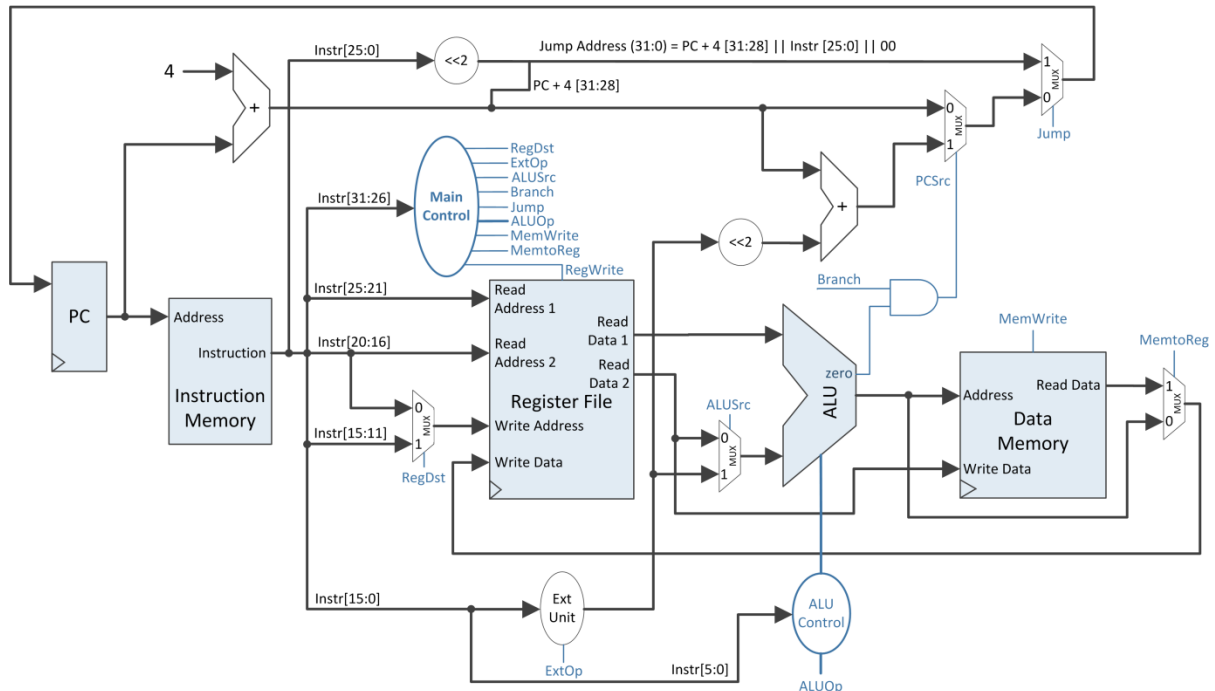


Figure 10-1: MIPS 32 Single-Cycle Data-Path + Control

The main issue with the single-cycle MIPS CPU is the length of the critical path, for the load word instruction (see lecture 04). The necessary time for transmitting the data along the critical path must be covered by the clock cycle time. This results in a long cycle time (slow clock).

In order to reduce the clock cycle time, the solution is to partition the data-path along the critical path with rising edge triggered registers (D flip-flops). These registers are inserted between the MIPS 32 functional units that coincide with the instruction execution phases: IF, ID, EX, MEM, WB. In this manner, one can simultaneously execute at most 5 instructions, each of them executing one of the five execution phases. The pipeline execution units are also referred to as **stages**.

The data-path together with the control unit for the pipelined MIPS 32 CPU is presented in Figure 10-2.

Each intermediate register will be referred depending on its position between the pipeline stages. The register between the IF stage and the ID stage is IF/ID, the one between ID and EX is ID/EX, etc.

The role of these intermediate registers is to hold the intermediate results of the instruction execution in order to provide these results to the next stage, in the next clock cycle.

Furthermore, the execution on the data-path depends on the control signals values, which are specific for each instruction. So, through the intermediate registers (starting with the ID/EX register) the control signals will also be provided for the next stages. The control signals are symbolically grouped after the stage name where they belong.

The control signals are transmitted together with the intermediate results until the stages where they are needed.

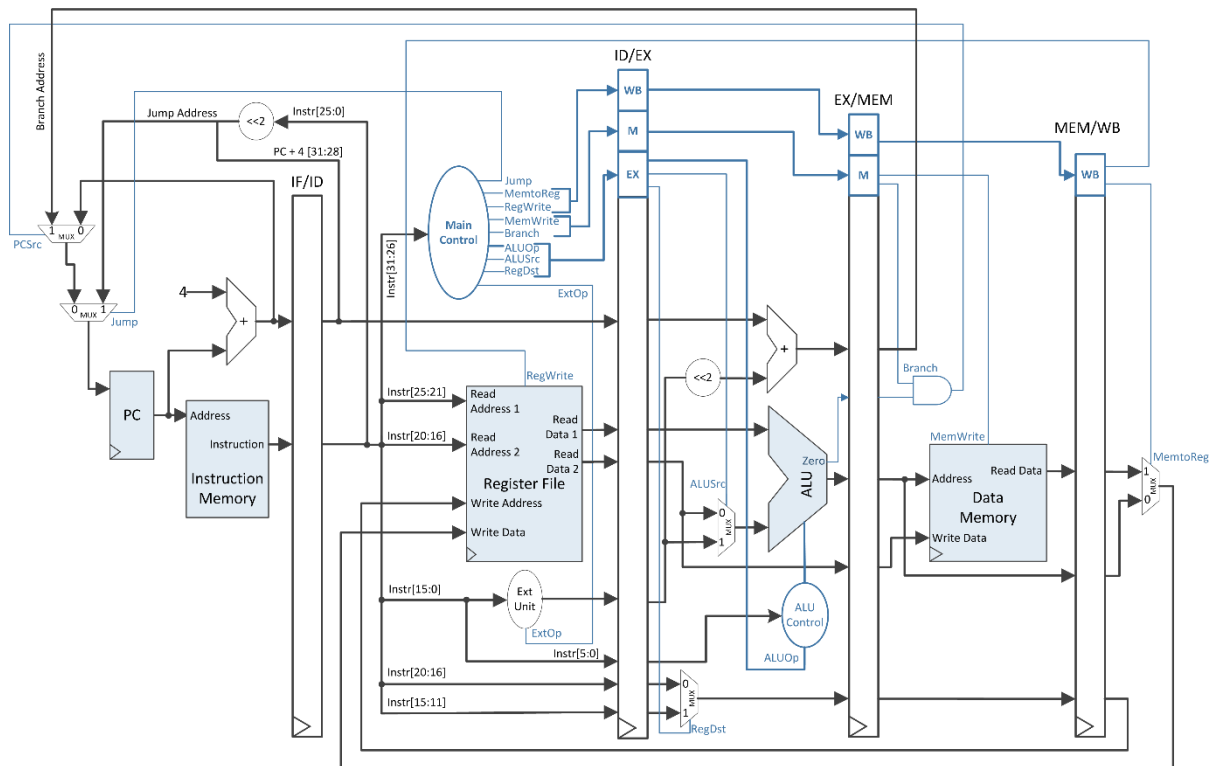


Figure 10-2: MIPS 32 Pipeline Data-Path + Control, obtained from the partitioning of the Single-Cycle Data-Path

Lecture 8 explains in more detail the design of the pipeline CPU; the details presented so far represent the necessary knowledge for transforming your own MIPS 16 single-cycle CPU into a pipeline one.

One notable difference between the two data-paths is that the multiplexer used for selecting the write address for the Register File is placed in EX, not in ID as in the single-cycle CPU case. There are 2 possible solutions:

- c) Leave it in the ID stage. In this case, the **RegDst** signal will not be transmitted through ID/EX and will be connected directly from the control unit.
- d) Move it to the EX stage, modifying the input / output ports of the ID and EX units, and transmit the **RegDst** signal according to the presented pipeline data-path.

