Students' Guide to

Practicals on Computer Architecture

Isuru Nawinne

Faculty of Engineering - University of Peradeniya

Students' Guide to Practicals on Computer Architecture

By Isuru Nawinne

ISBN 978-624-92913-0-0

Downloadable ebook and supplementary material available at

https://cepdnaclk.github.io/Computer-Architecture-Web

Publisher:

Dr. Isuru Nawinne,
Department of Computer Engineering, Faculty of Engineering,
University of Peradeniya,
Peradeniya 20400,
Sri Lanka.
isurunawinne@eng.pdn.ac.lk https://people.ce.pdn.ac.lk/staff/academic/isuru-nawinne/

Content

Preface

Computer architecture is best understood not only through theory, but through creation. This guidebook is designed to help students bridge the gap between abstract architectural concepts and tangible implementation. By building a simple CPU and memory system step by step, learners will gain an engineer's insight into how complex computers actually work—how instructions are executed, how data flows through hardware, and how the smallest design decisions influence performance and functionality.

This practical series is structured to complement traditional lectures in computer architecture. Each exercise is carefully designed to reinforce core concepts such as instruction execution, datapath design, control logic, memory hierarchy and caching. Rather than working with simulations alone, students are encouraged to think like designers—to experiment, to debug, and to iteratively refine their systems until theory comes alive in working hardware or detailed digital models.

The aim is not only to teach the mechanics of building a CPU, but also to *cultivate an appreciation for the elegance and complexity of computer systems*. By the end of this series, students should have both a deeper conceptual understanding and the practical confidence to design and analyze computing architectures on their own.

Acknowledgment

I would like to express my sincere appreciation to **Kisaru Liyanage**, for his invaluable support in testing and validating the exercises presented in this guidebook. His careful attention to detail, thoughtful feedback, and commitment to ensuring the clarity and accuracy of each practical task have contributed greatly to the quality and reliability of this work.

-

Isuru Nawinne

Senior Lecturer in Computer Engineering

Learning Methods

The best way to learn from this practical series is to treat each experiment as an *exploration*. You will learn most effectively when you combine three complementary approaches:

1. *Constructive Learning* — Build and test each component yourself. Mistakes are part of the design process. Debugging is where the deepest understanding emerges.
2. *Reflective Learning* — After completing each exercise, take time to analyze what you built. Ask why certain design choices work, and how they could be improved.
3. *Collaborative Learning* — Discuss your designs with peers. Explaining your reasoning and comparing solutions will expose new perspectives and strengthen your understanding.

Whenever possible, visualize signals, trace instruction flows, and relate what you see to the theoretical models learned in lectures. Remember that the purpose of these practicals is not just to complete a circuit or simulation—it is to *understand the computer as a living system of ideas, logic, and engineering.*

Tools Needed

All design and simulation work in this series will be carried out using **Verilog HDL**, a hardware description language widely used in digital design and industry. The following open-source tools are recommended:

- **Icarus Verilog** — for compiling and simulating Verilog designs.
  (Website: https://steveicarus.github.io/iverilog/)
- **GTKWave** — for viewing signal waveforms generated during simulation.
  (Website: http://gtkwave.sourceforge.net/)

These tools are lightweight, cross-platform, and ideal for educational use. Together, they allow you to write, compile, simulate, and debug your CPU and memory system designs with professional-level visibility into internal signals and timing behavior. Alternatively, commercial tools such as *ModelSim* may be used if access is available.

Introduction

In this practical series, you will be designing a simple 8-bit single-cycle processor which includes an ALU, a register file and control logic, using Verilog HDL. You will also design a simple memory hierarchy, including a cache memory for your processor. Follow the guidelines given here to build your processor.

The microarchitecture of a processor is designed based on an ISA. For this practical series, we will use a simple ISA described below. Initially, your processor should implement the instructions `add, sub, and, or, mov, loadi, j` and `beq`. All instructions are of 32-bit fixed length and should be encoded in the format shown below.

`OP-CODE (bits 31-24)`	`RD / IMM (bits 23-16)`	`RT (bits 15-8)`	`RS / IMM (bits7-0)`

`Bits(31-24):` `OP-CODE` field identifies the instruction's operation. This should be used by the control logic to interpret the remaining fields and derive the control signals.
`Bits(23-16):` A register (`RD`) to be written to in the register file, or an immediate value (jump or branch target offset).
`Bits(15-8):` A register (`RT`) to be read from in the register file.
`Bits(7-0):` A register (`RS`) to be read from in the register file, or an immediate value.

Here are some examples about the usage and descriptions of these instructions:

Instruction	Description
`add 4 1 2`	Add value in register 2 to value in register 1, and place the result in register 4
`sub 4 1 2`	Subtract value in register 2 from the value in register 1, and place the result in register 4
`and 4 1 2`	Perform bit-wise AND on values in registers 1 and 2, and place the result in register 4
`or  4 1 2`	Perform bit-wise OR on values in registers 1 and 2, and place the result in register 4
`j   0x02`	Pump 2 instructions forward from the next instruction to be executed, by manipulating the Program Counter. Ignore `bits 15-0`
`beq 0xFE 1 2`	If values in registers 1 and 2 are equal, branch 2 instructions backward by manipulating the Program Counter
`mov 4 1`	Copy the value in register 1 to register 4. Ignore `bits 15-8`
`loadi 4 0xFF`	Load the immediate value `0xFF` to register 4. Ignore `bits 15-8`

You will be building your processor in four steps:

- In **Practical 1**, you will build an 8-bit ALU which implements all the functional units required to support the instructions `add, sub, and, or, mov,` and `loadi`.
- In **Practical 2**, you will implement a simple 8×8 register file.
- In **Practical 3**, you will implement the control logic and integrate all the components from Practicals 1 and 2 together to work as a complete processor.
- In **Practical 4**, you will expand your processor to support `j` and `beq` instructions.

Once you have completed Practical 4, we will move on to adding a memory subsystem to your processor, including caches. For this, we will expand the ISA to include the following memory access instructions. In the new instructions, memory accessing can be done using two different addressing modes: register direct addressing or immediate addressing. See examples below:

Instruction	Description
`lwd 4 2`	Read memory at the address given in register 2 (`RS`) and store the result in register 4 (`RD`). Ignore `bits 15-8`
`lwi 4 0x1F`	Read memory at address `0x1F` (`IMM`) and store the result in register 4 (`RD`). Ignore `bits 15-8`
`swd 2 3`	Write value from register 2 (`RT`) to the memory at the address given in register 3 (`RS`). Ignore `bits 23-16`
`swi 2 0x8C`	Write value from register 2 (`RT`) to the memory at address `0x8C` (`IMM`). Ignore `bits 23-16`

You will be implementing your memory subsystem in three steps:

- In **Practical 5**, you will add hardware support for the instructions `lwd, lwi, swd,` and `swi`.
- In **Practical 6**, you will implement a direct-mapped data cache for your processor.
- In **Practical** 7, you will add instruction memory and instruction cache to complete the system.

Supplementary materials needed for the practicals are provided in a .zip file available at https://cepdnaclk.github.io/Computer-Architecture-Web.

Practical 1 - Arithmetic and Logic Unit (ALU)

At the heart of every computer processor is an Arithmetic Logic Unit (ALU). This is the part of the computer which performs arithmetic and logic operations on numbers, e.g. addition, subtraction, etc. Use Verilog HDL to implement an 8-bit ALU which can perform four different functions to support the instructions `add`, `sub`, `and`, `or`, `mov`, and `loadi` (note: we will <u>not</u> support `j` and `beq` at this stage). Figure 1, below, shows the interfaces of the ALU you will be implementing.

