Core Components

Carl-Johannes Johnsen

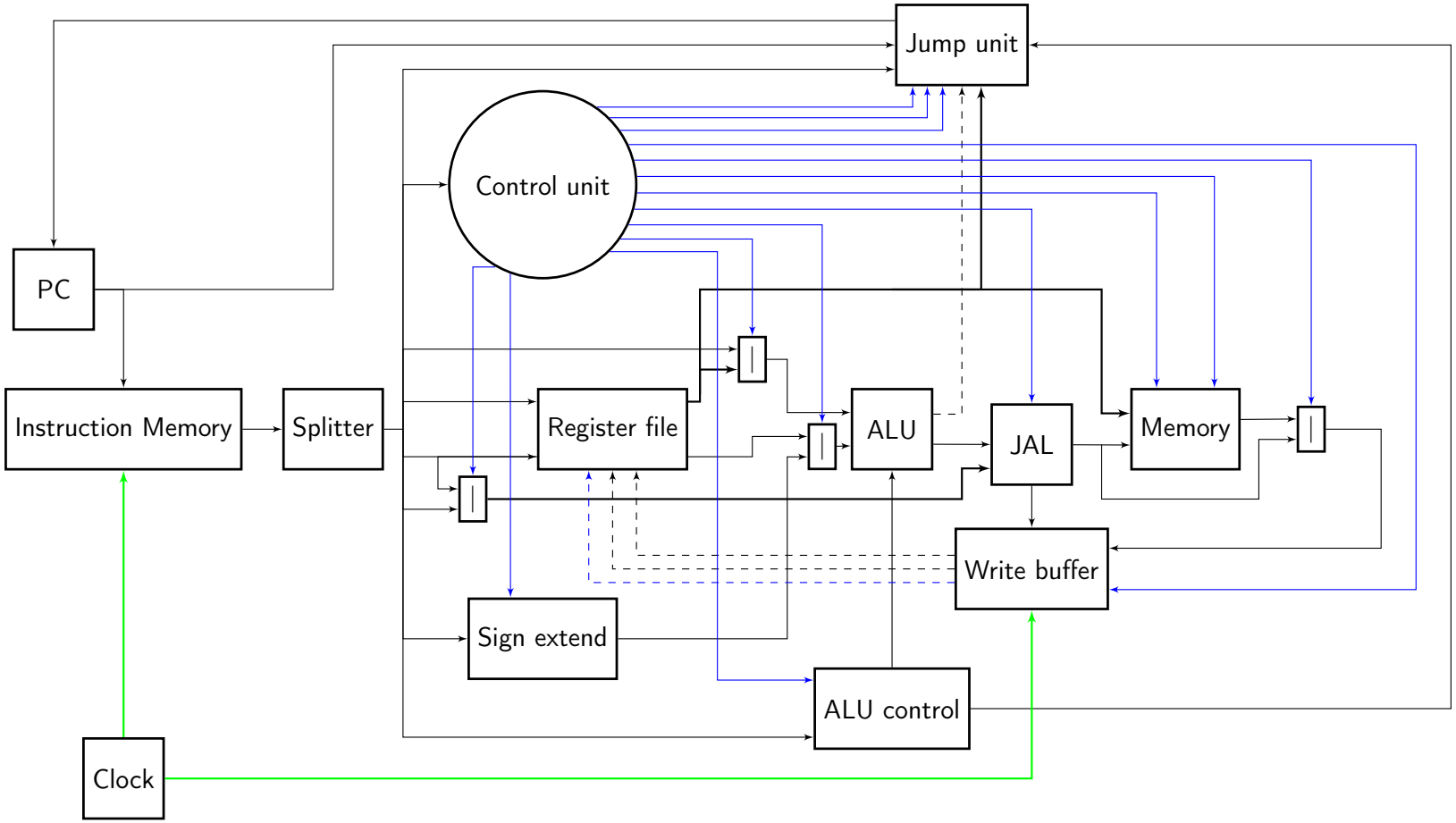
Department of Computer Science
University of Copenhagen

March 7, 2017

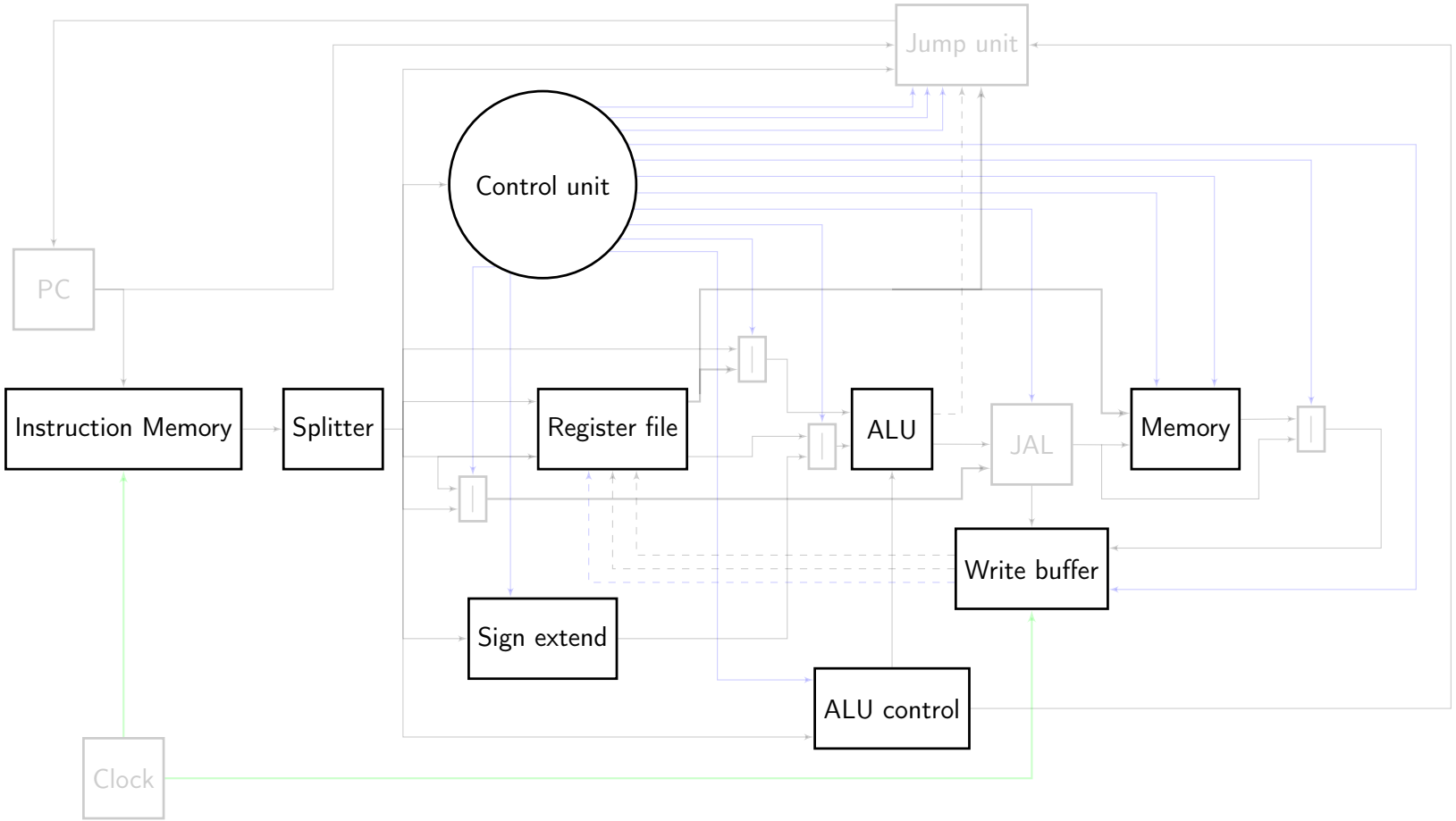
In this lecture, we will be looking at the major components of the MIPS processor.