Observation: The **MemRead** signal will be ignored, as in the single-cycle case.

10.3. Hazards in MIPS

Hazards are situations in which an instruction cannot be executed in the next clock period. The hazard can be classified as:

1. **Structural Hazards (resource dependency)**
 - 2 instructions try to use the same resource simultaneously for different purposes → resource constraints
2. **Data Hazards (data dependency)**
 - Attempt to use data before it is ready (available)
 - For an instruction in the ID phase, the operands might still be processed in other pipeline stages
3. **Control Hazards (condition and control dependency)**
 - The branch decision and branch target address are not known until the MEM stage. The jump address is computed in the ID stage.
 - Pipelining of jumps, branches and other instructions that modify the sequential flow of the program

These hazards have been thoroughly presented during lecture 8 (you are encouraged to read them!). Optimal solutions (in hardware) are relying on forwarding and stalling the pipeline (see the lecture notes for reference). For your MIPS 16 pipeline implementation, you should implement the software solution, modifying your program such that the data and control hazards are avoided.

The basic change in your program should be the following: introduce NoOp (No Operation) instructions between the instructions where the hazard exists.


NoOp instruction should not change anything in your processor (ex. sll \$0, \$0, 0; add \$0, \$0, \$0, etc.)

10.3.1. Structural Hazards

Structural hazards occur when instructions from two different pipeline stages are trying to use the same resource in the same cycle.

Special attention should be given to the structural hazard that can occur when two instructions at distance of 3 are using the same register (from the RF). In the following example, we presume that instr1 and instr2 do not have hazard with other instructions.

Structural hazard at RF	
add \$1, \$1, \$2	
instr1	
instr2	
add \$3, \$1, \$4	



Pipeline diagram (in each clock cycle we present the pipeline stage for each instruction):

Instr\Clk	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	...
add \$1, \$1, \$2	IF	ID	EX	MEM	WB				
instr1		IF	ID	EX	MEM	WB			
instr2			IF	ID	EX	MEM	WB		
add \$3, \$1, \$4				IF	ID	EX	MEM	WB	

During clock cycle CC5, the new value of \$1, generated by the first instruction, is in WB stage, being unwritten yet in RF. Therefore, in ID stage, the 4th instruction will read the old value of \$1, in cycle CC6 EX receiving the incorrect value.

There are 2 possible solutions:

1. Recommended: Modify RF block such that the writing is done in the middle of the clock cycle (test the falling edge – $\text{clk} = 0$ & $\text{clk}'\text{event}$). In this case, the RF read (being asynchronous), in the second part of CC5 the correct value of \$1 occurs and it is propagated forward to EX at CC5-CC6 transition.
2. Introduce a NoOp instruction

Without Hazard
add \$1, \$1, \$2
instr1
instr2
NoOp
add \$3, \$1, \$4

Attention! In the following, it is assumed that you have chosen the 1st option. Otherwise, introduce an extra NoOp where necessary.

10.3.2. Data Hazards

Data hazards (Read After Write or Load Data Hazard) occur when the current instruction use as source(s) the register that will be written by other instructions that are still executing in the pipeline.

(!) In order to establish where these hazards occur you need to draw the pipeline diagram and to understand how pipelining is done (when operands are read).

The following example contains most of the data hazards that might occur in your pipelined MIPS.

Instr. Nb.	Program
1	add \$1, \$2, \$3
2	add \$3, \$1, \$2
3	add \$4, \$1, \$2
4	add \$5, \$3, \$2
5	lw \$3, 5(\$5)
6	add \$4, \$5, \$3
7	sw \$3, 6(\$5)
8	beq \$3, \$4, -6

Hazard identification process is solved with NoOp insertions, starting from first instruction to the last one. Example:

Instr\Clk	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
add \$1, \$2, \$3	IF	ID	EX	MEM	WB(\$1)				
add \$3, \$1, \$2		IF	ID(\$1)	EX	MEM	WB(\$3)			
add \$4, \$1, \$2			IF	ID(\$1)	EX	MEM	WB		
add \$5, \$3, \$2				IF	ID(\$3)	EX	MEM	WB(\$5)	
lw \$3, 5(\$5)					IF	ID(\$5)	EX	MEM	WB(\$3)

Instr\Clk	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	...
add \$5, \$3, \$2	IF	ID(\$3)	EX	MEM	WB(\$5)				
lw \$3, 5(\$5)		IF	ID(\$5)	EX	MEM	WB(\$3)			
add \$4, \$5, \$3			IF	ID(\$3, \$5)	EX	MEM	WB(\$4)		
sw \$3, 6(\$5)				IF	ID(\$3)	EX	MEM	WB	
beq \$3, \$4, -6					IF	ID(\$4)	EX	MEM	WB

Hazards are solved iteratively, starting from the first occurrence. Solving a hazard between 2 successive instructions implicitly solves the hazards between the first instruction and the instruction at distance +2. Example: between instruction 1 and 2, and 1 and 3, there is a RAW hazard (after \$1). Hazard between 1 and 2 is solved first, delaying instruction 2 with 2 cycles (it should have ID on cycle CC5) => 2 NoOp. Therefore, all following instructions will be delayed with 2 cycles, so the hazard between 1 and 3 is also resolved.

There is a RAW hazard between the 2nd and 4th instructions, after \$3, which can be solved by delaying with 1 cycle, inserting a NoOp after 2 or before 4.

All other hazards are being solved, resulting the following program:

Instr. Nb.	Program
1	add \$1, \$2, \$3
2	NoOp
3	NoOp
4	add \$3, \$1, \$2
5	NoOp
6	add \$4, \$1, \$2
7	add \$5, \$3, \$2
8	NoOp
9	NoOp
10	lw \$3, 5(\$5)
11	NoOp
12	NoOp
13	add \$4, \$5, \$3
14	NoOp
15	sw \$3, 6(\$5)
16	beq \$3, \$4, -6

10.3.3. Control Hazards

Control hazards occur at instructions that alter the sequential flow of the program, when the next sequential instructions that follow (3 for BEQ and 1 for J) are implicitly executed.

For conditional jump instructions (beq, bne, etc.), the next 3 instructions will implicitly be executed, being already in the pipeline. Therefore, a simple (but not efficient) solution is to insert 3 NoOp's.

For unconditional jumps (j, jal, etc.), based on the data-path from Figure 5-5, these instructions are computing the jump address (and writing it in the PC register) in the ID stage. It means that only next instruction starts the execution, so one NoOp needs to be inserted after the j instruction. A better solution would be to insert the instruction that is after j before it, with condition that this instruction is not also a jump instruction (j, beq, etc.)

10.4. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- Xilinx project with "test_env" including the complete and correct implementation of the pipeline MIPS 16 CPU.

10.4.1. Verify the MIPS 16 Pipeline CPU design

You can evaluate the critical path by checking the clock frequency: Go to **Processes** → **Design Summary/Reports**. Open **Detailed Reports** → **Synthesis Report** and watch the section related to clock signal.

You should notice an increase in frequency (of 30-50%) caused by the pipelining of your MIPS (compared to the one observed in laboratory 9). One can observe that the increase in speed is not proportional to the number of pipeline stages. There is a multitude of reasons for that: stages are not balanced, the resulting circuit depends on the board's technology, the memories are implemented as distributed RAMs, etc.

10.4.2. Program analysis and hazard removal (paper and pencil)

Based on the example in section 3, identify the hazards in your program. Insert NoOp instructions where such an instruction is needed. Draw the pipeline diagram for at least 5 successive instructions in your program (for all, if there are not any hazards).

Note: By introducing the NoOp's you will need to adjust the addresses for the jump instructions in your program.