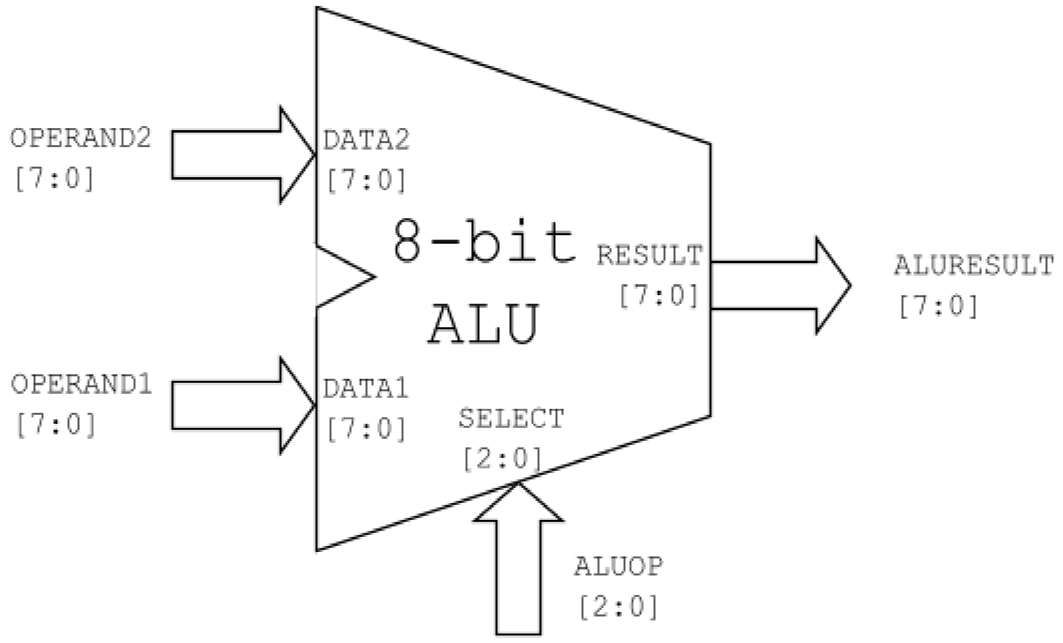


Figure 1: Interfaces of the ALU

The ALU that you are building should work with 8-bit operands. There should be two 8-bit input ports for operands (`DATA1` and `DATA2`), one 8-bit output port (`RESULT`) and one 3-bit control input port (`SELECT`), which should be used to pick the required function inside the ALU out of the available four functions, based on the instruction's `OP-CODE`. (You may notice that two bits are enough for the `SELECT` interface, as there are only four function choices. We're reserving the 3rd bit for future use.)

The 3-bit `ALUOP` control signal supplied to the `SELECT` port should be derived from `OP-CODE` using combinational logic by the control unit. You may define suitable `OP-CODE` values for the given instruction set and implement an appropriate mapping from `OP-CODE` to the `ALUOP` signal when you design the control logic.

The following module definition gives a template interface for your ALU:
`module alu(DATA1, DATA2, RESULT, SELECT)`

Make sure you use the same signal and register names as the ones given here. The Table below shows the four functions (operations) that your 8-bit ALU should be able to perform.

ALU Functions				
`SELECT`	Function	Description	Supported Instructions	Unit's Delay
`000`	FORWARD	(forward `DATA2` into `RESULT`) `DATA2→ RESULT`	`loadi, mov`	`#1`
`001`	ADD	(add `DATA1` and `DATA2`) `DATA1 + DATA2 → RESULT`	`add, sub`	`#2`
`010`	AND	(bitwise AND on `DATA1` with `DATA2`) `DATA1 &DATA2 → RESULT`	`and`	`#1`
`011`	OR	(bitwise OR on `DATA1` with `DATA2`) `DATA1 | DATA2 → RESULT`	`or`	`#1`
`1XX`	Reserved	Reserved for future functional units	-	-

These functional units must be implemented as separate modules and instantiated inside the *alu* module. FORWARD unit should simply send an operand value from `DATA2` to its output. This unit will be used by the `loadi` and `mov` instructions to place the respective source operand in the specified destination register. ADD, AND and OR functional units will use the values in `DATA1` and `DATA2`, perform the corresponding operation, and send the result to its output. The *alu* module should use a MUX to pick one of the functional units' outputs and send it to `RESULT` based on the `SELECT` value. To simulate the ALU latencies realistically, include the given artificial delays in each functional unit (note that the delays are for individual functional units, not the MUX that will choose the `RESULT`).

1. Design and implement the *alu* module using Verilog. Include a lot of comments. Make sure you properly deal with any unused bit combinations of the `SELECT` port. (Hint: using a case structure will make this job easy)
2. Write a testbench and simulate your alu module. Test with different combinations of `OPERAND1`, `OPERAND2` and `ALUOP` signal values.

Practical 2 - Register File

Next, you should implement a simple 8×8 register file. The purpose of the register file is to store output values generated by the ALU, and to supply the ALU's inputs with operands.

Your register file should be able to store eight 8-bit values (`register0 - register7`). It should contain one 8-bit data input port (`IN`) and two 8-bit data output ports (`OUT1` and `OUT2`). To specify which register you are reading or writing with a given port, you must include three address ports (`INADDRESS`, `OUT1ADDRESS`, `OUT2ADDRESS`).

You must also include a control input port `WRITE` to accommodate the `WRITEENABLE` control signal. Since the register file is a sequential unit, you will need `CLOCK` and `RESET` signals for synchronization.

A block diagram of the register file is shown in Figure 2 below.

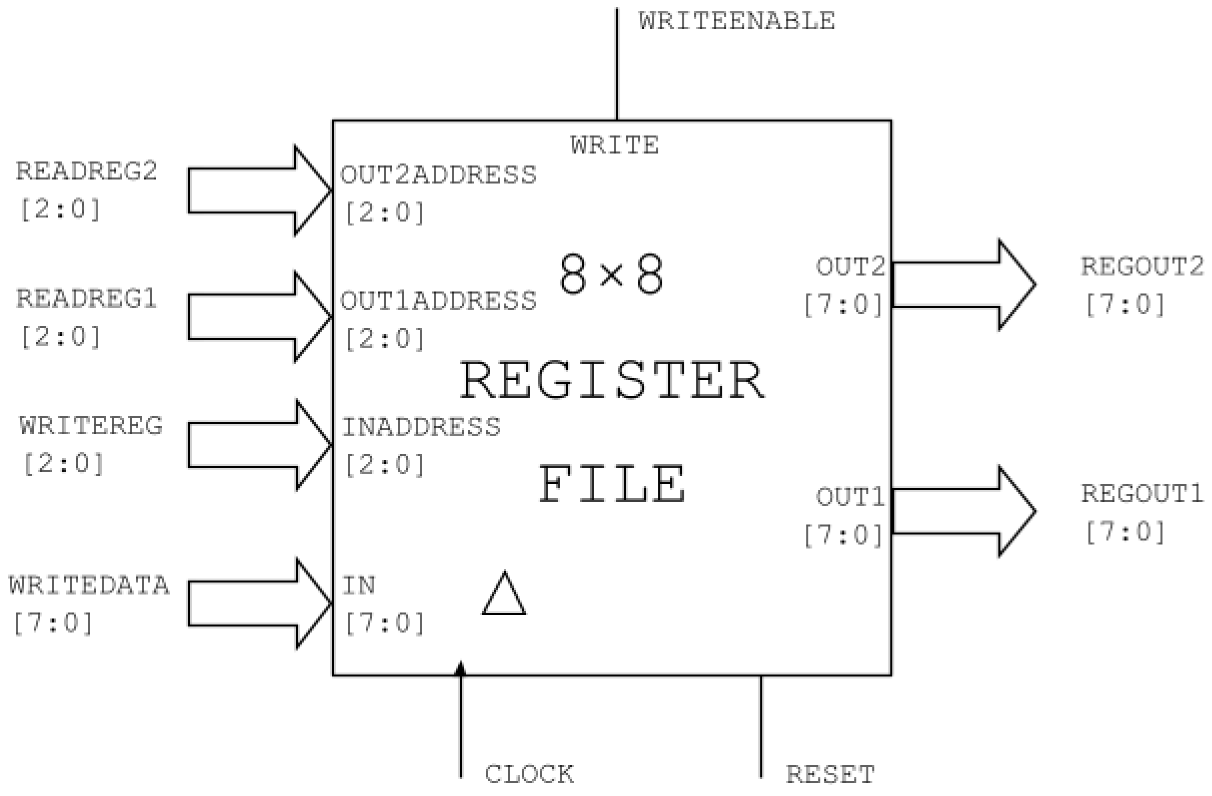


Figure 2: Interfaces of the Register File

The following module definition gives a template interface for your register file:

module reg_file(IN,OUT1,OUT2,INADDRESS,OUT1ADDRESS,OUT2ADDRESS, WRITE, CLK, RESET)