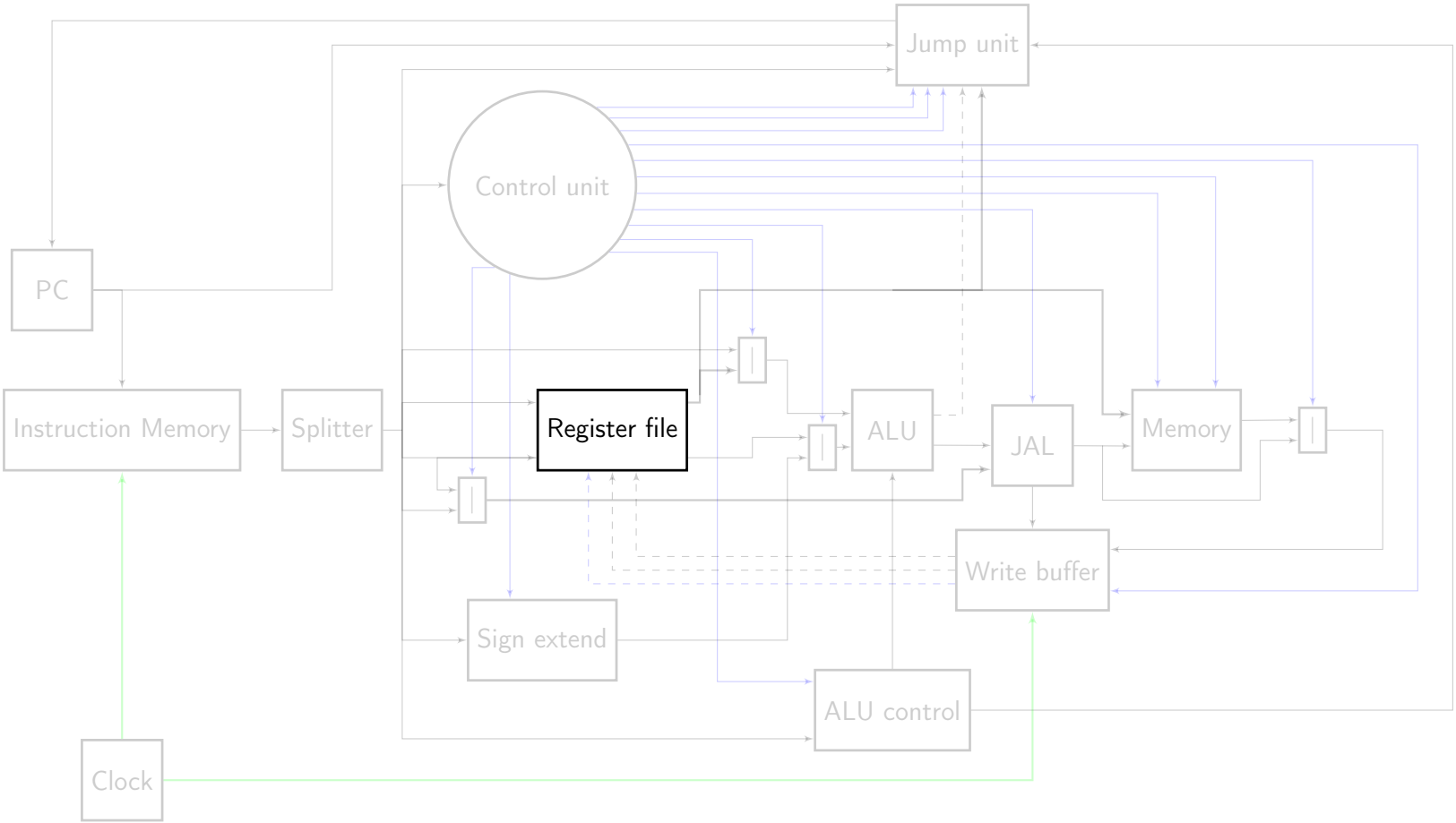
By combining these components, we should be able to construct a single cycle MIPS processor.

Overview of the processor



Overview of the processor




Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The register file is the component that holds values for the processor. It is the first step in a memory hierarchy, and is thus the fastest memory available.

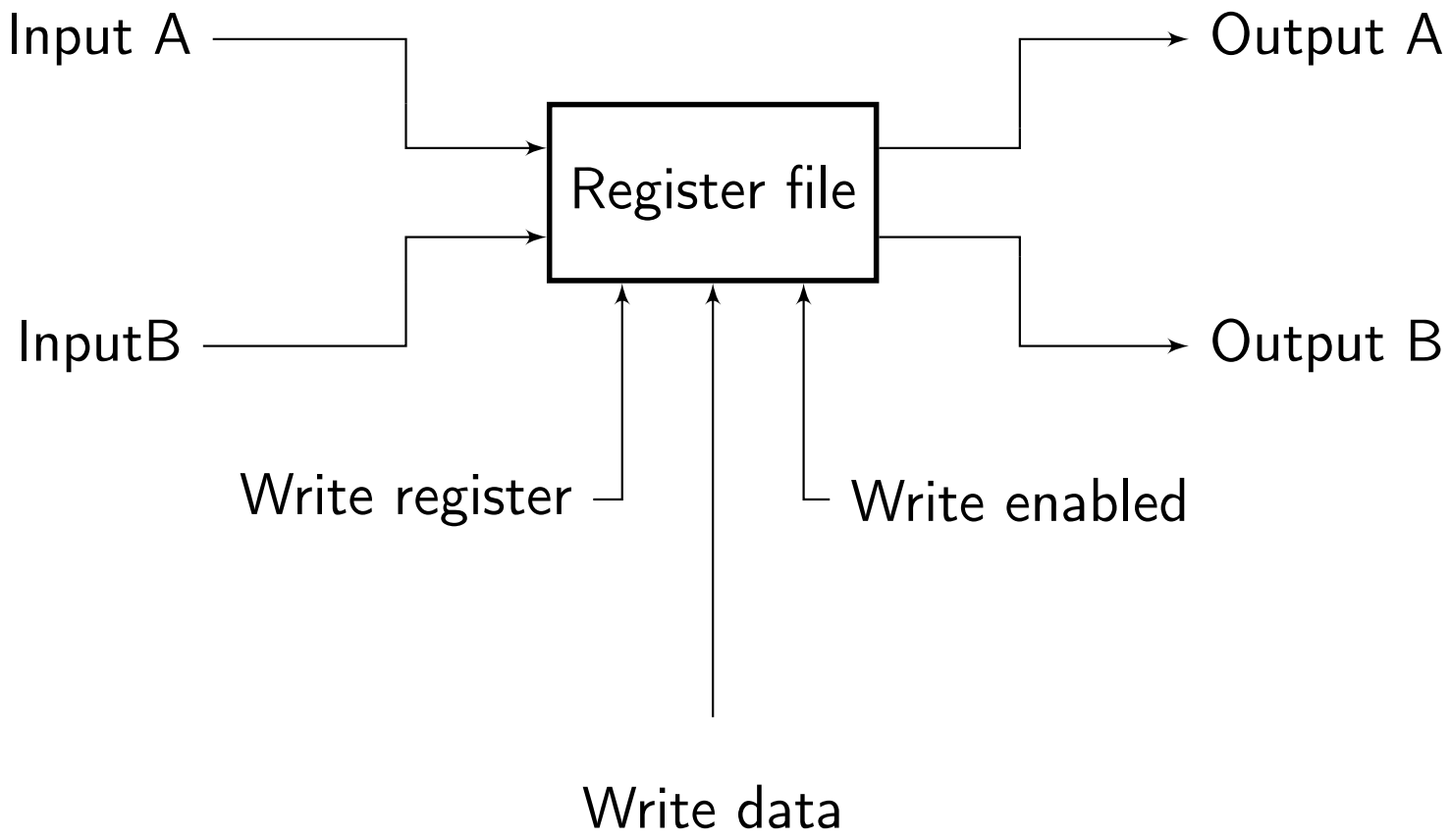
There are 32 registers in a 32-bit MIPS processor, each holding a 32-bit value. The registers are divided into groups based on their usage. This does not matter from a hardware perspective, except for register 0, which is immutable and always 0.