Modify the (assembly) program in the instruction memory

10.4.3. Test and evaluate the MIPS 16 pipeline

Test your design on the FPGA board. You have 2 options:

- a. If your pipeline implementation was correct, without any mapping mistakes etc., then watching your final results is enough (results should be identical with your single cycle implementation).
- b. If the results are different, then you should trace your program step-by-step.

Use the same display procedure as the one used for your single-cycle MIPS (with the multiplexor on switches for selecting different data to be displayed on the SSD). It is important to understand that now your outputs (for your switches configuration) will not be the same as in the single-cycle implementation. You have 5 instructions in the pipeline; some of them will be NoOp.

You can display the control signals on the LEDs. Use the delayed control signals, i.e. the control signals delayed to the stage where they are used.

If necessary, display other signals/change the displayed signals, from different stages, on the SSD.

10.4.4. Hardware optimizations for the MIPS Pipeline CPU (optional).

If you have finished and tested your MIPS pipeline CPU, you can modify your solution in order to implement the following components of the complete pipeline processor:

- a. Hazard detection unit
- b. Forwarding unit
- c. Move the branch in the ID stage
- d. Hardware stalls for the LW (RAW hazard), BEQ and J instructions.

In the end, you should have a complete pipeline implementation, as it is presented in the lecture material.

10.5. References

- [1] Computer Architecture Lectures 3, 4 & 8 slides.
- [2] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", 5th edition, ed. Morgan–Kaufmann, 2013.
- [3] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design: A Quantitative Approach", 5th edition, ed. Morgan-Kaufmann, 2011.
- [4] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
- [5] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
- [6] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
 - Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratory 11

11. Finite State Machines and Serial Communication

11.1. Objectives

Study, design, implement and test

- **Finite State Machines**
- **Serial Communication**

Familiarize the students with

- Xilinx® ISE WebPack
- Digilent Development Boards (**DDB**)
 - [Digilent Basys Board – Reference Manual](#)
 - [Digilent Basys 2 Board – Reference Manual](#)
 - [Digilent Basys 3 Board – Reference Manual](#)

11.2. Theoretical Background

11.2.1. Finite State Machines

A finite state machine or FSM is a model of behavior composed of a finite number of states, transitions between those states, and actions. A finite state machine is used to describe an abstract model of a control unit. XST proposes a large set of templates to describe FSMs. By default, XST tries to distinguish FSMs from VHDL or Verilog code, and apply several state encoding techniques to obtain better performance or less area.

XST supports the following state encoding techniques:

- Auto – the best suited encoding algorithm for each FSM.
- One-hot – associate one code bit and one flip-flop per state. At a given clock cycle during operation, one and only one bit of the state variable is asserted. Only two bits toggle during a transition between two states. One-hot encoding is appropriate with most FPGA targets where a large number of flip-flops are available. It is also a good alternative when trying to optimize speed or to reduce power dissipation.
- Gray – guarantees that only one bit switches between two consecutive states. It is appropriate for controllers exhibiting long paths without branching. In

addition, this coding technique minimizes hazards and glitches. Very good results can be obtained when implementing the state register with T flip-flops.

- Compact – consists of minimizing the number of bits in the state variables and flip-flops. Compact encoding is appropriate when trying to optimize area.
- Johnson – like Gray, it shows benefits with state machines containing long paths with no branching.
- Sequential – consists of identifying long paths and applying successive radix two codes to the states on these paths. Next state equations are minimized.
- Speed1 – oriented for speed optimization. The number of bits for a state register depends on the particular FSM, but generally, it is greater than the number of FSM states.
- User – original encoding specified in the HDL file.

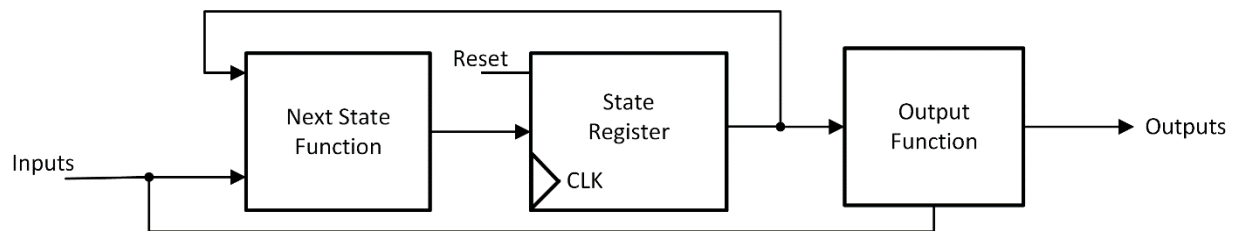


Figure 11-1: FSM Representation Incorporating Mealy and Moore Machines

When describing a finite state machine in VHDL you may have several processes (1, 2 or 3) depending upon how you consider and decompose the different parts of the preceding model. Appendix 7 (adapted from [1]) describes the VHDL finite state machine implementations.

11.2.2. Serial Communication – UART

Serial communication is the transmission or reception of data one bit at a time. Today's computers generally address data in bytes or some multiple thereof. A serial port is used to convert each byte to a stream of ones and zeroes as well as to convert streams of ones and zeroes to bytes. The serial port contains an electronic chip called a Universal Asynchronous Receiver/Transmitter (UART) that actually does the conversion. Serial transmission of digital information (bits) through a single wire or other medium is much more cost effective than parallel transmission through multiple wires.

When transmitting a byte, the UART first sends a START BIT followed by the data (generally 8 bits, but could be 5, 6, 7, or 8 bits), followed by STOP BITS. The sequence is repeated for each byte sent.

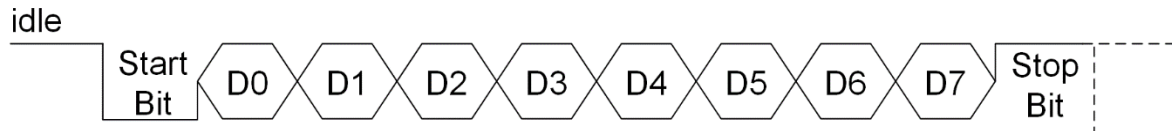


Figure 11-2: Serial Transmission Timing Diagram

Serial transmission does not involve a clock signal. The information is included in the baud rate (**number of bits per second**). Common baud rates are 2400, 4800, 9600 and 19200. This means that a bit transmitted through the serial line is valid for a given time period (the inverse of the baud rate).

The start bit is always 0, the data bits are transmitted with the LSB (least significant bit) first and MSB (most significant bit) last and the stop bit is always 1. In serial communication, the stop bit duration can have multiple values: 1, 1.5 or 2 bit periods in length. Besides the synchronization provided by the use of start and stop bits, an additional bit called a parity bit may optionally be transmitted along with the data. A parity bit affords a small amount of error checking, to help detect data corruption that might occur during transmission. One can choose even parity, odd parity, mark parity, space parity or none at all. When even or odd parity is being used, the number of marks (logical 1 bits) in each data byte is counted, and a single bit is transmitted following the data bits, to indicate whether the number of 1-bits just sent is even or odd.

The data sent through serial communication is encoded using ASCII codes (Appendix 8). Assume we want to send the letter 'A' over the serial communication channel. The binary representation of the letter 'A' is 01000001 (0x41_{hex}). Remembering that bits are transmitted from least significant bit (LSB) to most significant bit (MSB), the bit stream transmitted would be as follows for the line characteristics 8 bits, no parity, 1 stop bit, 9600 baud: **LSB (0 1 0 0 0 0 1 0 1) MSB**. This represents (Start Bit) (Data Bits) (Stop Bit). For a binary two-level signal, a data rate of one bit per second is equivalent to one Baud. To calculate the actual byte transfer rate, simply divide the baud rate by the number of bits that must be transferred for each byte of data. In the case of the above example, each character requires 10 bits to be transmitted for each character. As such, at 9600 baud, up to 960 bytes can be transferred in one second.