The port `IN` represents the data input, with `INADDRESS` providing the register number to store that data in. The ports `OUT1` and `OUT2` are parallel data outputs, where `OUT1ADDRESS` and `OUT2ADDRESS`, respectively, provide the register numbers where data should be retrieved from.

Registers identified by `OUT1ADDRESS` and `OUT2ADDRESS` should be read asynchronously and the values should be loaded onto `OUT1` and `OUT2` respectively.

Writing to the register file must be done synchronously. When `WRITEENABLE` signal at the `WRITE` port is set high, rising edge of `CLOCK` should make the data present on the `IN` port to be written to the input register specified by the `INADDRESS`.

Register file reset should happen synchronously at the positive edge of the clock if the `RESET` signal is high. All registers should be cleared (written zero) in a reset event.

To simulate the register file read and write latencies realistically, include artificial delays of two time units (#2) for register reading and one time unit (#1) for writing operations including reset.

You need to **pay careful attention to the timings**. Test your design thoroughly using several inputs until you make sure the desired behaviors is achieved. Use GTKWave (or any other similar tool that you prefer) to help you visualize the timings.

1. Implement the behavioral model for the Register File. Represent your registers as an array of words and use a structured procedure to update register contents and register file outputs. Include **a lot of comments**.
2. Implement a testbench for your Register File, and thoroughly test your design.
3. Obtain timing diagrams clearly showing the reading and writing of registers.

Practical 3 - Integration & Control

Now you should compose a working CPU using your ALU and Register File, supporting the instructions `add`, `sub`, `and`, `or`, `mov`, and `loadi`. To do this, you will need to implement the control logic in a top-level module (you may call this module as *cpu*). Your CPU needs an instruction fetching mechanism and a **Program Counter** (PC) register, which points to the next instruction. You may choose to have a *control_unit* module and instantiate it within your top-level *cpu* module, or you may choose to implement all control logic in your *cpu* top-level module itself.

The following module definition gives a template interface for your CPU:

```
module cpu(PC, INSTRUCTION, CLK, RESET)
```

Since we do not have an instruction memory module yet to hold instructions, you should keep the instructions as an array of hardcoded instruction words (array size = 1024 bytes, i.e. 256 instructions) in the testbench file which you use to test your *cpu*. Your *cpu's* instruction fetching mechanism should read the hardcoded instructions asynchronously from the testbench, based on the address provided by PC.

You need combinational control logic to **decode** a fetched instruction, extract the `OP-CODE`, source/destination registers and immediate values. Based on the `OP-CODE` (bits 31-26), all control signals should be generated and sent to the Register File, ALU and other components appropriately. Bits 25-0 need to be directly sent to appropriate places where they may be used, without needing to wait.

For arithmetic instructions (`add` and `sub`), assume that the operands are **signed integers** with negative values presented in Two's Complement format. You will need to perform the Two's Complement operation on the second operand before supplying it to the ALU, in order to support both `add` and `sub` instructions using the same adder functional unit. You will need to use MUXs to achieve the desired control.

Pay careful attention to how you coordinate the timings of instruction fetching, decoding, register reading, execution, and register writing. To realistically simulate the latencies of instruction fetching, decoding and execution, you should include the following artificial timing delays in your CPU:

- PC Update (write to PC register) = One time unit (#1)
- Instruction Memory Read = Two time units (#2)
- Instruction Decode (generating control signals) = One time unit (#1)
- Two's complement operation = One time unit (#1)

In order to automatically increment the PC value by 4, you will need to include a dedicated adder. We will assume that this adder has a latency of one time unit (#1), which will work in parallel to instruction memory reading.

Use GTKWave (or any other similar tool that you prefer) to help you visualize the timings.