A register file has 5 inputs:

- Read address A
- Read address B
- Write enabled
- Write address
- Write data

And it has 2 outputs:

- Output A
- Output B


Input A
Output A
Register file
InputB
Output B
Write register
Write enabled
Write data

It is important to ensure that when an instruction reads from the register file, it should always get the latest data. I.e. if a instruction reads from the same register as the previous instruction wrote to, it should read the value written by the previous instruction.

This is easy to fix in the single cycle processor, as we just need to write before reading.

To implement the register file in SME, we construct a process holding an int array. It should have the inputs and outputs as specified.

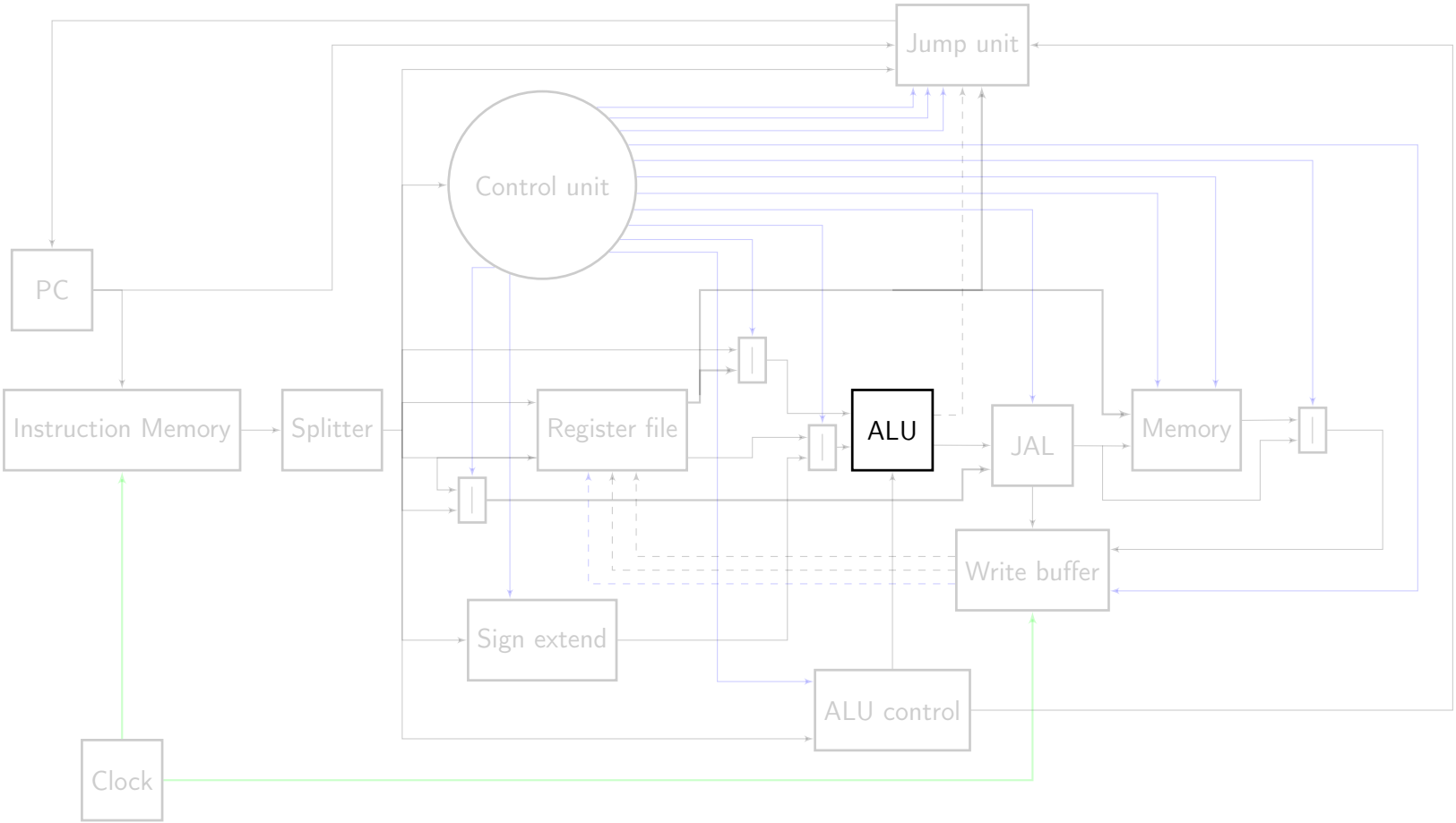
Upon receiving input, it should write the write data into the array at the given write address, if the `RegWrite` flag has been set. Then it should output the value at read address A on output A, and the analogous for read address B and output B.

Note: remember that register 0 is immutable and always 0.

Testing the register file is very trivial. We construct a tester process, which sends some values on the write data bus, along with some addresses and the `RegWrite` flag set.

Then it just sends some addresses on the Read address A and Read address B buses, and verifies that the register file outputs the values stored at these addresses.

It is also important to verify that the behaviour of register zero is as it should be.


Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

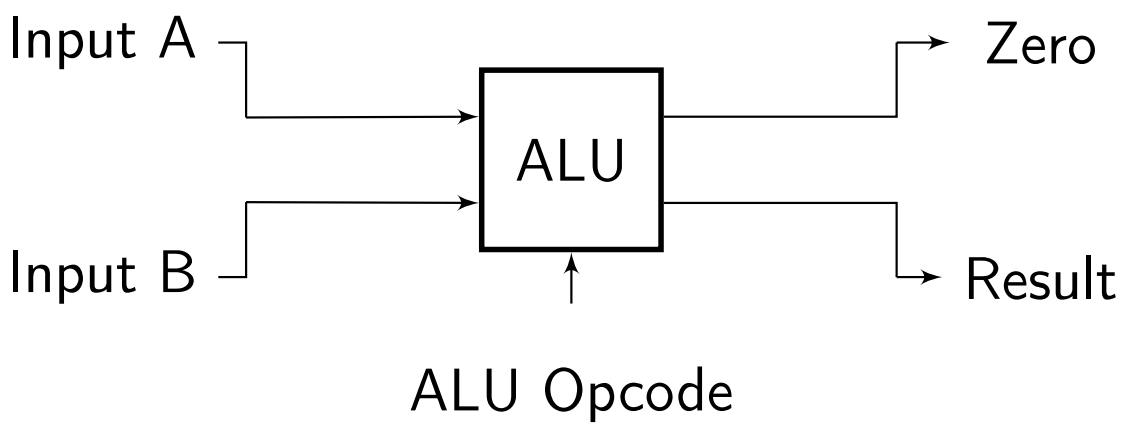
The ALU (Arithmetic Logic Unit) is the part of the processor, which makes the actual computation.