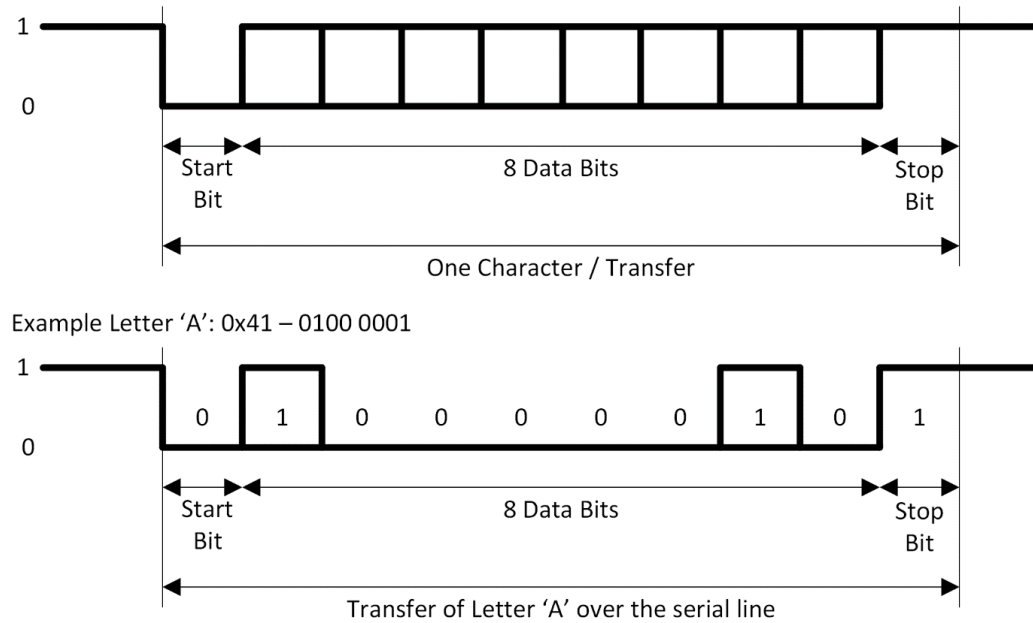


Figure 11-3: Serial Transmission Example (8 data bits, no parity)

For accurate serial communication, on the receiving end, an oversampling scheme is commonly used to locate the middle position of the transmitted bits, i.e., where the actual sample is taken. The most common oversampling rate is 16 times the baud rate. Therefore, each serial bit is sampled 16 times but only one sample is saved.

Each UART contains a shift register which is the fundamental method of conversion between serial and parallel forms.

11.3. Laboratory Assignments

11.3.1. USB-UART

For Basys1 & 2 Development boards: Read the [Pmod USB-UART](#) reference manual. The figure below shows the connection of the USB-UART peripheral module to the FPGA board.

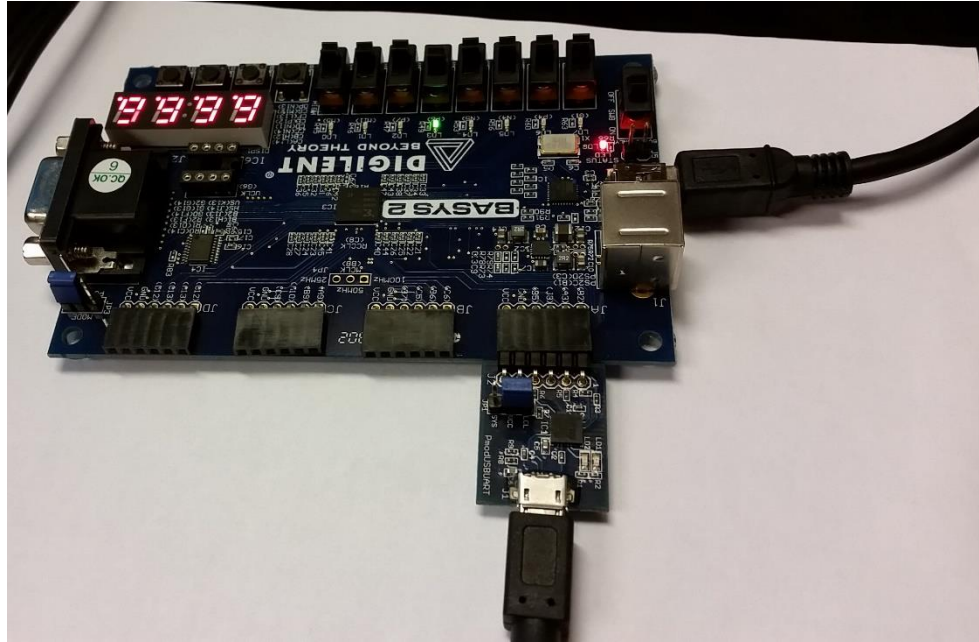


Figure 11-4: Pmod USB-UART connection to the FPGA board

Use the USB-Mini USB cable to power the board and the USB-Micro USB cable for serial data communication.

For Basys3 Development board: Your board is already equipped with an **USB UART Bridge** (FTDI chip) for serial communication. Refer to the Basys 3 reference manual for more details.

Download and open the [HTERM](#) terminal program. Alternatively, you can use the hyper-terminal software available in windows or download any other terminal software known to you / available on the web.

You need to define the RX (input) and TX (output) ports into your “test_env” project and in the UCF/XDC file. Use your board’s reference manual to locate the correct pin numbers. Attention: The TX of the FPGA board is the RX of the Pmod USB-UART module / FTDI chip and the RX of the FPGA board is the TX of the Pmod USB-UART module.

11.3.2. Serial Transmit FSM

Design a baud rate generator that would ensure a 9600 baud rate (9600 bits per second) communication over the serial cable. Use a counter to generate the BAUD_ENable signal (generate a ‘1’ every bit time interval).

Baud rate generation:

- For 25 MHz, clock period ~40 ns, input clock must be divided by 2604.
- For 50 MHz, clock period ~20 ns, input clock must be divided by 5208.

- For 100 MHz, clock period ~10 ns, input clock must be divided by 10416.

Define a new entity for the transmission FSM. The next figure presents the ports of this entity.

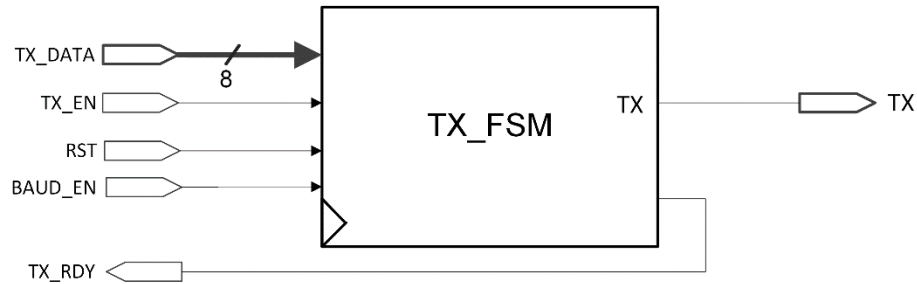


Figure 11-5: TX_FSM Entity Description

The detailed FSM implementation is presented in the figure below. A state transition is triggered only in the clock cycle when BAUD_ENable is '1'. This ensures that a bit will be valid for the baud rate period. The BIT_CNT is a signal with the functionality of a counter inside the FSM; it holds the current transmitting bit value. It should be incremented in the bit state and should be reset after each serial transfer (you can do that in the idle state, or in all states except the bit state).

