Danmarks Tekniske Universitet

February 9, 2025

Work distribution

We split the work evenly between group members and everyone had a input in all parts of our project both for the code and the report.

1 Design

1.1 Introduction

The design for our CPU hardware architecture was based off a MIPS processor and supports a slightly reduced version of the MIPS instruction set. With this instruction set, assembly code was written to compute the erosion algorithm covered in Assignment 1. This code was then run on a simulation of the CPU and its performance is compared to CPU performance in use today.

1.2 Separation of concerns

The CPU consists of a series of modules that were each implemented and tested individually. Implementations of the program memory and data memory were given to us, and we from there implemented the control unit, ALU, program counter and register file.

The Control Unit's purpose is to set values to true depending on the Opcode it receives. Our CPUTop wires together the memory, control unit and ALU. Additionally the CPUTop also includes a few Mux operations, which are dependent on the output of Control Unit. Other than the "statically" assigned wires in CPUTop, the only way it can be controlled is by what Control Unit outputs. Hence we have a solid implementation of our CPU architecture, where each component serve a specific purpose.

During the planning phase we focused on ensuring that the Control Unit was closely coupled to the CPUTop. The Control Unit outputs only boolean values telling the other components in CPUTop what action they should take (e.g. read/write from register file, stop/run the program counter...) with the notable exception of the ALU. Due to the complexity of operations that the ALU must support, we decided to pass the opcode directly to the ALU. The ALU then decides the correct output on a case-by-case basis.

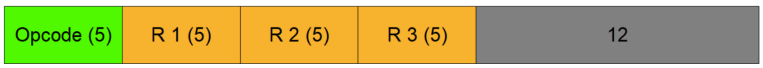
1.3 Instruction set

The instruction set consists of the following operations:

Command	Positions	Description
ADD	ADD R1 R2 RD 32	RD = R1 + R2
SUB	SUB R1 R2 RD	RD = R1 - R2
MULT	MULT R1 R2 RD	RD = R1 * R2
LD	LD R1 RD 32	Load the data at address R1 into RD
SD	SD R1 R2	Set the data address of R1 to the data in R2
ADDI	ADDI R1 RD IMM	RD = R1 + IMM
SUBI	SUBI R1 RD IMM	RD = R1 - IMM
LI	LI RD IMM	RD = IMM
SDI	SDI R1 IMM	Set the data address of R1 to IMM
JGT	JGT R1 R2 IMM	Jump to line IMM if R1 ¿ R2
JE	JE R1 R2 IMM	Jump to line IMM if R1 = R2
JR	JR IMM	Jumps to IMM
END	END	Ends execution

1.4 Instruction Encoding

We have two types of instruction encoding of which all instructions are subsets of. If a register is not used we have defaulted the value of it to 0 in our code:



The instructions that go under this type of encoding are as follows:

1. ADD, SUB and MULT where R1 and R2 are read registers and R3 is the write register
2. LD where R1 is the read register, R2 is the write register, and R3 is unnecessary
3. SD where both R1 and R2 are read registers and no write register is needed

The instructions that go under the other type of encoding are the following:

1. ADDI and SUBI where R1 is the read register and R2 is the write register
2. LI where only R2 has to be specified as a write register and R1 is unneeded
3. SDI where R1 is a read register with the corresponding memorry adress stored and R2 is unneeded
4. JGT and JE, where both R1 and R2 are read registers
5. JR where no registers are needed and only the Immediate has to be specified

Finally there is also the END instruction which does not follow any of these patterns since only the opcode has to be specified for it to work.

Despite for example the JR instruction only using an opcode and immediate address, we still chose to group it with the two registers and immediate instead of for example an opcode and 27 bits for immediate. This was in order to simplify our architecture. It we had allowed for more bits for the opcode and immediate, then we would need additional mux logic in the CPUTop to keep track of when to use either 17 or 27 bits. The consequence of this is that if we use an opcode, 2 register and an immediate then we can jump to fewer places in contrast to the single opcode and immediate. We decided that this wasn't a large problem since our program would never contain more lines of instructions than what could be stored in 17 bits.

1.5 CPU Architecture

We based our CPU architecture off a previous architecture that supports the MIPS instruction set. We however made some changes to the architecture to support extra operations that MIPS does not support. Most importantly, our CPU supports store data immediate (SDI) and load immediate (LI). A diagram of the layout used can be seen in figure 1.

The immediate operations are especially useful, as it simplifies the usage of registers. We do not need to perform an ADDI operation to load in a value to a register. Instead an immediate value can be loaded in. When we wrote the erosion program with our instructions, we could use far less lines, which reduced the possibilities of making errors.

Our usage of immediate operations also meshes well with the fact that we currently only support 16 registers. If the immediate operations were omitted we would have to also write instructions to set temporary registers.