It has three inputs:

- Input A
- Input B
- ALU opcode

and produces two outputs:

- Result
- Zero flag


Input A
Input B
ALU
Zero
Result
ALU Opcode

Upon receiving all of its inputs, it takes the values from Input A and Input B, performs the operation specified by the ALU opcode on them, outputs the result on the Result bus, and finally sets the Zero flag to 1, if the result was 0.

As with the register file, to implement the ALU, we construct a process in SME. It should have the inputs and outputs as specified.

The process should start by reading the values from Input A and Input B. Then, it should `switch` on the value from the ALU opcode, and based on that perform the associated operation. Finally it should output on the Zero flag, if the result of the computation was 0.

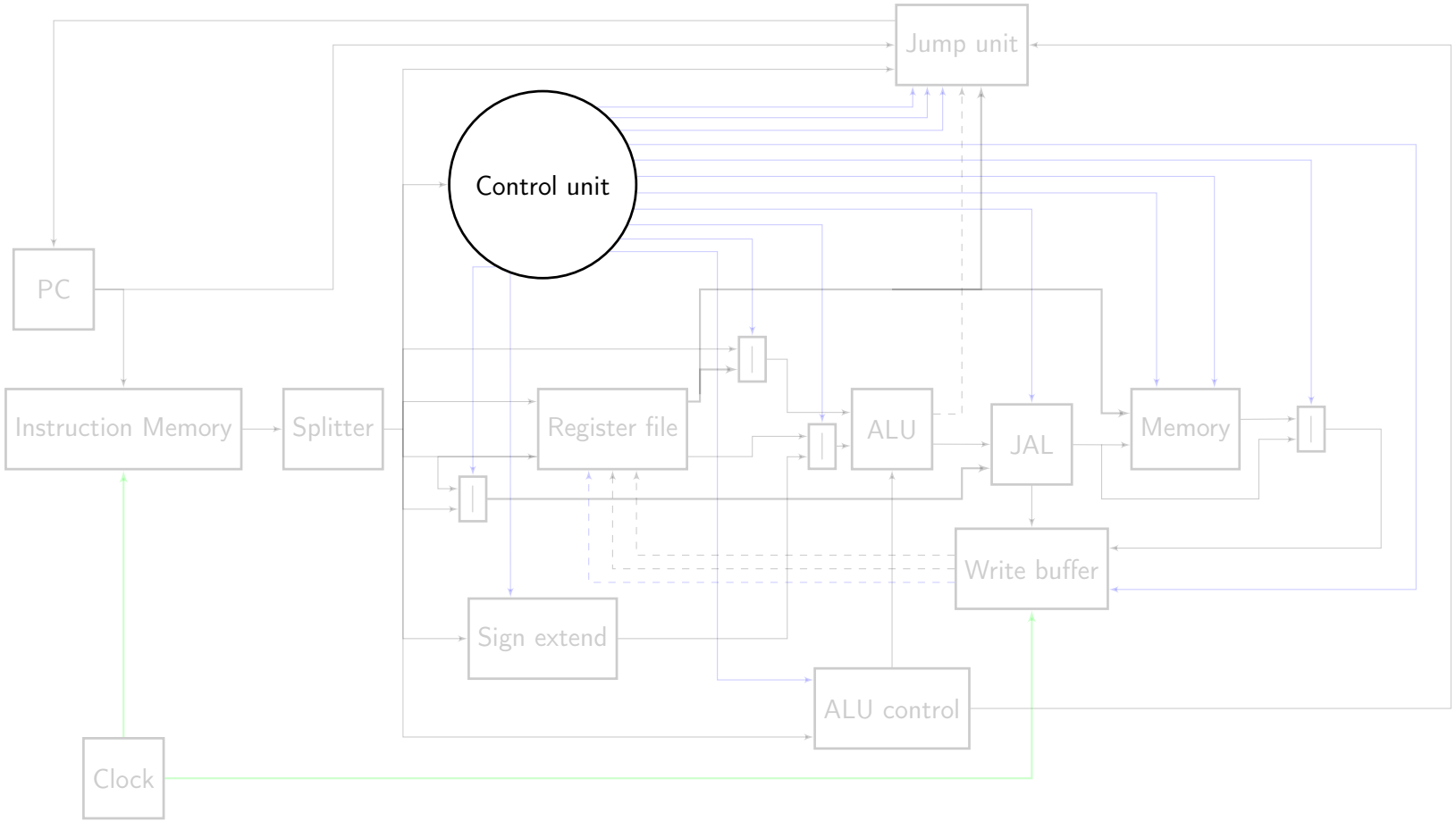
Note: the `switch` on the ALU opcode can become more human readable by using an `enum`

We follow the procedure in the book, and make sure the ALU support the following operations:

- `add`
- `sub`
- `and`
- `or`
- `slt`

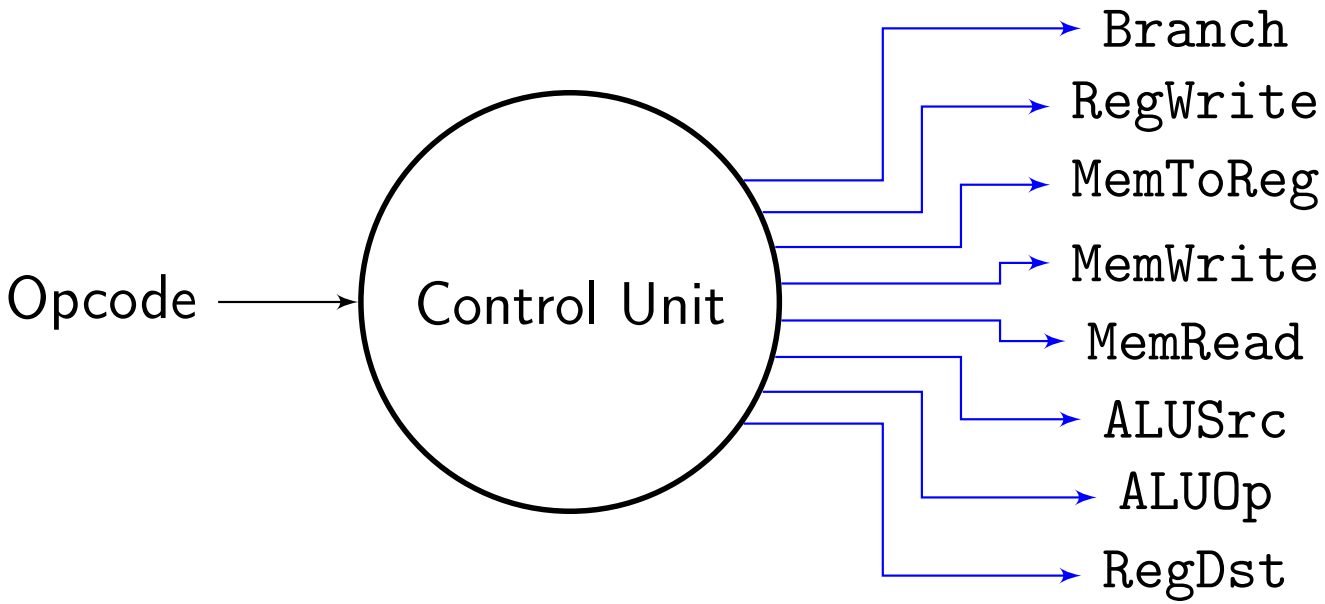
We will be adding more operations later on.

Testing the ALU is again, very trivial. We should construct a tester process, which sends values on the Input A, Input B and ALU opcode busses, and verifies that the value on the Result bus and the Zero flag is as expected.


Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The Control Unit is part of the decoding step. It takes the opcode of the instruction, and based on the opcode, it sets control flags, which are used troughout the processor.
Since we are following the book, we start by having the following flags:

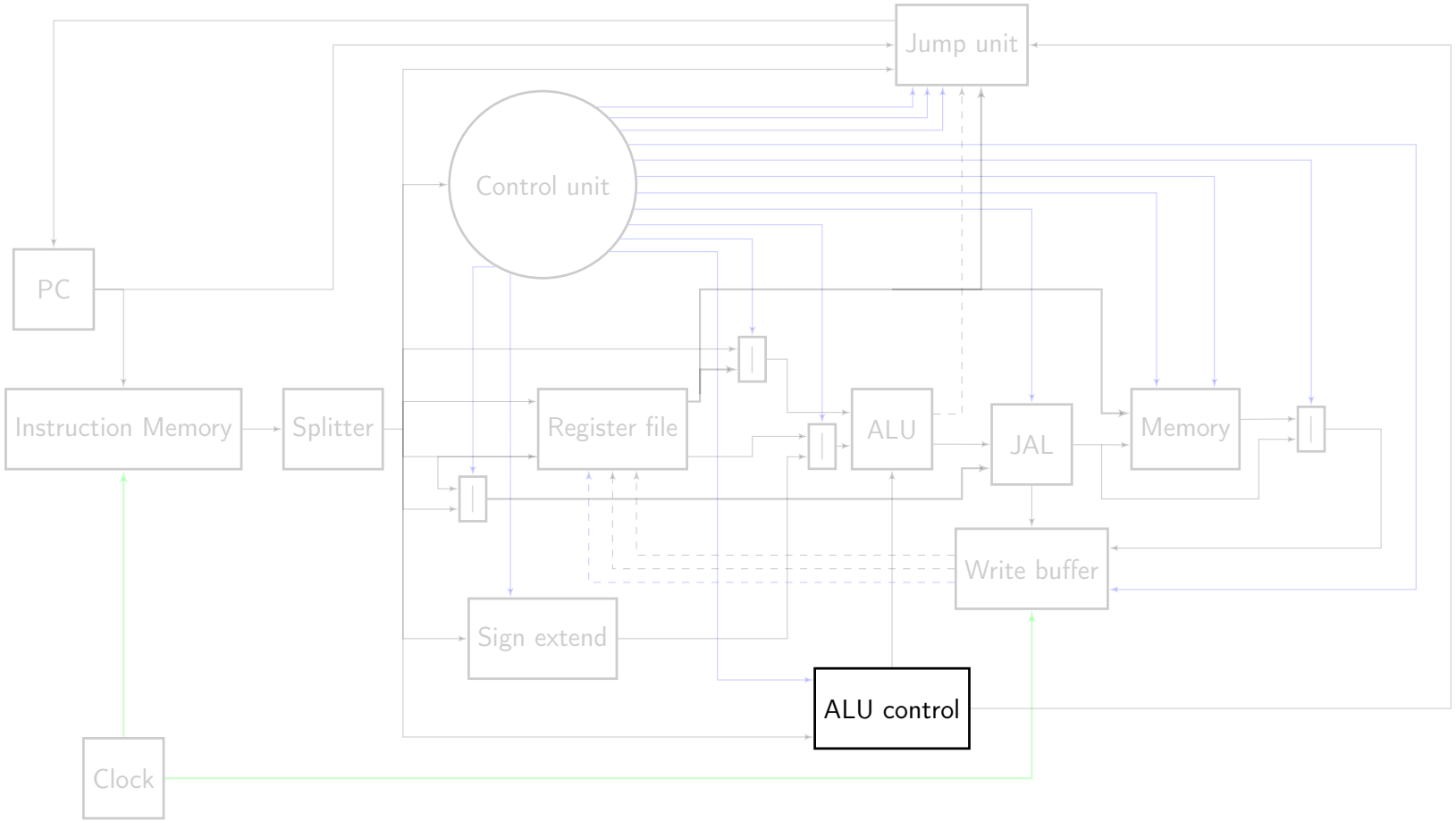
- `RegDst`
- `Branch`
- `MemRead`
- `MemToReg`
- `ALUOp`
- `MemWrite`
- `ALUSrc`
- `RegWrite`


Opcode
Control Unit
Branch
RegWrite
MemToReg
MemWrite
MemRead
ALUSrc
ALUOp
RegDst

To implement the Control Unit, we have a `switch` statement on the opcode. For each case, we set the flags accordingly.

Note: as with the ALU, the source code becomes more readably with a `enum` on the opcode and on the ALUOp.

As before, we construct a tester process, which sends different opcodes to the Control Unit, and verifies its output. It should be tested whether or not all of the expected input opcodes performs as expected.

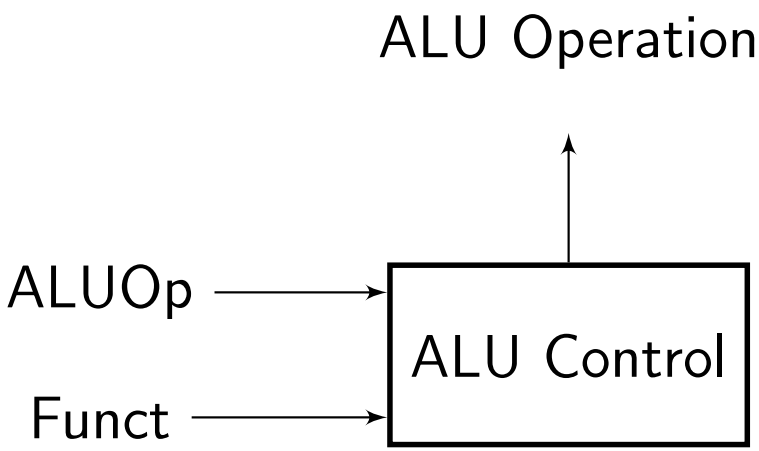

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The ALU Control is used for generating the ALUOp for the ALU. It takes two inputs:

- ALUOp
- Funct

And produces one output:

- ALU Opcode


ALU Operation
ALUOp
Funct
ALU Control

The ALU Control starts by looking at the ALUOp from the Control Unit, to identify whether or not the instruction is in R-format, in which case it should look at the Funct.