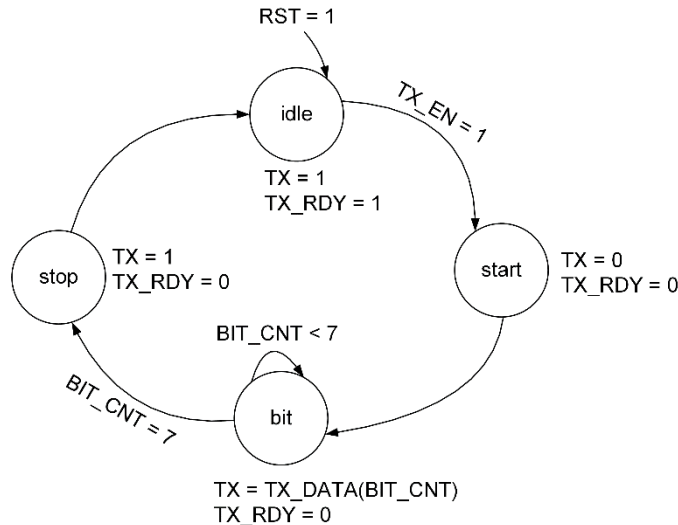


Figure 11-6: TX_FSM Implementation

Write the VHDL code and implement in the “test_env” project the TX_FSM state machine. Use a FSM with 2 or 3 processes (see appendix 7). Test the communication between the FPGA board and the PC. The parameters of the serial communication are: 1 start bit, 8 data bits, 1 stop bit, no parity bit, 9600 baud rate. Make sure that these settings are also configured in the HTERM / hyper-terminal application.

In order to test the serial transmission from the FPGA board to the PC, connect the TX_DATA input to the switches, the TX_EN signal to a MPG enable, RST to '0' or another MPG enable. Make sure that the switches show a valid ASCII code.

Define the correct methodology of asserting the TX_EN signal in order to initiate a single serial data transfer (use a D flip-flop with a set and a reset).

11.3.3. I/O from the MIPS CPU

Connect the TX_FSM into your own MIPS processor implementation. At this point, you are allowed to use your finished and complete processor (single-cycle or pipeline).

You have to send 16-bits of data from your MIPS processor to the PC. Depending on the result of your program, define what field you will send (register with the final result, memory location, etc.).

Example:

When your program has finished execution the result is in R7 and the PC is 0x0020. Add a new instruction to your program: `addi R7, R7, 0`. Define a 16-bit register whose value will be written from the RD1/ALURes signal, write it in this register (write enable with the value of the PC) and initiate the serial transfer.

Remember that when sending over the serial line the 8-bits represent an ASCII character, hence you are required to make 4 transfers in order to send the alphanumerical encoding of the 4 x 4-bit hexadecimal value (use a decoder/ROM to generate the 8-bit ASCII representation for a hexadecimal value).

Define the methodology to send the 16-bit data over the serial line. Use the TX_RDY signal to control the 4 serial transfers.

11.4. References

- [1] XST User Guide
- [2] Digilent Basys Board – Reference Manual
- [3] Digilent Basys 2 Board – Reference Manual
- [4] Digilent Basys 3 Board – Reference Manual
- [5] Digilent Pmod USB-UART – Reference Manual
- [6] <http://www.asciitable.com/>
- [7] <http://www.der-hammer.info/terminal/>

Laboratory 12

12. Finite State Machines and Serial Communication (2)

12.1. Objectives

Study, design, implement and test

- **Finite State Machines**
- **Serial Communication**

Familiarize the students with

- Xilinx® ISE WebPack
- Digilent Development Boards (**DDB**)
 - [Digilent Basys Board – Reference Manual](#)
 - [Digilent Basys 2 Board – Reference Manual](#)
 - [Digilent Basys 3 Board – Reference Manual](#)

12.2. Theoretical Background

Oversampling mechanism for UART Receive

When transmitting a byte, the UART first sends a START BIT followed by the data (general 8 bits, but could be 5, 6, 7, or 8 bits), followed by STOP BITS. The sequence is repeated for each byte sent.

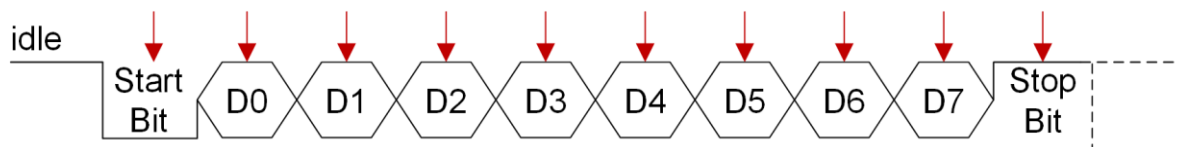


Figure 12-1: Timing Diagram for serial transmission (8-bit Data Example). The red arrows indicate when the bits of data should be read at the receiver.

Serial transmission does not involve a clock signal. The information is included in the baud rate (**number of bits per second**). Common baud rates are 2400, 4800, 9600 and 19200. This means that a bit transmitted through the serial line is valid for a given time period (the inverse of the baud rate). More details on the transmission over the serial line can be found in the previous laboratory.

When receiving a UART packet, one must read (sample) the input signal and extract the data bits sent over the serial line bit by bit. At a first glance, the sample rate for the receiver should coincide with the sample rate (baud rate) of the transmitter; i.e. the rate at which the data was sent. However, this is **WRONG** and can yield in bad transfers at the receiver end, due to imperfect synchronizations (**the receiver and the transmitter are in two different clock domains, the baud rate is generated independent at the receiver and the transmitter, asynchronous communication – no common clock signal**) between the receiver and the transmitter (double reading the same bit, missing the start bit, not reading the first bit and reading the sign bit, etc.). The frequency at which such events can occur depends on the difference between the sampling rates of the transmitter and receiver. Even if the differences would be very small at a significant number of samplings for successive bits, the error in communication can occur. For example, a difference of 0.1% between the two sampling rates, when transmitting 1000 bits the error appears once. When we perform the serial transfer with 10-bits per character (1 start bit, 8 data bits and 1 stop bit) it results that one character from 100 will be falsely received.

This problem is tackled using oversampling: the input receive signal is read (sampled) at a higher rate than the one used at the transmitter. This permits the detection of the middle of the start bit interval, thus allowing the data bits to be read approximately in the middle of the bit interval, thus eliminating the risk of gaps and receiving false data. For each new character, the middle of the start bit will be determined so this is the only synchronization mechanism used between the receiver and the transmitter.

Oversampling rates are multiple of the transmitter baud-rate: 2, 4, 8, etc. The most usual oversampling rate is 16 times the baud rate of the sender. Each bit that is received over the serial line is sampled (read) 16 times, but only one of the samples is saved (the middle one). The maximum delay for detecting the start bit is 1/16 from the bit interval.

Any UART circuit contains a shift register that is used for converting the received serial data into its parallel form.

12.3. Laboratory Assignments

12.3.1. Serial Receive FSM

Design a baud rate generator that would ensure a 9600 baud rate (9600 bits per second) communication over the serial cable. Use a counter to generate the BAUD_ENable signal (generate a '1' every bit time interval). For the serial receive communication you need to implement an oversampling mechanism of 16.

Baud rate generation for oversampling of 16:

- For 25 MHz, clock period ~ 40 ns, input clock must be divided by ~ 163.

- For 50 MHz, clock period ~ 20 ns, input clock must be divided by ~ 326.
- For 100 MHz, clock period ~ 10 ns, input clock must be divided by ~ 651.

Define a new entity for the receive FSM. The next figure presents the ports of this entity.

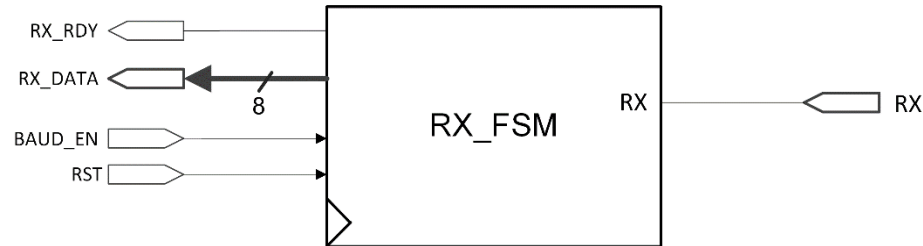


Figure 12-2: RX_FSM Entity Description

The detailed FSM implementation is presented in the figure below. A state transition is triggered only in the clock cycle when BAUD_ENable is '1'. This ensures that a bit will be valid for the baud rate period.