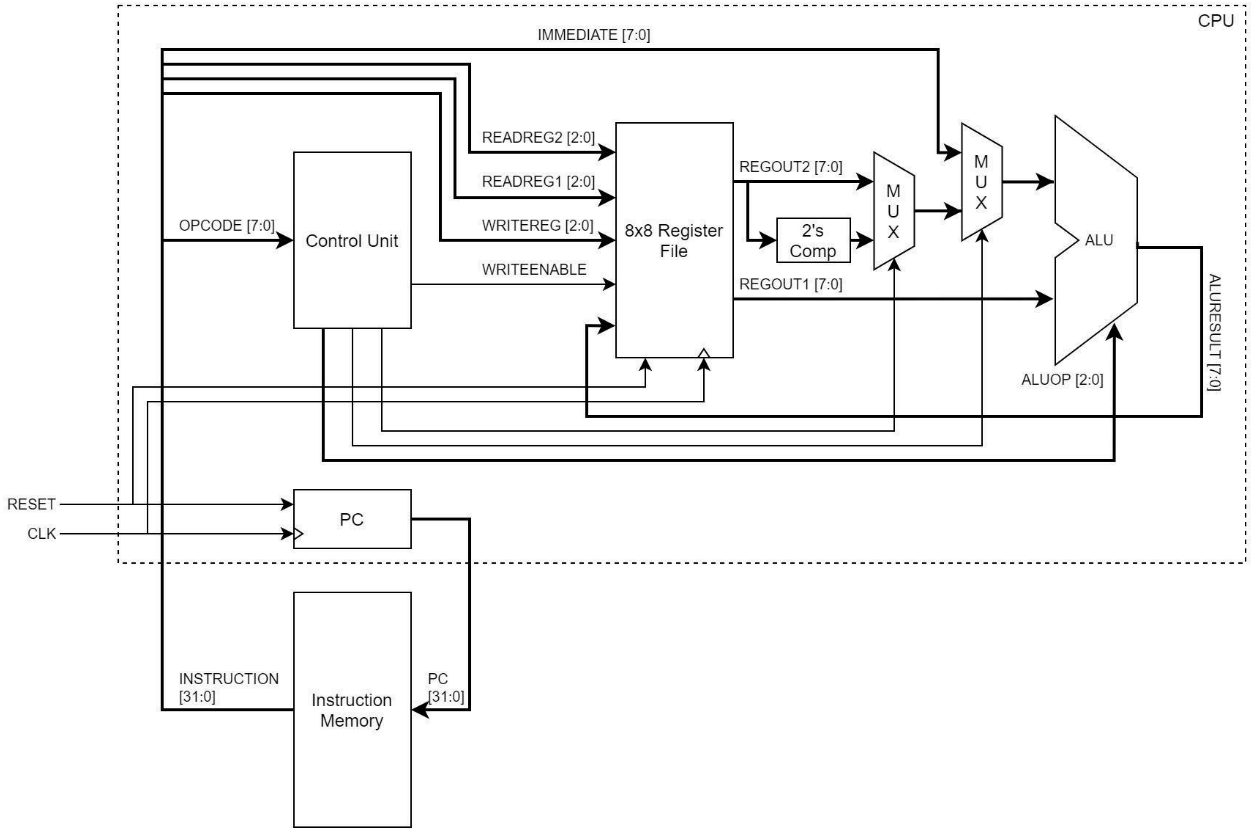


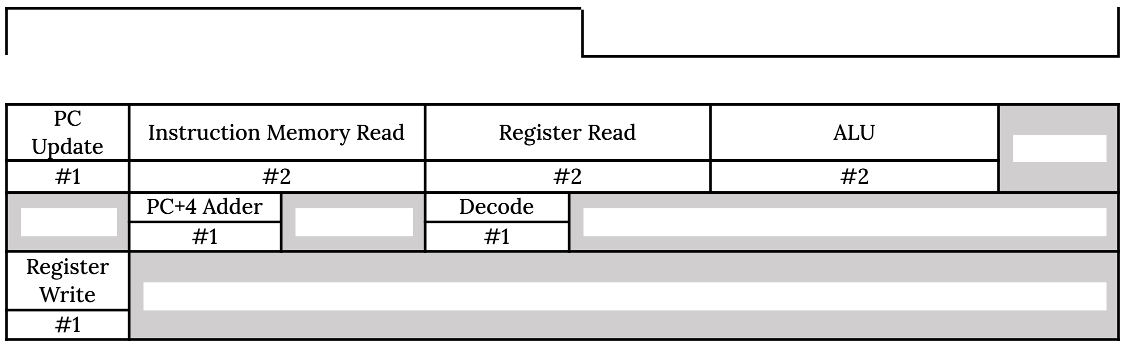
Figure 3: Overview of CPU

An overall block diagram for this simple CPU is provided in Figure 3. All components except the instruction memory should be housed within your top-level *cpu* module.

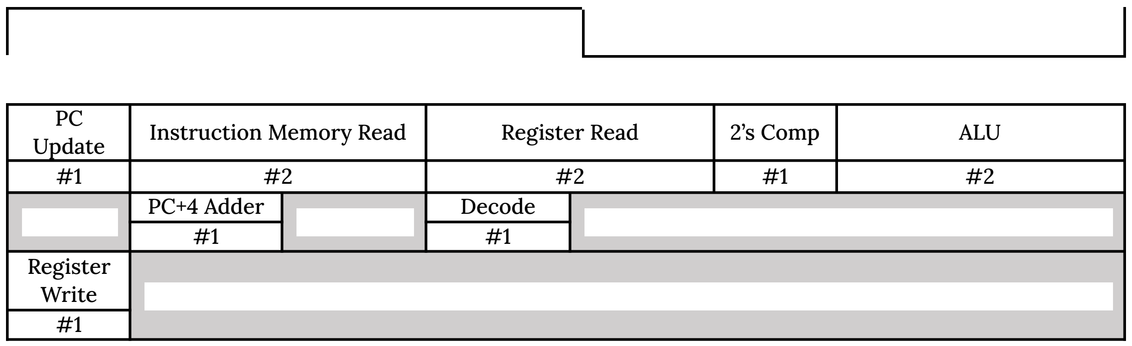
You must thoroughly test your *cpu* using several different software programs (instruction sequences). For this, you need to prepare programs as machine code and hardcode them inside your testbench file, one program at a time. Since it is easier to write textual assembly programs (rather than writing machine code), you may use the provided *Assembler* tool to convert textual assembly into machine code. Note that you must add your `OP-CODE` value definitions to the *assembler.c* file before using it to generate the assembler tool. A shell script is provided to you, which can convert assembled machine code into a memory image which you can easily copy onto your testbench.

The diagram given in Figure 4 shows the worst-case timing of your single-cycle CPU for the `add`, `sub`, `and`, `or`, `mov`, and `loadi` instructions. One clock cycle spans for eight (8) time units duration, rising edge to rising edge. Every instruction should complete within one clock cycle, and any data to be written to the register file should be ready by the rising edge at the end of the clock cycle. Writing to registers and PC should be synchronized to the rising edges of the clock, while reading of registers should happen asynchronously. The given timing delays should be artificially added to the corresponding operations in order to realistically simulate the latencies observable in a synthesized datapath. For the sake of simplicity, we will be assuming that our multiplexers and wires have negligible delays.

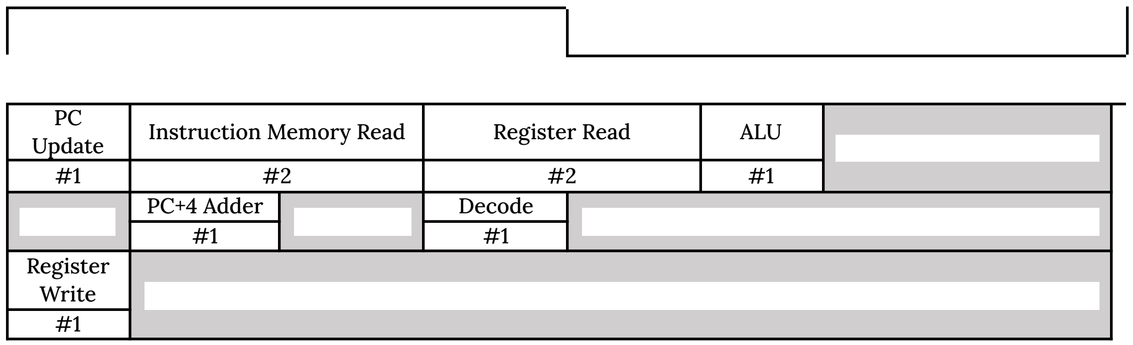
`add:`



`sub:`



`and/or/mov:`



`loadi:`

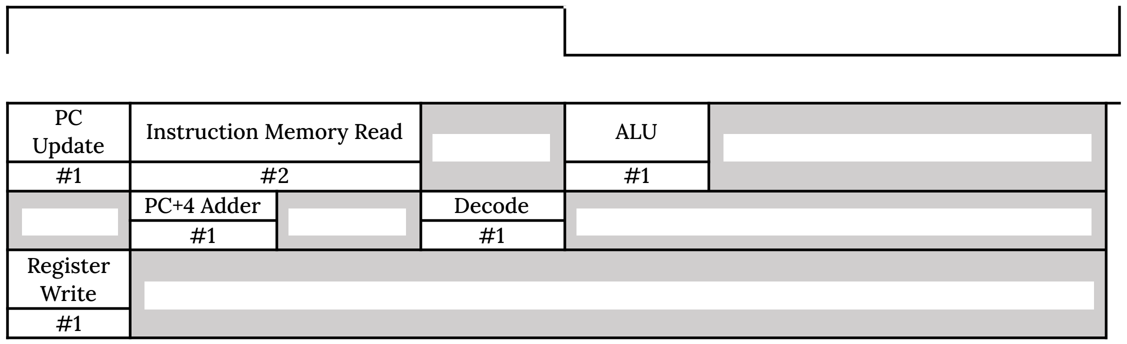


Figure 4: Timing Details for the Datapath

You must also implement the reset functionality of the CPU, so that the program can be restarted by setting the `RESET` signal high for a short period of time. If the `RESET` signal is high when writing to PC at a positive clock edge, write zero to PC instead of the next PC value in order to restart the program. So the CPU will read the instruction memory at address zero. In addition to this, your register file content should also be reset at the same time.

Note that data memory reading and writing is not implemented yet. You will be adding those functionalities in Practical 5.

1. Build the top-level *cpu* module to integrate the ALU and Register File using appropriate control logic. Include **a lot of comments**.
2. Write a testbench and thoroughly test your completed design. Hardcode your software program (instruction sequence) within the testbench file.
3. Obtain timing diagrams clearly showing the synchronized operation of the datapath and control signals for the given six instructions.

Practical 4 - Flow Control Instructions

Now that you have a working CPU which supports `add`, `sub`, `and`, `or`, `mov`, and `loadi` instructions, it's time to add microarchitectural support for the flow control instructions `j` and `beq`. For this, you will need to modify your top-level *cpu* and *alu* modules.