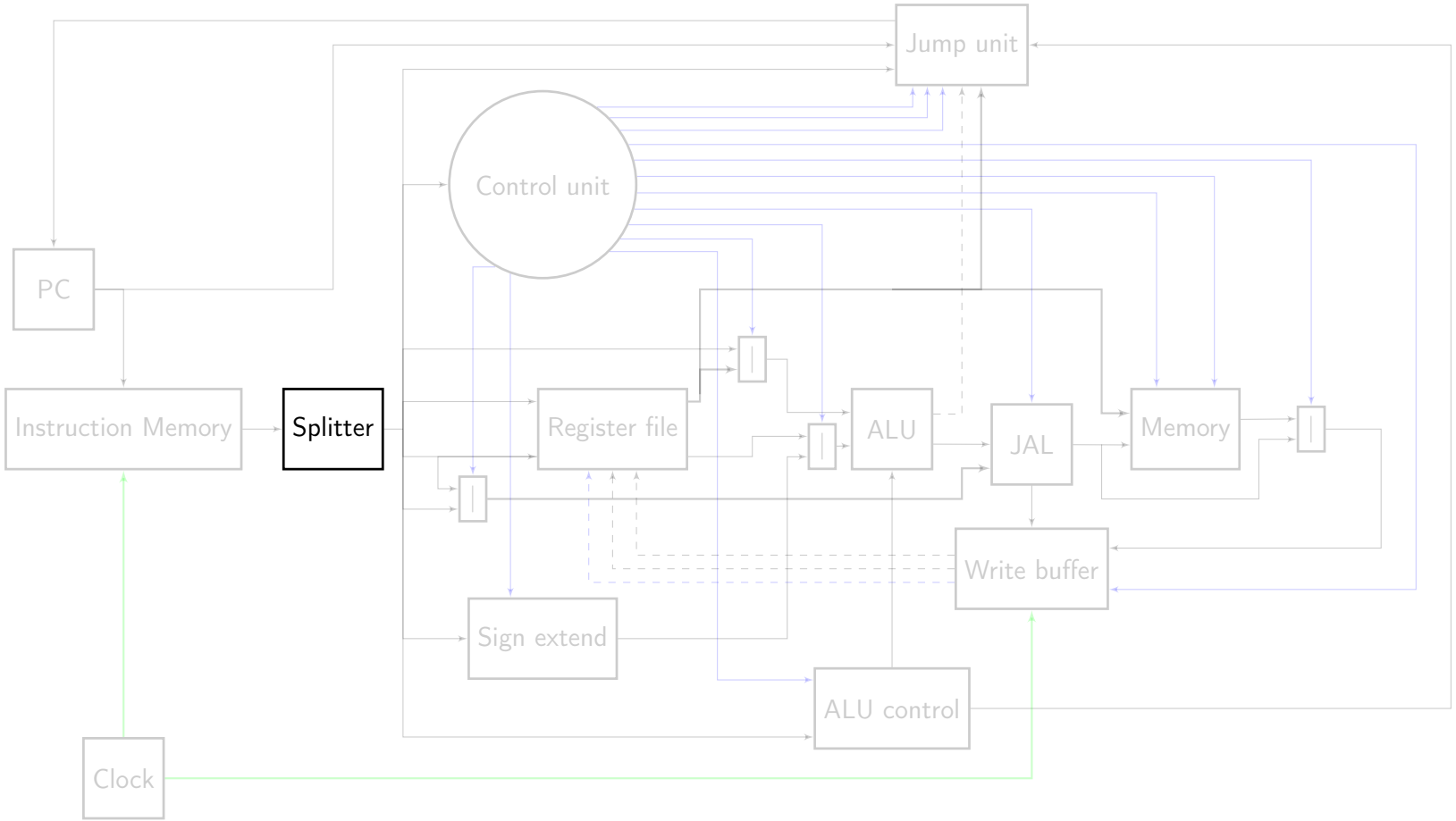
In either case, it outputs on the ALU Operation bus, which operation the ALU should perform.

The SME process should start by looking at the opcode, and have an if on whether or not it is in R-format.

In case it is, it should have a `switch` on the Funct input. If it is not, it should have a `switch` on the opcode. In each case of either of the `switchs`, it should output to the ALU Operation bus.

Note: as always, the source code becomes more readable with a `enum` on the Funct.

The testing is performed in the same manner as with the Control Unit.

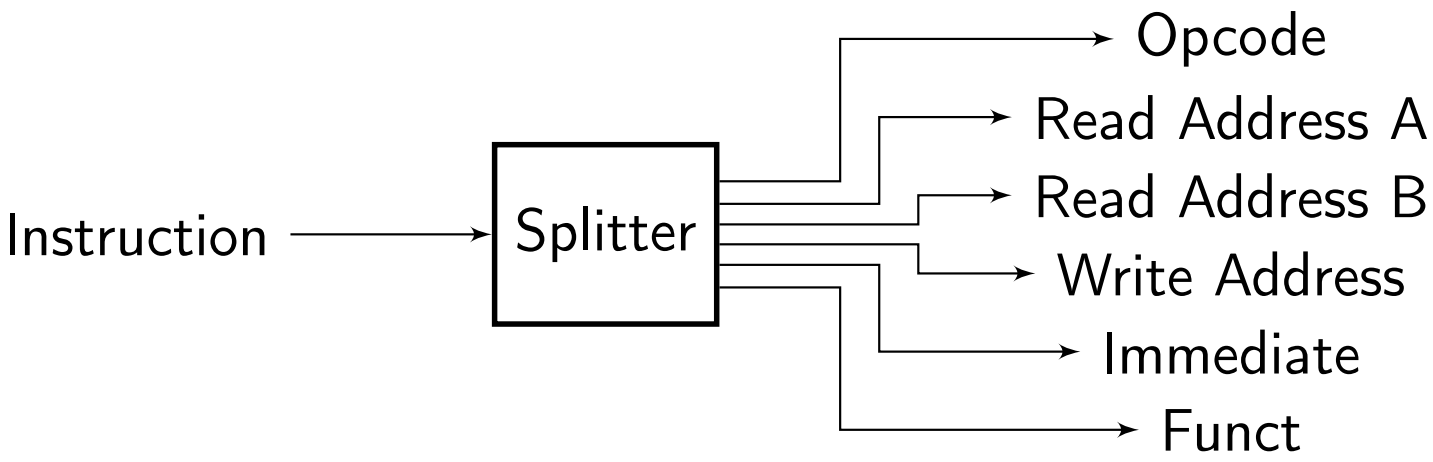

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The Splitter is a very simple component. It takes the instruction read from memory, and splits it up onto multiple busses. It has one input:

- Instruction

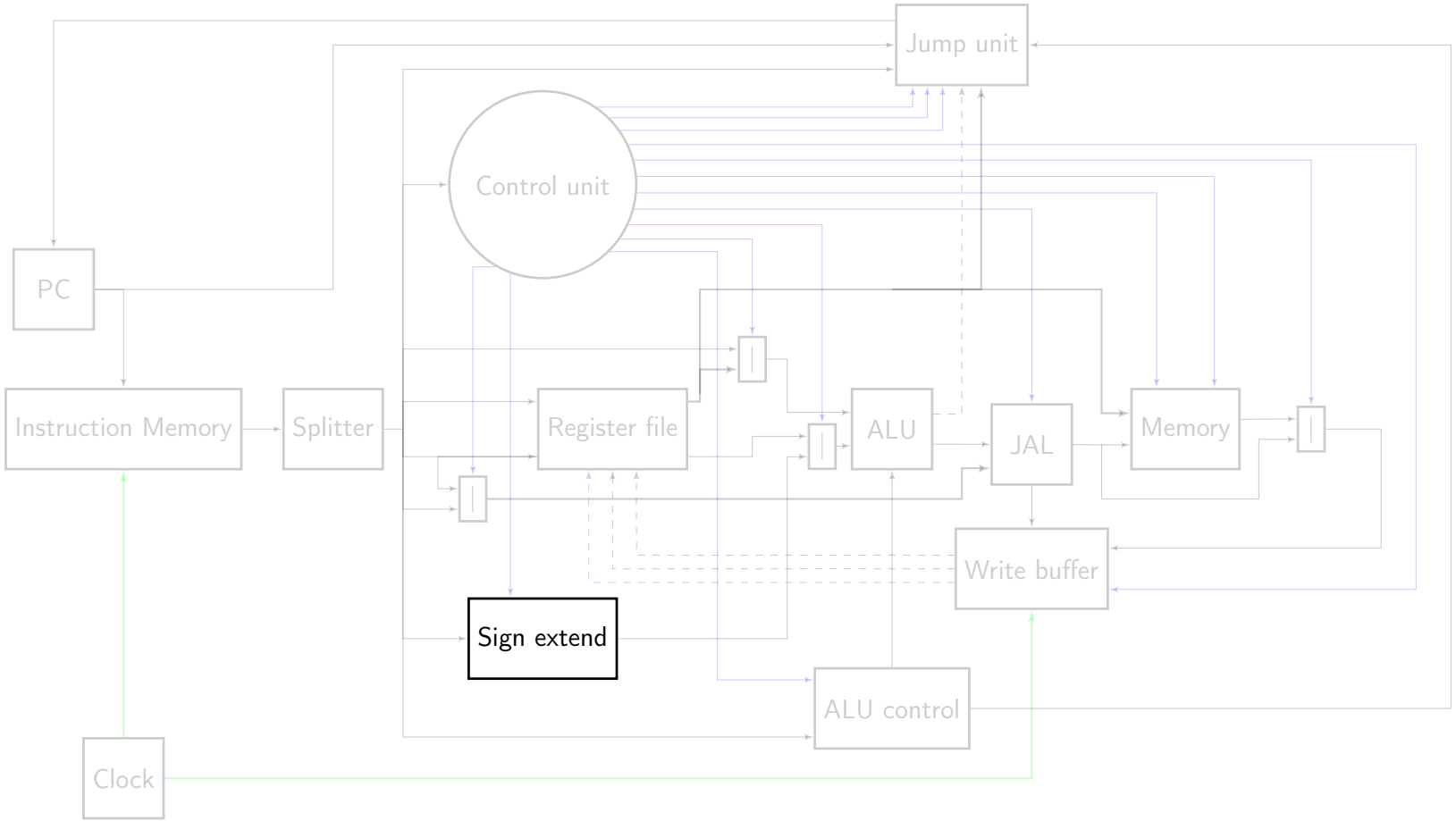
and it produces 6 outputs (bit numbers are inclusive):