For the RX_FSM you have to use two auxiliary counters: BAUD_CNT and BIT_CNT.

The BIT_CNT is similar to the one in the TX_FSM, i.e. a signal with the functionality of a counter inside the RX_FSM; it holds the current transmitting bit number. It should be incremented in the bit state and should be reset after each serial transfer (you can do that in the idle state, or in all states except the bit state).

The BAUD_CNT is a signal is a signal with the functionality of a counter inside the RX_FSM; it counts the number of BAUD_ENables in order to ensure a correct oversampling mechanism. Remember that you use an oversampling factor of 16.

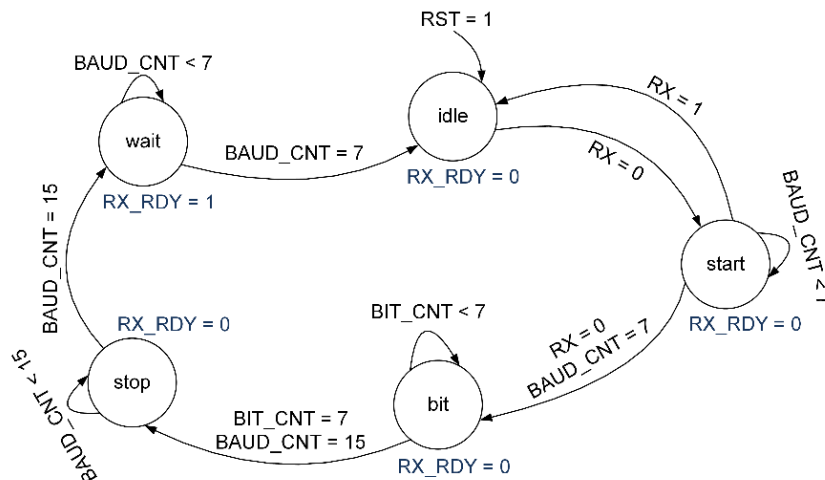


Figure 12-3: RX_FSM Implementation

Write the VHDL code and implement in the “test_env” project the RX_FSM state machine. Use a FSM with 2 or 3 processes (see appendix 7, laboratory 11). You also have to implement a shift register in order to receive the correct data from the serial input line. The RX signal will be shifted in this shift register only once per bit interval; i.e. in the middle of the transmitting interval. Test the communication between the FPGA board and the PC. The parameters of the serial communication are: 1 start bit, 8 data bits, 1 stop bit, no parity bit, 9600 baud rate. Make sure that these settings are also configured in the HTERM / hyper-terminal application. You have to identify the serial port where the module is connected – exactly like in the previous lab.

In order to test the serial transmission from the computer to the FPGA board, connect the RX_DATA output to the SSD (2 digits), RST to ‘0’ or a MPG enable signal. On the SSD, you will see the 8-bit ASCII code representation of the characters that you are sending from the PC.

12.3.2. I/O from the MIPS CPU – optional

Connect the RX_FSM into your own MIPS processor implementation. At this point, you are allowed to use your finished and complete processor (single-cycle or pipeline).

You have to receive 16-bits of valid data from the computer and feed this data into your MIPS processor. Depending on your program you can define what fields will be written with the data coming from the computer (register from the Register File, Data Memory location or even the Instructions from the Instruction Memory).

Remember that when receiving data from the serial RX line the 8-bits from a data transfer represent an ASCII character, hence you are required to make 4 transfers in order to receive the alphanumerical encoding of the 4 x 4-bit hexadecimal value (use a decoder/ROM to generate the 4-bit hexadecimal data and then concatenate 4 receive transfers in order to obtain the correct 16-bit data that will be fed to your processor).

Define the methodology to receive the 16-bit data over the serial RX line. Use the RX_RDY signal to control the writing of the data into your processor.

12.4. References

- [1] XST User Guide
- [2] Digilent Basys Board – Reference Manual
- [3] Digilent Basys 2 Board – Reference Manual
- [4] Digilent Basys 3 Board – Reference Manual
- [5] Digilent Pmod USB-UART – Reference Manual
- [6] <http://www.asciitable.com/>
- [7] <http://www.der-hammer.info/terminal/>

F. Appendix 6 – MIPS Instruction Reference

Note: ALL immediate values should be sign extended.

Exception: For logical operations immediate values should be zero extended.
After extensions, you treat them as signed or unsigned 32-bit numbers.

For the non-immediate instructions, the only difference between signed and unsigned instructions (ex ADD vs. ADDU) is that signed instructions can generate an overflow.

The instruction formats are given, you can figure out the binary instruction codes. The instruction descriptions are given below. Additional details can be found here: "[MIPS Single Cycle Processor](#)", John Alexander, Barret Schloerke, Daniel Sedam, Iowa State University

ADD – Add

Description:	Adds two registers and stores the result in a register
Operation:	$\$d \leftarrow \$s + \$t$; advance_pc (4);
Syntax:	add \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0000

ADDI – Add immediate

Description:	Adds a register and a signed immediate value and stores the result in a register
Operation:	$\$t \leftarrow \$s + \text{imm}$; advance_pc (4);
Syntax:	addi \$t, \$s, imm
Encoding:	0010 00ss ssst tttt iiiiii iiiiii iiiiii

ADDIU – Add immediate unsigned

Description:	Adds a register and an unsigned immediate value and stores the result in a register
Operation:	$\$t \leftarrow \$s + \text{imm}$; advance_pc (4);
Syntax:	addiu \$t, \$s, imm
Encoding:	0010 01ss ssst tttt iiiiii iiiiii iiiiii

ADDU – Add unsigned

Description:	Adds two registers and stores the result in a register
Operation:	$\$d \leftarrow \$s + \$t$; advance_pc (4);
Syntax:	addu \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0001

AND – Bitwise and

Description:	Bitwise ands two registers and stores the result in a register
Operation:	$\$d \leftarrow \$s \& \$t$; advance_pc (4);
Syntax:	and \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0100

ANDI – Bitwise and immediate

Description:	Bitwise ands a register and an immediate value and stores the result in a register
Operation:	$\$t \leftarrow \$s \& \text{imm}$; advance_pc (4);
Syntax:	andi \$t, \$s, imm
Encoding:	0011 00ss ssst tttt iiiiii iiiiii iiiiii

BEQ – Branch on equal

Description:	Branches if the two registers are equal
Operation:	if $\$s == \t advance_pc (offset << 2); else advance_pc (4);
Syntax:	beq \$s, \$t, offset
Encoding:	0001 00ss ssst tttt iiiiii iiiiii iiiiii

BGEZ – Branch on greater than or equal to zero

Description:	Branches if the register is greater than or equal to zero
Operation:	if $\$s \geq 0$ advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgez \$s, offset
Encoding:	0000 01ss sss0 0001 iiiiii iiiiii iiiiii

BGEZAL – Branch on greater than or equal to zero and link

Description:	Branches if the register is greater than or equal to zero and saves the return address in \$31
Operation:	if $\$s \geq 0$ $\$31 = PC + 8$ (or $nPC + 4$); advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgezal \$s, offset
Encoding:	0000 01ss sss1 0001 iiiiii iiiiii iiiiii

BGTZ – Branch on greater than zero

Description:	Branches if the register is greater than zero
Operation:	if $\$s > 0$ advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgtz \$s, offset
Encoding:	0001 11ss sss0 0000 iiiiii iiiiii iiiiii

BLEZ – Branch on less than or equal to zero

Description:	Branches if the register is less than or equal to zero
Operation:	if $\$s \leq 0$ advance_pc (offset << 2); else advance_pc (4);
Syntax:	blez \$s, offset
Encoding:	0001 10ss sss0 0000 iiiiii iiiiii iiiiii