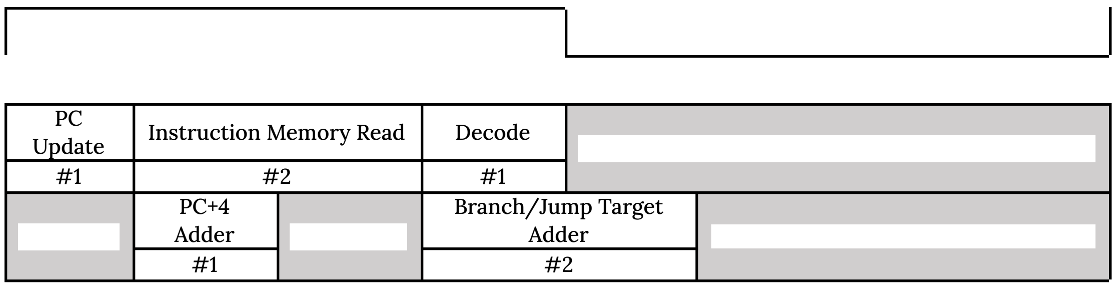
The *alu* needs an additional output port (`ZERO`) to indicate whether the result of a subtract operation is zero or not, in order to implement the `beq` instruction.

The *cpu* needs a new adder which can be used to compute the branch/jump target address based on the next PC value and the offset provided by the branch/jump instruction. We will assume that the new adder has a latency of two time units (#2), which will work in parallel to the ALU.

The control functionality within the top-level *cpu* module needs to be modified to: 1) manipulate the Program Counter using the immediate jump/branch target offsets provided in `j` and `beq` instructions (you will need to use adders and multiplexers); and 2) generate the additional control signals required.

The timing diagrams for `j` and `beq` instructions are shown in Figure 5 below.

`j:`



`beq:`

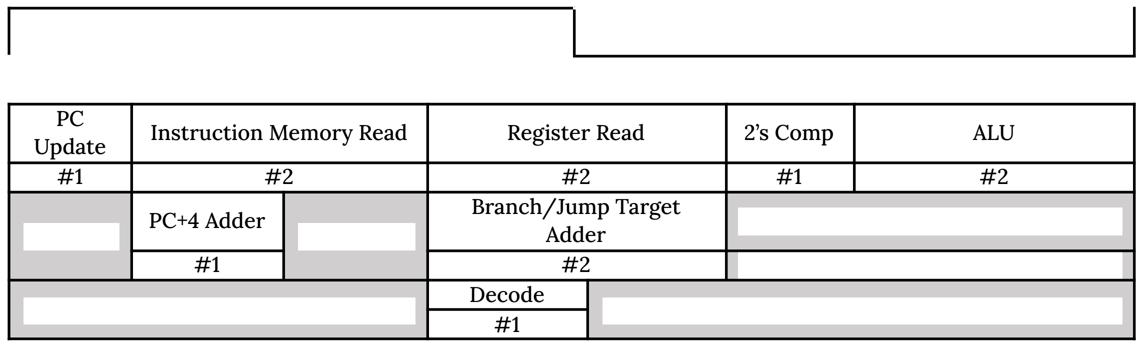


Figure 5: Timing Details for the Datapath

Before you go and modify your code, first **draw a complete block diagram** of the datapath and control by extending Figure 3 with the added components. Make sure to keep copies of your original files before modifying.

1. Modify the top-level *cpu* module and *alu* module to support the `j` and `beq` instructions. Include **a lot of comments.**
2. Write a testbench and thoroughly test your upgraded design. Hardcode your software program (instruction sequence) within the testbench file.
3. Obtain timing diagrams clearly showing the synchronized operation of the datapath and control signals for the two new instructions.

Practical 5 - Accessing Data Memory

Now you will be adding a memory sub-system for your single-cycle CPU. Some systems store both instructions and data in the same memory device, while other systems use separate memory devices for instructions and data. For our system, we will use separate memory devices.

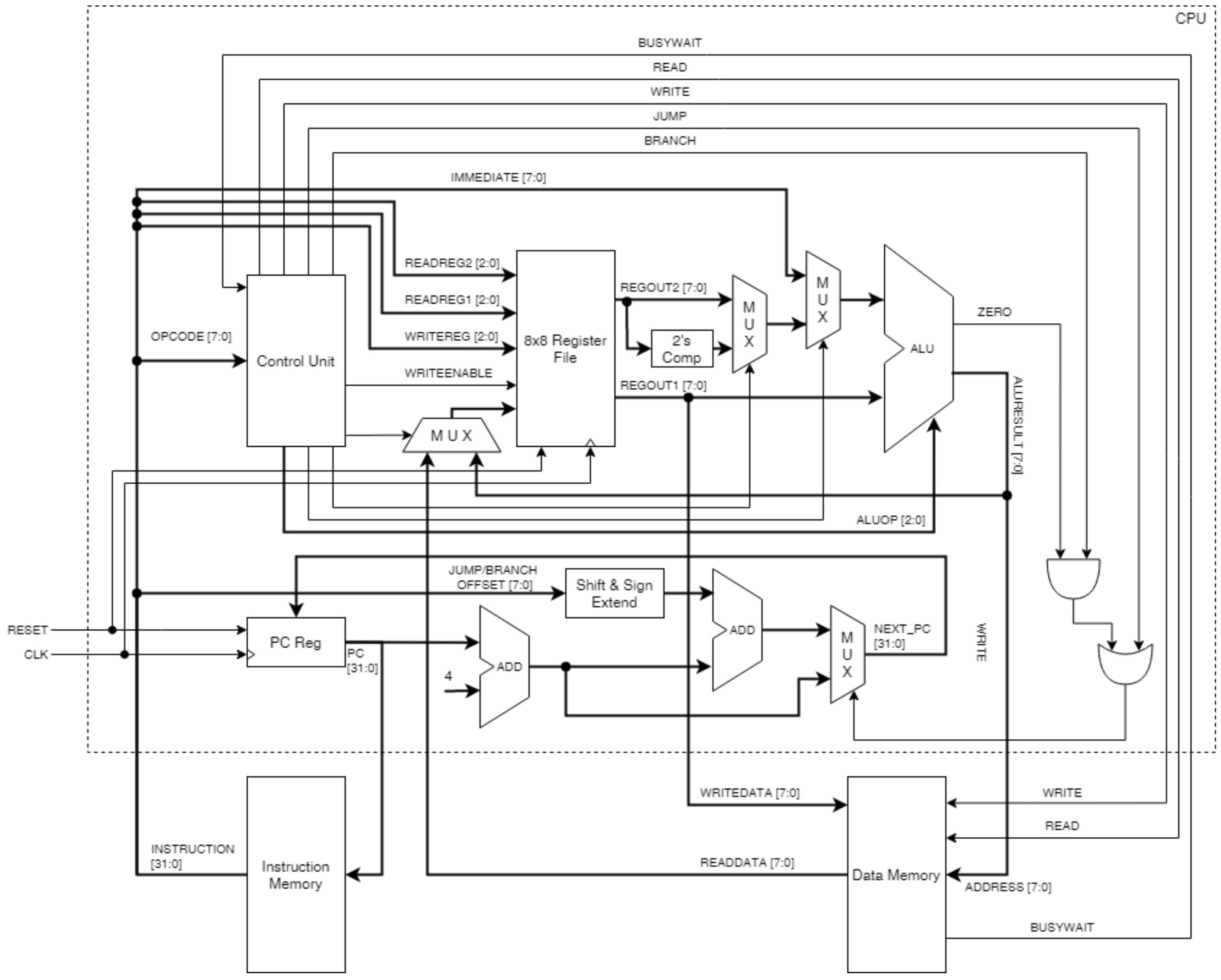


Figure 6: CPU with Data Memory

The diagram in Figure 6 shows a 256 Byte Data Memory module being connected to the CPU that you built in the previous lab. A sample memory module is given to you, which uses 256 8-bit registers to store data. The memory module uses the following signals to interface with the processor:

`ADDRESS`: Location in memory being accessed (read/write). ALU provides the value for this signal.