For the program counter to start ticking up and the therefore make the program run CPUTop.io.run has to be set to true. This is not done in the actual program counter but for example done in the CPUTopTester.scala file

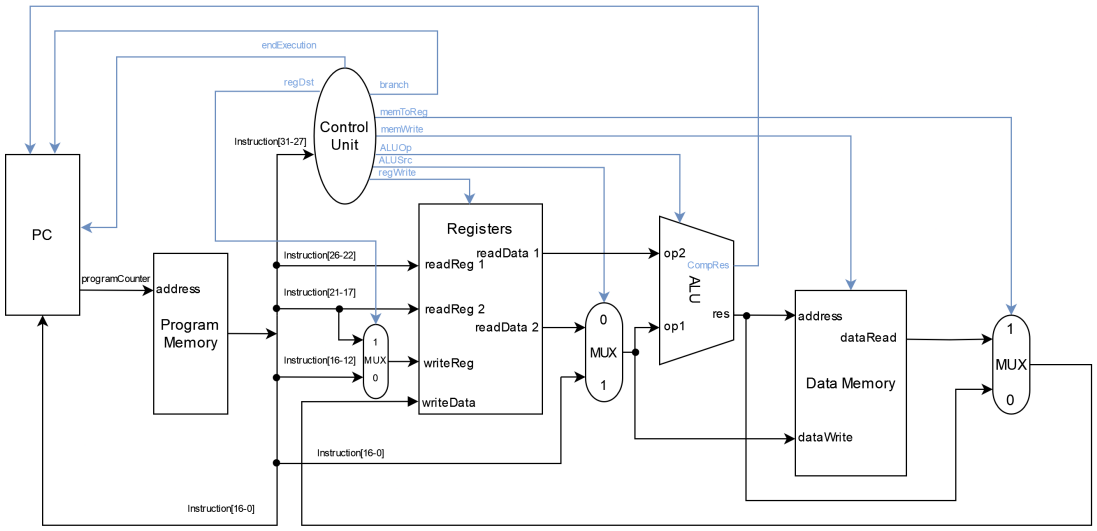


Figure 1: The MIPS-like architecture implemented.

1.6 Erosion algorithm in Assembly

```
LI X 0
LI CEILX 19
LI WIDTH 20
LI OUTIMGADDR 400
LI INIMGADDR 0
LI ZEROREG 0
[XSTART] JGT X CEILX [XFORLOOPEND]
    LI Y 0
    LI CEILY 19
[YSTART] JGT Y CEILY [YFORLOOPEND]
    MULT Y WIDTH TEMP
    ADD OUTIMGADDR TEMP OUTPIXELADDR
    ADD OUTPIXELADDR X OUTPIXELADDR
    MULT Y WIDTH TEMP
    ADD INIMGADDR TEMP INPIXELADDR
    ADD INPIXELADDR X INPIXELADDR
    JEQ X ZEROREG [ZEROSET]
    JEQ Y ZEROREG [ZEROSET]
    JEQ X CEILX [ZEROSET]
    JEQ Y CEILY [ZEROSET]
    LD INPIXELADDR INPIXELVALUE
    JEQ INPIXELVALUE ZEROREG [ZEROSET]
    SUBI X TEMP 1
    MULT Y WIDTH TEMP2
    ADD TEMP2 TEMP NEIGHBORADDR
    ADD NEIGHBORADDR INIMGADDR NEIGHBORADDR
    LD NEIGHBORADDR NEIGHBORVALUE
    JEQ NEIGHBORVALUE ZEROREG [ZEROSET]
    ADDI X TEMP 1
    ADD TEMP2 TEMP NEIGHBORADDR
    ADD NEIGHBORADDR INIMGADDR NEIGHBORADDR
    LD NEIGHBORADDR NEIGHBORVALUE
    JEQ NEIGHBORVALUE ZEROREG [ZEROSET]
    SUBI Y TEMP_Y 1
    MULT TEMP_Y WIDTH TEMP2
    ADD TEMP2 X NEIGHBORADDR
    ADD NEIGHBORADDR INIMGADDR NEIGHBORADDR
    LD NEIGHBORADDR NEIGHBORVALUE
    JEQ NEIGHBORVALUE ZEROREG [ZEROSET]
    ADDI Y TEMP_Y 1
    MULT TEMP_Y WIDTH TEMP2
    ADD TEMP2 X NEIGHBORADDR
    ADD NEIGHBORADDR INIMGADDR NEIGHBORADDR
    LD NEIGHBORADDR NEIGHBORVALUE
    JEQ NEIGHBORVALUE ZEROREG [ZEROSET]
    SDI OUTPIXELADDR 255
    JR [CONTINUE]
[ZEROSET] SDI OUTPIXELADDR 0
[CONTINUE] ADDI Y Y 1
    JR [YSTART]
[YFORLOOPEND] ADDI X X 1
    JR [XSTART]
[XFORLOOPEND]
```

The assembly code showcased uses variable names for both registers and jump locations where jump locations are specified in square brackets. This is to make the code more understandable for the person reading it. To convert this assembly to actual instructions we developed a python script that could compile it to the correct 32-bit instructions.