- Opcode - bits 26-31
- Read Address A - bits 21-25
- Read Address B - bits 16-20
- Write Address - bits 11-15
- Immediate - bits 0-15
- Funct - bits 0-5


Instruction
Splitter
Opcode
Read Address A
Read Address B
Write Address
Immediate
Funct

Implementing the Splitter in SME is very simple. We just take the instruction, and use C# bit hacking to extract the bits that we need, and send the result out on the corresponding busses.

As always, testing is done by constructing a tester process, which sends input to the Splitter, and which verifies the output from the Splitter.

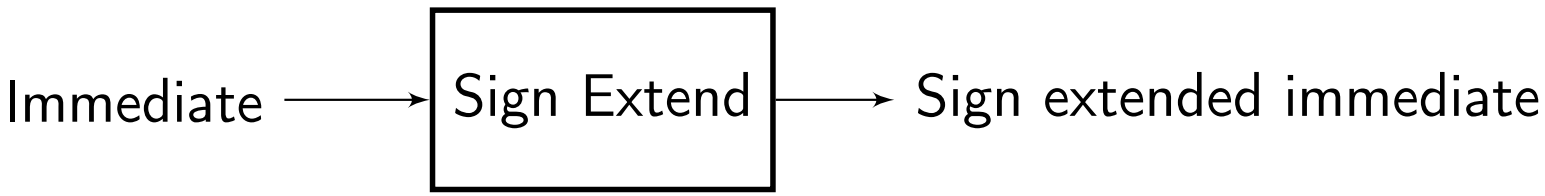

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The Sign Extend is used for extracting values from the instruction. It takes its input, which is 16-bit, and converts it into a 32-bit value, extending the sign if present. It takes one 16-bit input:

- Immediate

and produces one 32-bit output:

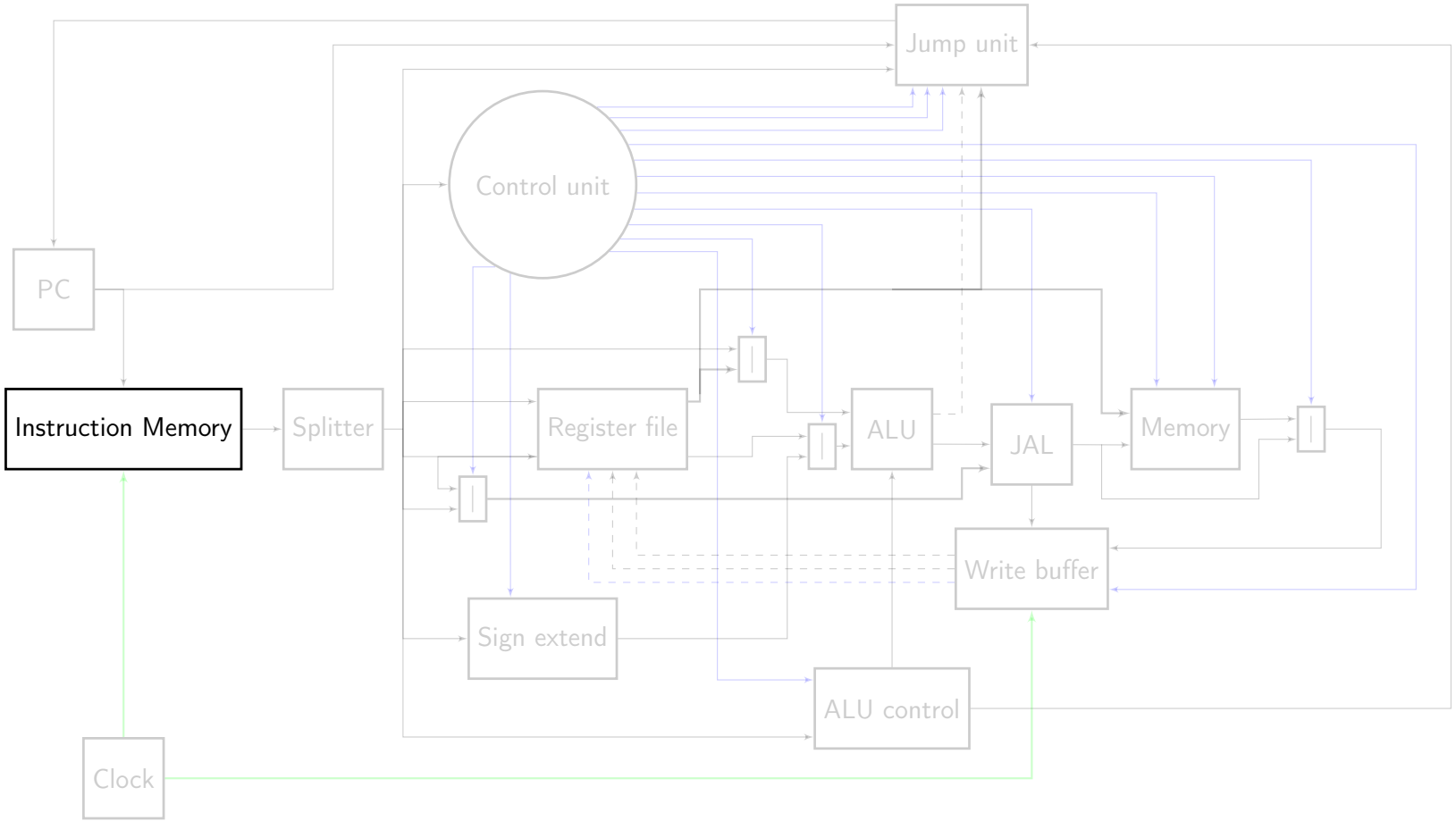
- Sign extended immediate


Immediate
Sign Extend
Sign extended immediate

The SME process takes the 16-bit immediate as input. Upon receiving all of its input, it outputs the input on its 32-bit output bus. In this case, C# handles the extending for us.

The testing is analogous to the previous tests.

Note: It is an good idea to test both negative and positive numbers.