`WRITEDATA`: Data value to be stored in the memory, at the location pointed to by `ADDRESS`. This value is supplied by the Register File, then used by the data memory on a positive clock edge.

`READDATA`: Data value read from the memory, at the location pointed to by `ADDRESS`. This value is sent by the memory on a positive clock edge to the Register File for storing.

`READ`: Control signal to request the memory to perform a read operation on the provided `ADDRESS`. This signal is supplied by the CPU control unit, and cleared when memory de-asserts the `BUSYWAIT` signal.

`WRITE`: Control signal to request the memory to perform a write operation on the provided `ADDRESS`. This signal is supplied by the CPU control unit, and cleared when memory de-asserts the `BUSYWAIT` signal.

`BUSYWAIT`: Memory asserts this signal when CPU sets `READ/WRITE` control signals, and keeps it asserted while the operation is in progress. CPU control unit should stall the processor and hold the `ADDRESS` and `READ/WRITE` control signals stable while this wait signal is asserted. Memory de-asserts `BUSYWAIT` when a reading or writing operation is concluded. The next instruction should not be fetched by CPU until `BUSYWAIT` is de-asserted by the memory.

Study the given memory module and see how to connect it to your CPU. Note that a latency of 5 CPU clock cycles (#40 time units) is artificially added to the read and write operations inside the memory module in order to simulate it with realistic timing.

You have to implement hardware support for four new instructions (`lwd`, `lwi`, `swd` and `swi`) in your CPU, to access the new data memory. The new instructions will follow a similar encoding format as previous instructions.

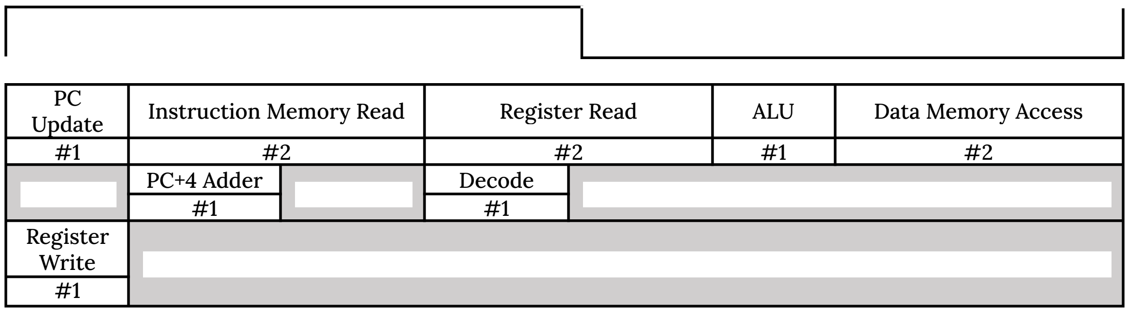
OP-CODE (bits 31-24)	RD (bits 23-16)	RT (bits 15-8)	RS/IMM (bits 7-0)

In the new instructions, memory accessing can be done using two different addressing modes: register direct addressing; or immediate addressing. See examples below:

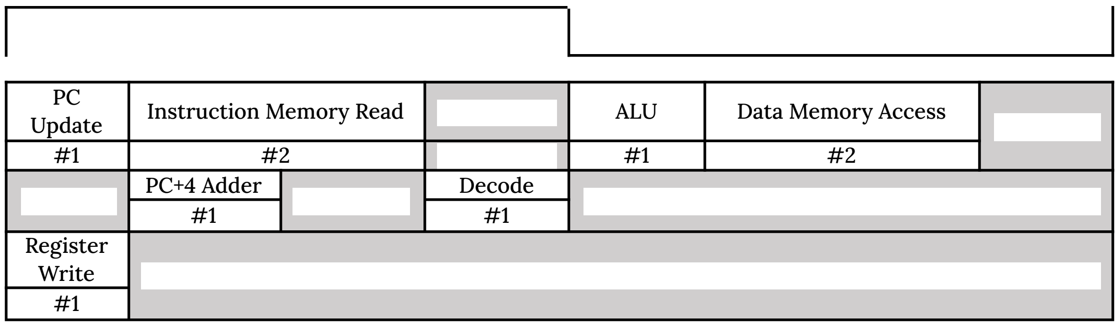
Instruction	Description
`lwd 4 2`	Read memory at address given in register 2 (`RS`) and store result in register 4 (`RD`). Ignore `bits 15-8`
`lwi 4 0x1F`	Read memory at address `0x1F` (`IMM`) and store result it in register 4 (`RD`). Ignore `bits 15-8`
`swd 2 3`	Write value from register 2 (`RT`) to the memory at address given in register 3 (`RS`). Ignore `bits 23-16`
`swi 2 0x8C`	Write value from register 2 (`RT`) to the memory at address `0x8C` (`IMM`). Ignore `bits 23-16`

Datapath timing for the new memory access instructions are given below. Note that Data Memory Access time here is based on ideal caches. Your system doesn't have caches in Practical 5 and accessing memory directly will incur much higher delays, consequently stalling the CPU for several clock cycles.

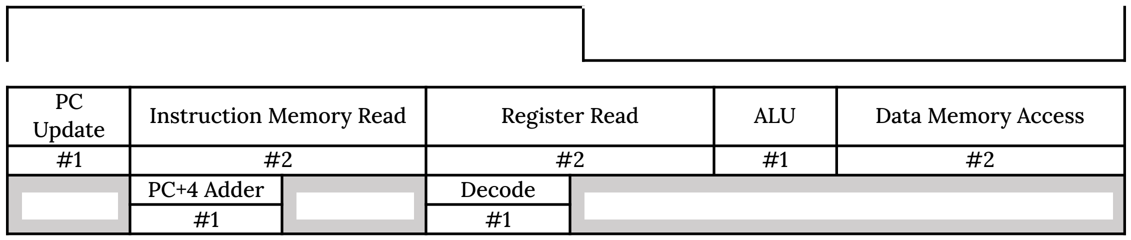
`lwd` (register direct addressing):



`lwi` (immediate addressing):



`swd` (register direct addressing):



`swi` (immediate addressing):

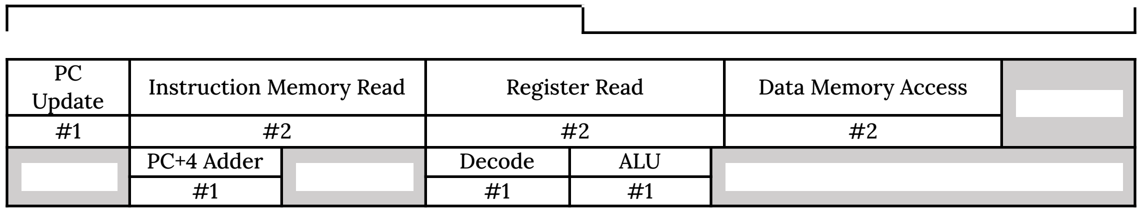


Figure 7: Timing Details for the Memory Access Instructions

1. Connect the given *data_memory* module to your *cpu* as specified, via a testbench. Implement the new instructions, and modify the *cpu* accordingly. Make sure to stall the processor when the `BUSYWAIT` signal is asserted by the Memory. Include **a lot of comments.**
2. Test your processor and memory with several software programs containing the new instructions (in addition to the instructions from Lab 5). The program can be hardcoded inside the testbench or loaded from a file. Store/load data values in the data memory via your program.
3. Obtain timing diagrams clearly showing signals related to memory accesses.