BLTZ – Branch on less than zero

Description:	Branches if the register is less than zero
Operation:	if $\$s < 0$ advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltz \$s, offset
Encoding:	0000 01ss sss0 0000 iiiiii iiiiii iiiiii

BLTZAL – Branch on less than zero and link

Description:	Branches if the register is less than zero and saves the return address in \$31
Operation:	if $\$s < 0$ \$31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltzal \$s, offset
Encoding:	0000 01ss sss1 0000 iiiiii iiiiii iiiiii

BNE – Branch on not equal

Description:	Branches if the two registers are not equal
Operation:	if $\$s \neq \t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bne \$s, \$t, offset
Encoding:	0001 01ss ssst tttt iiiiii iiiiii iiiiii

DIV – Divide

Description:	Divides \$s by \$t and stores the quotient in \$LO and the remainder in \$HI
Operation:	$\$LO \leftarrow \$s / \$t$; $\$HI \leftarrow \$s \% \$t$; advance_pc (4);
Syntax:	div \$s, \$t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1010

DIVU – Divide unsigned

Description:	Divides \$s by \$t and stores the quotient in \$LO and the remainder in \$HI
Operation:	$\$LO \leftarrow \$s / \$t$; $\$HI \leftarrow \$s \% \$t$; advance_pc (4);
Syntax:	divu \$s, \$t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1011

J – Jump

Description:	Jumps to the calculated address
Operation:	$PC \leftarrow nPC$; $nPC = (PC \& 0xf0000000) (target \ll 2)$;
Syntax:	j target
Encoding:	0000 10ii iiiiii iiiiii iiiiii iiiiii

JAL – Jump and link

Description:	Jumps to the calculated address and stores the return address in \$31
Operation:	$\$31 \leftarrow PC + 8$ (or $nPC + 4$); $PC = nPC$; $nPC = (PC \& 0xf0000000) (target \ll 2)$;
Syntax:	jal target
Encoding:	0000 11ii iiiiii iiiiii iiiiii iiiiii

JR – Jump register

Description:	Jump to the address contained in register \$s
Operation:	$PC \leftarrow nPC$; $nPC = \$s$;
Syntax:	jr \$s
Encoding:	0000 00ss sss0 0000 0000 0000 0000 1000

LB – Load byte

Description:	A byte is loaded into a register from the specified address.
Operation:	$\$t \leftarrow MEM[\$s + \text{offset}]$; advance_pc (4);
Syntax:	lb \$t, offset(\$s)
Encoding:	1000 00ss ssst tttt iiiiii iiiiii iiiiii

LUI – Load upper immediate

Description:	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation:	$\$t \leftarrow (\text{imm} \ll 16)$; advance_pc (4);
Syntax:	lui \$t, imm
Encoding:	0011 11-- ---t tttt iiiiii iiiiii iiiiii

LW – Load word

Description:	A word is loaded into a register from the specified address.
Operation:	$\$t \leftarrow MEM[\$s + \text{offset}]$; advance_pc (4);
Syntax:	lw \$t, offset(\$s)
Encoding:	1000 11ss ssst tttt iiiiii iiiiii iiiiii

MFHI – Move from HI

Description:	The contents of register HI are moved to the specified register.
Operation:	$\$d \leftarrow \HI ; advance_pc (4);
Syntax:	mfhi \$d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0000

MFLO – Move from LO

Description:	The contents of register LO are moved to the specified register.
Operation:	$\$d \leftarrow \LO ; advance_pc (4);
Syntax:	mflo \$d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0010

MULT – Multiply

Description:	Multiplies \$s by \$t and stores the result in \$Hi and \$LO.
Operation:	$\$Hi, \$LO \leftarrow \$s * \t ; advance_pc (4);
Syntax:	mult \$s, \$t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1000

MULTU – Multiply unsigned

Description:	Multiplies \$s by \$t and stores the result in \$Hi and \$LO.
Operation:	$\$Hi, \$LO \leftarrow \$s * \t ; advance_pc (4);
Syntax:	multu \$s, \$t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1001

NOOP – no operation

Description:	Performs no operation.
Operation:	advance_pc (4);
Syntax:	noop
Encoding:	0000 0000 0000 0000 0000 0000 0000 0000

Note: The encoding for a NOOP represents the instruction SLL \$0, \$0, 0 which has no side effects. In fact, nearly every instruction that has \$0 as its destination register will have no side effect and can thus be considered a NoOP instruction.

OR – Bitwise or

Description:	Bitwise logical ors two registers and stores the result in a register
Operation:	$\$d \leftarrow \$s \$t$; advance_pc (4);
Syntax:	or \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0101

ORI – Bitwise or immediate

Description:	Bitwise ors a register and an immediate value and stores the result in a register
Operation:	$\$t \leftarrow \$s \text{imm}$; advance_pc (4);
Syntax:	ori \$t, \$s, imm
Encoding:	0011 01ss ssst tttt iiiiii iiiiii iiiiii

SB – Store byte

Description:	The least significant byte of \$t is stored at the specified address.
Operation:	$\text{MEM}[\$s + \text{offset}] \leftarrow (0xff \& \$t)$; advance_pc (4);
Syntax:	sb \$t, offset(\$s)
Encoding:	1010 00ss ssst tttt iiiiii iiiiii iiiiii

SLL – Shift left logical

Description:	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation:	$\$d \leftarrow \$t \ll h$; advance_pc (4);
Syntax:	sll \$d, \$t, h
Encoding:	0000 00ss ssst tttt dddd dhhh hh00 0000

SLLV – Shift left logical variable

Description:	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation:	$\$d \leftarrow \$t \ll \$s$; advance_pc (4);
Syntax:	sllv \$d, \$t, \$s
Encoding:	0000 00ss ssst tttt dddd d--- --00 0100

SLT – Set on less than (signed)

Description:	If \$s is less than \$t, \$d is set to one. It gets zero otherwise.
Operation:	if $\$s < \t $\$d \leftarrow 1$; advance_pc (4); else $\$d \leftarrow 0$; advance_pc (4);
Syntax:	slt \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1010

SLTI – Set on less than immediate (signed)

Description:	If \$s is less than immediate, \$t is set to one. It gets zero otherwise.
Operation:	if $\$s < \text{imm}$ $\$t \leftarrow 1$; advance_pc (4); else $\$t \leftarrow 0$; advance_pc (4);
Syntax:	slti \$t, \$s, imm
Encoding:	0010 10ss ssst tttt iiiiii iiiiii iiiiii

SLTIU – Set on less than immediate unsigned

Description:	If \$s is less than the unsigned immediate, \$t is set to one. It gets zero otherwise.
Operation:	if $\$s < \text{imm}$ $\$t \leftarrow 1$; advance_pc (4); else $\$t \leftarrow 0$; advance_pc (4);
Syntax:	sltiu \$t, \$s, imm
Encoding:	0010 11ss ssst tttt iiiiii iiiiii iiiiii

SLTU – Set on less than unsigned

Description:	If \$s is less than \$t, \$d is set to one. It gets zero otherwise.
Operation:	if $\$s < \t $\$d \leftarrow 1$; advance_pc (4); else $\$d \leftarrow 0$; advance_pc (4);
Syntax:	sltu \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1011

SRA – Shift right arithmetic

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation:	$\$d \leftarrow \$t \gg h$; advance_pc (4);
Syntax:	sra \$d, \$t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0011

SRL – Shift right logical

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
Operation:	$\$d \leftarrow \$t \gg h$; advance_pc (4);
Syntax:	srl \$d, \$t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0010

SRLV – Shift right logical variable

Description:	Shifts a register value right by the amount specified in \$s and places the value in the destination register. Zeroes are shifted in.
Operation:	$\$d \leftarrow \$t \gg \$s$; advance_pc (4);
Syntax:	srlv \$d, \$t, \$s
Encoding:	0000 00ss ssst tttt dddd d000 0000 0110