2 Implementation

2.1 Control Unit

ControlUnit contains the logic which sets up the data flow through the CPU for each instruction. It takes as input the op code of the instruction, and uses a switch statement to determine the correct output. Its output consists of boolean values which are used as *sel* for muxes in CPUTop and the individual components. The result is that the individual components only receive information relevant to their function. For example, the Register file only receives *regWrite* from ControlUnit, but it does not know what specific operation is being carried out.

2.2 ALU

The only exception to this is the opcode which is passed to the ALU. The original opcode is passed to the ALU which also has a switch statement to determine the appropriate operation to perform on the operands. The result of this is that the ControlUnit and ALU are more tightly coupled, however this also simplifies the communication interface between the two modules.

we debated whether the operation decoding for the ALU should be put in CPUTop or in the ALU itself. We ultimately decided to move the logic to the ALU to ensure each module has a single responsibility.

2.3 Program Counter

Our program counter is also designed to work independently. This means at it ensures that the next instruction address is automatically updated based on the information that it receives from the instruction memory and the Control unit.

2.4 CPUTop

CPUTop was designed with a minimum of logic. It connects the different components, with some muxes to control the input to each value.

3 Test and analysis

3.1 Unit tests

Unit tests were implemented first in the ALU, program counter, and the register file. Care was taken to ensure that every possible combination of inputs to these modules was tested (including ensuring that the register only writes data when ‘regWrite‘ is set to true). Unit tests were also implemented for ControlUnit for each instruction in the instruction set. The expected output was derived from the diagram in 1 by manually drawing the data flow through the circuit for each instruction.

Tests in ALUTester were moved into a function so that the outputs could be verified with a range of numbers.

3.2 Integration tests

Our integration test was written in Program1 and implemented in CPUTopTester. Currently, the output image of the program is printed to the terminal and is easily compared visually to the expected output. During development we also wrote smaller integration tests (such as testing an individual command or constructing a simple for loop). These smaller programs were not kept as Program1 implements all of them already.

3.3 Theoretical running time

The algorithm took 7159 cycles to run erosion on the 20x20 image. Expanding this to a 950x950 image gives an expected running time of $7159 \times (950/20)^2 = 16'152'493$ operations. Therefore the running time per clock cycle would be:

- 64MHz: $16152493/(64 * 10^6) = 0.25$ seconds
- 500MHz: $16152493/(500 * 10^6) = 0.032$ seconds
- 1GHz: $16152493/(1 * 10^9) = 0.016$ seconds
- 2GHz: $16152493/(2 * 10^9) = 0.008$ seconds
- 2GHz: $16152493/(2.6 * 10^9) = 0.0062$ seconds

3.4 Comparison with PC

When run on a 2.6GHz Intel Mac, the same erosion algorithm (written in C and compiled with GCC) ran in 1.142 milliseconds, almost 5.5 times faster than expected with our CPU. Possible reasons for this can be seen in subsection 3.6.

We do not have a compiler. We have only made an assembler which converts the instructions over to machine code. With C we have compiled our program using gcc. Due to the abstraction layer which comes with the code we strictly don't know what the compiled code actually looks like. However we know for a fact that it is performing optimizations on our code.

Additionally our CPU architecture does not perform branch predictions. For the CPU architecture that C gets compiled to, it does perform predictions to speed up the execution time.

3.5 Size of CPU

At an estimate, the CPU would contain:

- 16 32-bit registers for the register file
- 1 16-bit register for the program counter
- 3 1-bit muxes in CPUTop (alu.io.op2 and dataMemory.io.dataWrite use the same Mux)
- one 4-bit multiplexer in ALU and ControlUnit
- 2^{16} 32-bit memory (256 kB) for ProgramMemory and data memory

3.6 Improvements and extensions

Possible improvements on the design of the CPU include:

- More advance compiler: The algorithm was written by hand instead of being compiled from a higher level language like C. Being compiled by a more advanced compiler such as GCC might optimise the assembly in places where we have not.
- Expanded instruction set: While the instruction set implemented is able to perform every operation needed to run a program, adding more specialized instructions (such as a conditional move, which moves a value without requiring a jump instruction) could improve both the readability of the assembly and the running time of the program.
- Pipelining: Most modern CPUs allow multiple instructions to flow through the CPU in stages, in a process called pipelining. MIPS does not have pipeline stages and this may decrease running time.

- Parallel processing: A variety of other parallel processing techniques (such as multi-threading) designed to operate on a collection of data at a time could be introduced to significantly speed up the running time of the program, at the expense of making the architecture and assembly more complex.

It should also be noted ChatGPT was partly used in the writing of the make file and for fixing certain errors between Mac and Windows.