Practical 6 - Data Cache

Now that your CPU can access data in memory, your next task is to implement a simple data cache. The goal of using a cache is to reduce the time spent on accessing memory for most accesses based on locality (make the common case fast!). The data cache will act as an intermediary between CPU and data memory (see Figure 8 below). For the sake of simplicity, your cache module should use the same signals as the memory module in Practical 5 when connecting to the CPU.

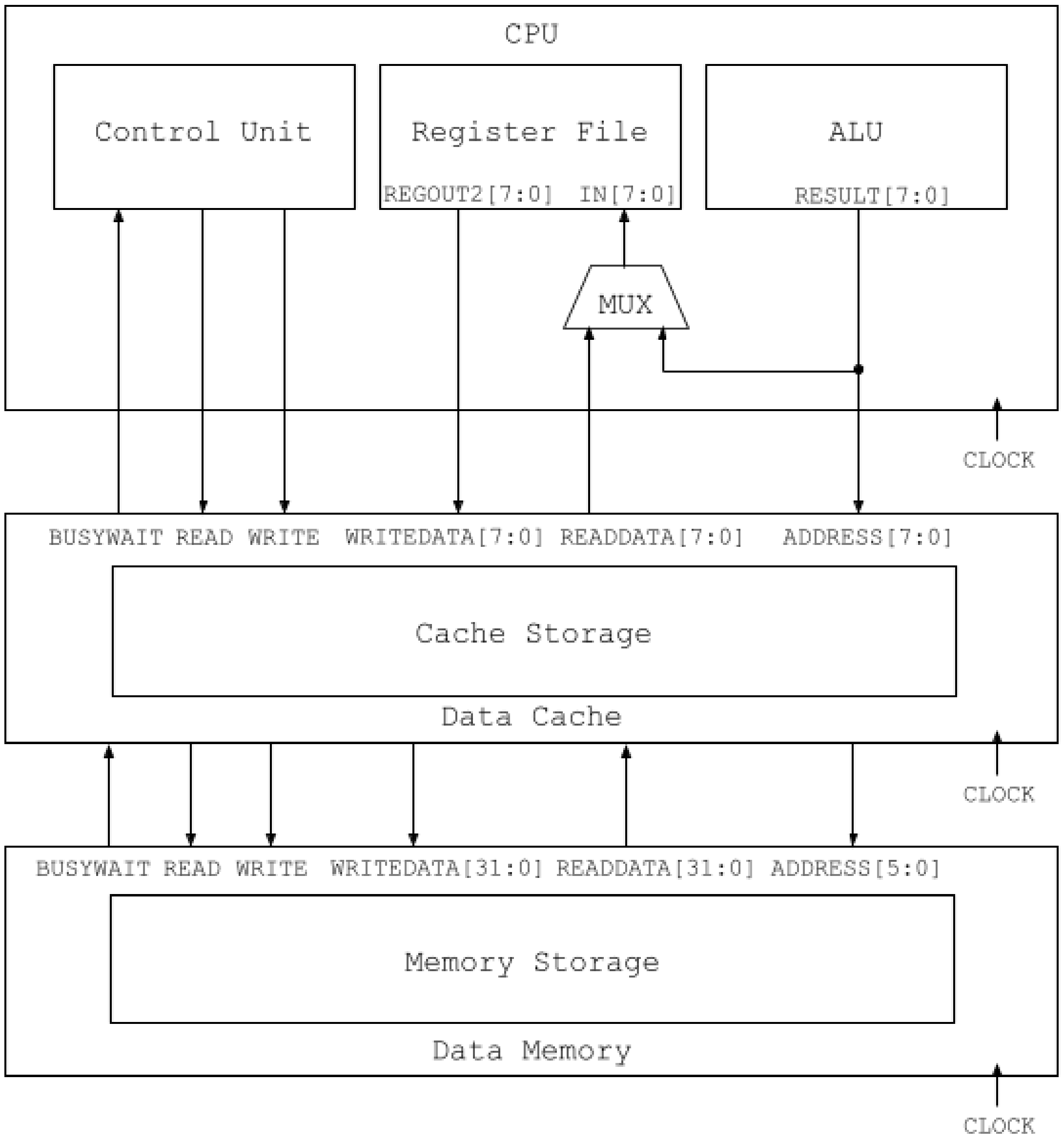


Figure 2: CPU with Data Cache and Data Memory

The following are the parameters to be used when designing your data cache:

- Data ***word-size*** is 1 Byte (8 bits), as defined by the ISA.
- ***Cache size*** is 32 Bytes.
- Use a ***block-size*** of 4 Bytes (∴ cache can hold eight data blocks).
- Use ***direct-mapped*** block placement.
- You need to store the corresponding ***address tag*** along with every data block.
- You need to store a ***valid bit*** along with every data block. At the beginning, all cache entries are empty, therefore invalid.
- You need to store a ***dirty bit*** along with every data block. This should be used to indicate data blocks which are "inconsistent" with the memory. When a block receives a write access, its dirty bit should be set.
- Use ***write-back*** policy for writes. When a block is evicted from the cache AND its dirty bit is set, that block should be written to the memory. An eviction can happen in the event of a cache miss.

In our single-cycle CPU, the address for memory access is made available through the ALU while the `READ/WRITE` control signals are generated early by the control unit (similar to single-cycle MIPS). Therefore, your cache controller should assert its `BUSYWAIT` signal when a `READ/WRITE` control signal is detected, to notify the CPU.

Our CPU expects the memory sub-system to respond in under #2 time units (as `lwd` and `swd` instructions only have #2 time units for memory access, while `lwi` and `swi` instructions have up to #3 time units until the end of the clock cycle). Therefore, the cache must resolve hits within that time, in order to let the CPU continue without stalling in the event of a hit. However, cache misses will consume a longer time and the CPU needs to be stalled in such events. The `BUSYWAIT` signal should be held asserted until either the hit is resolved or miss is handled accordingly.

The CPU accesses a single word at a time (word-size = 1 Byte) using an 8-bit memory address. Your cache should split the address into Tag, Index and Offset sections appropriately. Finding the correct cache entry and extracting stored data block, tag, valid and dirty bits should be done based on the Index. Include an artificial indexing latency of #1 time unit when extracting these stored values. Then perform Tag comparison and validation to determine whether the access is a hit or a miss. Include an artificial latency of #0.9 time units for the tag comparison. These operations should be carried out asynchronously.

Read-hit:

Cache should select the requested data word from the block based on the *Offset*, and send the data word to the CPU asynchronously. Include an artificial latency of #1 time unit in the data word selection. Note that this data word selection latency can overlap with tag comparison, while sending the data to the CPU should be done afterwards based on the hit status. Which means our cache can detect the hit status #1.9 time units after `ADDRESS` is received. Therefore, in the event of a hit, cache controller can de-assert the `BUSYWAIT` signal in order to prevent the CPU from stalling.

Write-hit:

Cache should write the data provided by the CPU to the correct word within the block, based on the *Offset*. Include an artificial latency of #1 time unit in this writing operation. Note that this writing latency **cannot** overlap with tag comparison, as it must depend on the hit status (because we use the write-back policy). The corresponding valid and dirty bits must also be updated at the same time. Since #1.9 time units have already elapsed to detect hit status before writing can be done, cache controller can write the data at the positive edge of the clock (at the start of the next clock cycle). As there is no need to stall the CPU on a write-hit, cache controller can de-assert the `BUSYWAIT` signal once the hit is detected, just like in a read-hit.

Read-miss:

If the existing block is not dirty, the missing data block should be fetched from memory. For this, cache controller should assert the memory `READ` control signal as soon as the miss is detected. Reading the missing block from memory can then start at the positive clock edge. Note that you should use the new memory module for Practical 6, which deals with blocks of 4-Byte data instead of individual Bytes. A 6-bit block-address should be used when the cache is accessing the memory. Reading a block will take a long time (4×5 = 20 cycles), so the cache controller must wait until the memory de-asserts its `BUSYWAIT` signal while holding the relevant signals stable.