SUB – Subtract

Description:	Subtracts two registers and stores the result in a register
Operation:	$\$d \leftarrow \$s - \$t$; advance_pc (4);
Syntax:	sub \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0010

SUBU – Subtract unsigned

Description:	Subtracts two registers and stores the result in a register
Operation:	$\$d \leftarrow \$s - \$t$; advance_pc (4);
Syntax:	subu \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0011

SW – Store word

Description:	The contents of \$t is stored at the specified address.
Operation:	$\text{MEM}[\$s + \text{offset}] \leftarrow \t ; advance_pc (4);
Syntax:	sw \$t, offset(\$s)
Encoding:	1010 11ss ssst tttt iiiiii iiiiii iiiiii

SYSCALL – System call

Description:	Generates a software interrupt.
Operation:	advance_pc (4);
Syntax:	syscall
Encoding:	0000 00-- ---- ---- ---- ---- --00 1100

XOR – Bitwise exclusive or

Description:	Exclusive ors two registers and stores the result in a register
Operation:	$\$d \leftarrow \$s \wedge \$t$; advance_pc (4);
Syntax:	xor \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d--- --10 0110

XORI – Bitwise exclusive or immediate

Description:	Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation:	$\$t \leftarrow \$s \wedge \text{imm}$; advance_pc (4);
Syntax:	xori \$t, \$s, imm
Encoding:	0011 10ss ssst tttt iiii iiii iiii iiii

G. Appendix 7 – Finite State Machine Implementations

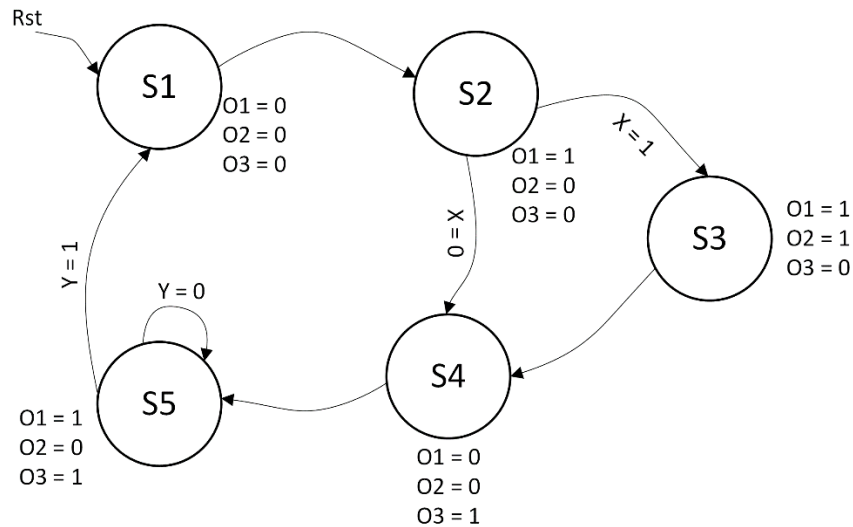


Figure G-1: Finite State Machine Example (XST User Guide)

IO Pins	Description
clk	Positive Edge Clock
Rst	Asynchronous Reset (Active High)
X, Y	FSM Inputs
O1, O2, O3	FSM Outputs

Table G.1: FSM Pin Descriptions

The Xilinx Synthesis technology recognizes Finite State Machines written in VHDL with 1, 2 or 3 processes. A coding example for the Finite State Machine presented in Figure F.1, for each kind of implementation, is given on the next pages. You have to adapt the Finite State Machine implementation to your own FSM description.

VHDL Coding Example: FSM with One Process

```

entity fsm_1 is
  port (
    clk, rst, x, y : IN std_logic;
    o1, o2, o3      : OUT std_logic
  );
end entity;

architecture beh1 of fsm_1 is
  type state_type is (s1, s2, s3, s4, s5);
  signal state : state_type;
begin

  process (clk, rst, x, y)
  begin
    if (rst='1') then
      state <= s1;
      o1 <= '0'; o2 <= '0'; o3 <= '0';
    elsif (clk='1' and clk'event) then
      case state is
        when s1 => state <= s2;
                     o1 <= '1'; o2 <= '0'; o3 <= '0';
        when s2 => if x = '1' then
                     state <= s3;
                     o1 <= '1'; o2 <= '1'; o3 <= '0';
                   else
                     state <= s4;
                     o1 <= '0'; o2 <= '0'; o3 <= '1';
                   end if;
        when s3 => state <= s4;
                     o1 <= '0'; o2 <= '0'; o3 <= '1';
        when s4 => state <= s5;
                     o1 <= '1'; o2 <= '0'; o3 <= '1';
        when s5 => if y = '1' then
                     state <= s1;
                     o1 <= '0'; o2 <= '0'; o3 <= '0';
                   else
                     state <= s5;
                     o1 <= '1'; o2 <= '0'; o3 <= '1';
                   end if;
      end case;
    end if;
  end process;

end beh1;

```


VHDL Coding Example: FSM with Two Processes

```

entity fsm_2 is
  port (
    clk, rst, x, y : IN std_logic;
    o1, o2, o3      : OUT std_logic
  );
end entity;

architecture beh1 of fsm_2 is
  type state_type is (s1, s2, s3, s4, s5);
  signal state : state_type;
begin

  process1: process (clk, rst, x, y)
  begin
    if (rst='1') then
      state <= s1;
    elsif (clk='1' and clk'event) then
      case state is
        when s1 => state <= s2;
        when s2 => if x = '1' then
                      state <= s3;
                    else
                      state <= s4;
                    end if;
        when s3 => state <= s4;
        when s4 => state <= s5;
        when s5 => if y = '1' then
                      state <= s1;
                    else
                      state <= s5;
                    end if;
      end case;
    end if;
  end process process1;

  process2: process (state)
  begin
    case state is
      when s1 => o1<='0'; o2<='0'; o3<='0';
      when s2 => o1<='1'; o2<='0'; o3<='0';
      when s3 => o1<='1'; o2<='1'; o3<='0';
      when s4 => o1<='1'; o2<='0'; o3<='0';
      when s5 => o1<='1'; o2<='0'; o3<='1';
    end case;
  end process process2;
end architecture beh1;

```

```
        end case;  
    end process process2;  
end beh1;
```

VHDL Coding Example: FSM with Three Processes

```
entity fsm_3 is
  port (
    clk, rst, x, y : IN std_logic;
    o1, o2, o3     : OUT std_logic
  );
end entity;

architecture beh1 of fsm_3 is
  type state_type is (s1, s2, s3, s4, s5);
  signal state, next_state : state_type;
begin
  process1: process (clk, rst)
  begin
    if (reset = '1') then
      state <= s1;
    elsif (clk = '1' and clk'event) then
      state <= next_state;
    end if;
  end process process1;

  process2: process (state, x, y)
  begin
    case state is
      when s1 => next_state <= s2;
      when s2 => if x = '1' then
                    next_state <= s3;
                  else
                    next_state <= s4;
                  end if;
      when s3 => next_state <= s4;
      when s4 => next_state <= s5;
      when s5 => if y = '1' then
                    next_state <= s1;
                  else
                    next_state <= s5;
                  end if;
    end case;
  end process process2;

  process3: process (state)
  begin
    case state is
```

```
        when s1 => o1<='0'; o2<='0'; o3<='0';
        when s2 => o1<='1'; o2<='0'; o3<='0';
        when s3 => o1<='1'; o2<='1'; o3<='0';
        when s4 => o1<='1'; o2<='0'; o3<='0';
        when s5 => o1<='1'; o2<='0'; o3<='1';
    end case;
end process process3;
end beh1;
```

H. Appendix 8 – ASCII Codes Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.comFigure H-1: ASCII Codes (<http://www.asciitable.com/>)