If the existing block is dirty, that block must be written back to the memory before fetching the missing block. For this, cache controller should assert the memory `WRITE` control signal as soon as the miss is detected. Writing back the existing block to memory can then start at the next positive clock edge. Writing back a block will also take a long time (4×5 = 20 cycles), so the cache controller must wait until the memory de-asserts its `BUSYWAIT` signal while holding the relevant signals stable.

On the positive clock edge that the write-back completes, the cache should assert the `READ` control signal. Fetching the missing data block from the memory can then start at the next positive clock edge. Which means there is a 1 cycle gap between write-back and fetch events.

After fetching the missing data block from the memory, the cache should write the fetched data block into the indexed cache entry and update the tag, valid and dirty bits accordingly. Include an artificial latency of #1 time unit in this writing operation. Then the original read access can be served by the asynchronous circuitry, where the status of the hit signal will change after further #1.9 time units, and consequently de-assert the `BUSYWAIT` signal of the cache while sending the requested data word to the CPU.

Therefore, the total data miss penalty add up to 42 CPU cycles if the existing block was dirty, or 21 CPU cycles otherwise.

Write-miss:

A write-back should be performed based on the dirty bit of the existing block, then the missing data block should be fetched from memory and written to the indexed cache entry (similar to a read-miss).

Then the original write access can be served using the asynchronous circuitry, where the status of the hit signal will update after further #1.9 time units, and consequently de-assert the `BUSYWAIT` signal of the cache. The data word sent by the CPU should then be written to the indexed cache entry at the start of the next clock cycle.

The miss penalty should be the same as that for a read-miss.

Note that you may design and implement a *finite-state machine* for the part of the cache controller that handles cache misses from the subsequent positive clock edge.

In order to properly simulate fractional delays such as #0.9, you will need to set the timescale of the simulation accordingly. To do that, you can use the `` `timescale <time_unit>/<time_precision> `` syntax in Verilog (you can read more about it from here: https://www.chipverify.com/verilog/verilog-timescale). It's recommended to use a timescale of 1ns/100ps for your implementation. You can include the line to set the timescale (e.g. `` `timescale  1ns/100ps ``) at the beginning of your Verilog file. Also, make sure you include the same timescale in all the Verilog files so that the same timescale is used by all the modules in your design.

1. Implement the data cache module as specified and connect to the CPU via a testbench. Include **a lot of comments.**

2. Test your system thoroughly with several software programs containing data access instructions. Programs can be hardcoded inside the testbench or loaded from a file.

3. Compare your system's performance with the cache-less one from Practical 5, using test programs, and write a brief report.

4. Obtain timing diagrams clearly showing signals related to cache and CPU control.

Practical 7 - Instruction Cache & Memory

Once you have the data cache and data memory working properly, use the same concepts to add an instruction cache and an instruction memory to your system. Note that you will not be using the hardcoded instruction array in your test bench (with #2 time units artificial reading latency) anymore.

The following are the parameters to be used when designing the instruction cache:

- Instruction ***word-size*** is 4Bytes (32 bits), as defined by the ISA.
- ***Cache size*** is 128Bytes
- Use a ***block-size*** of 16 Bytes (∴ cache can hold eight instruction blocks)
- Use ***direct-mapped*** block placement
- You need to store the corresponding ***address tag*** along with every instruction block
- You need to store a ***valid bit*** with every instruction block. At the beginning, all cache entries are empty, therefore invalid.
- Dirty bit and write policies are not needed, as the CPU does not write to the instruction memory. Evicted blocks can simply be discarded.

Your instruction words are 4 Bytes wide. The instruction memory should be able to hold 256 instruction words, making the total size of instruction memory 1024 Bytes. The CPU accesses a single instruction word at a time using a 10-bit word address (from the program counter) where the two least significant bits are zeros.

The instruction cache should split the address into *Tag*, *Index* and *Offset* sections appropriately. Finding the correct cache entry and extracting stored data block, tag and valid bits should be done based on the *Index*. Include an artificial indexing latency of #1 time unit when extracting the stored values. Then perform *Tag* comparison and validation to determine whether the access is a hit or a miss. Include an artificial latency of #0.9 time units for the tag comparison.

Read hits should be handled asynchronously, similar to the data cache, with the only differences being the size of the address and word-size. Include an artificial latency of #1 time unit for selecting the requested instruction word from the block (which can happen in parallel to the tag comparison).

In the event of a miss, the CPU must be stalled using a BUSYWAIT signal. Cache controller should assert the memory READ control signal as soon as the miss is detected and start fetching the missing 16-byte block from the instruction memory on the next positive clock edge.

Note that you should use the separate instruction memory module for Practical 7, which deals with blocks of 16 Bytes. A 6-bit block-address should be used when the cache is accessing the instruction memory. The fetching will take a long time (16×5 = 80 cycles), so the cache must wait (holding the relevant signals stable) until the memory de-asserts its `BUSYWAIT` signal to retrieve the instruction block.

On the positive clock edge this memory read completes, the cache controller should write the fetched block into the indexed cache entry and update the tag and valid bit accordingly. Include an artificial latency of #1 time unit in this writing operation. Then the original read access can be served by the asynchronous logic, where the status of the hit signal will change after further #1.9 time units, and consequently de-assert the `BUSYWAIT` signal of the cache at the next positive clock edge while sending the requested instruction word to the CPU.

Therefore, the total instruction miss penalty adds up 81 CPU cycles.

1. Implement the instruction cache module as specified and connect to the CPU via the testbench, along with the instruction memory module. Include **a lot of comments.**

2. Test your system thoroughly with several software programs. Programs can be hardcoded inside the instruction memory module or loaded from a file.

3. Obtain timing diagrams clearly showing signals related to instruction cache and CPU control.

Practical 8 - Extended ISA

"There's a Practical 8?!? You didn't tell us!" Well, <u>only if you are up for the challenge</u>, extend your processor to support the following instructions as a bonus task.

Instruction	Description
`mult 4 1 2`	Multiply the value in register 1 by the value in register 2, and place the result in register 4
`sll 4 1 0x02`	Apply logical shift left 2 times on the value in register 1, and place the result in register 4
`srl 4 1 0x02`	Apply a logical shift right 2 times on the value in register 1, and place the result in register 4
`sra 4 1 0x02`	Apply arithmetic shift right 2 times on the value in register 1, and place the result in register 4
`ror 4 1 0x02`	Apply rotate right 2 times on the value in register 1, and place the result in register 4
`bne 0x02 1 2`	If the values in registers 1 and 2 are not equal, branch 2 instructions forward

Note that you should use the same 3-bit `ALUOP` signal as in the previous part. Since you can only have 8 functional units inside the ALU with the 3-bit ALUOP control signal, you need to find a way for the new instructions to share functional units (*hint*: *can* `sll` *and* `srl` *share a single functional unit*?). You are discouraged to use additional control signals, unless absolutely necessary.

You must make reasonable assumptions for the latencies of any added hardware, and make sure that the new instructions can still complete within one clock cycle (8 time units).

As this is a bonus exercise, you must implement any new functional unit modules without using simple operations (e.g. using operator << for left shifting will not be acceptable).

You must provide a clear description of the instruction encodings, assigned opcodes, timing, and changes made to the datapath+control as a separate report. Two instructions correctly implemented with proper documentation will earn you 4 bonus marks. Thereafter, you can earn 4 more marks per additional instruction with proper documentation. Submit a compressed file groupXX_lab5_part5.zip containing all files of your design along with a testbench and your report.

Have fun coding. May the force be with you!

Students' Guide to Practicals on Computer Architecture

By Isuru Nawinne

9 786249 291300