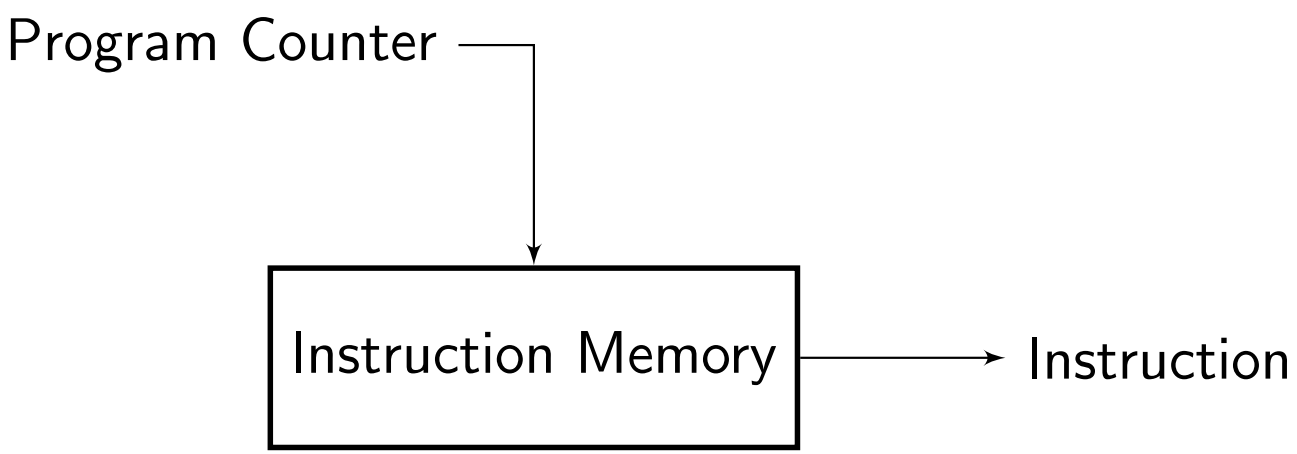

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The Instruction Memory is the part of the processor, which holds the program. It has a chunk of memory, and for each clock, it outputs the memory at the given address. It has one input:

- Program Counter

and it produces one output:

- Instruction

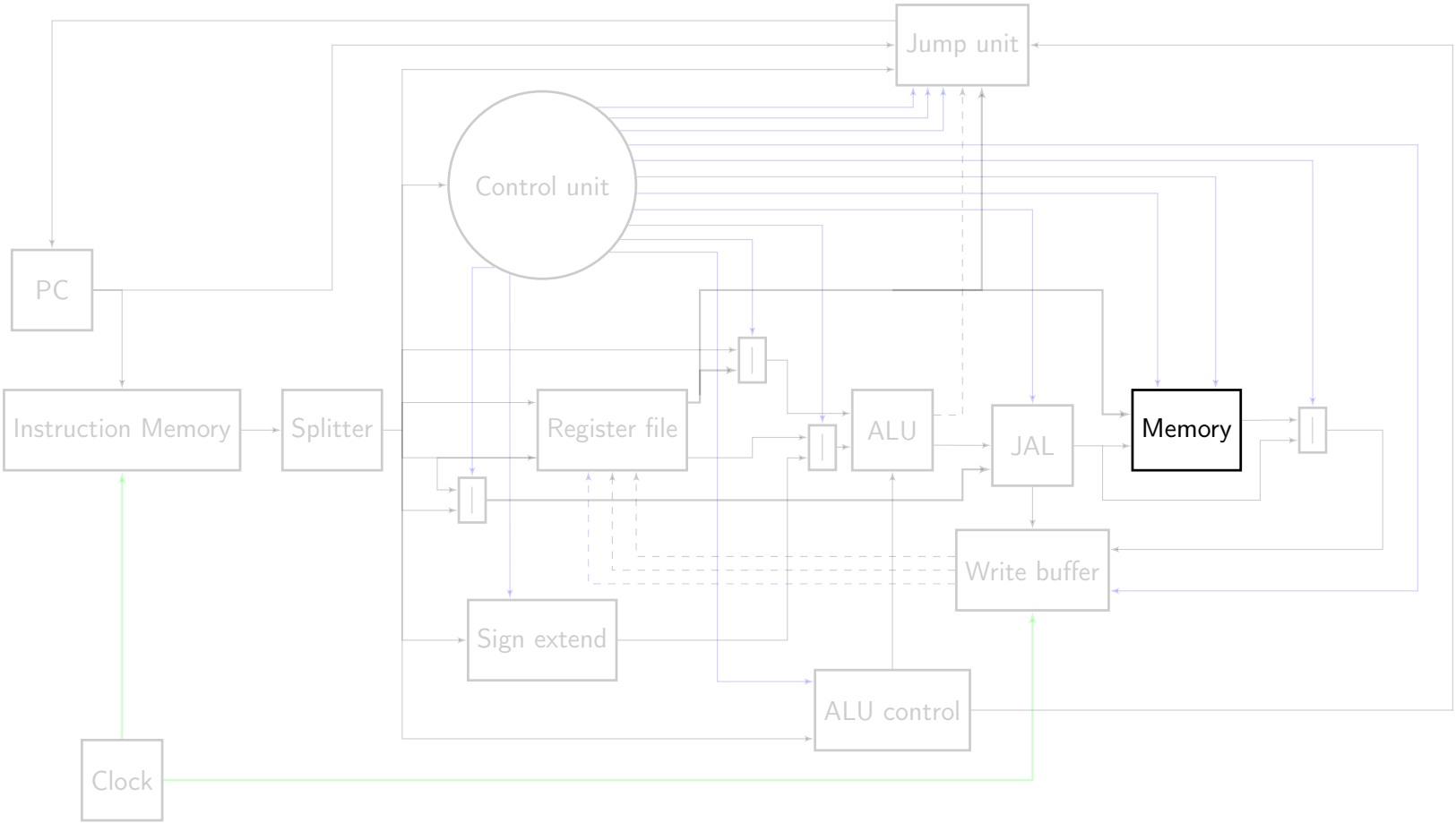

Program Counter
Instruction Memory
Instruction

Upon receiving all of its inputs, the Instruction Memory reads the address from the Program Counter bus, and outputs the value within its memory at the read address, on the instruction bus.

Usually memory should be a byte array. However, since we are working with C#, we can just use a `int` array for now, as then we dont have to worry about word alignment for now.

The Instruction Memory is tested by having a tester process, which inputs some reading addresses, and verifies whether or not they are as expected.

Note: since there is no way to input a program into the Instruction Memory, the program should be hardcoded for now.

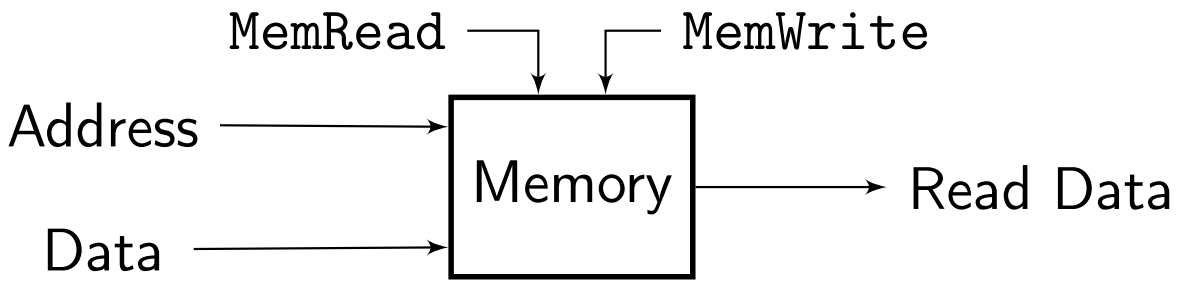

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The Memory Unit is the main memory. It can either be written to or read from. The values from and to the memory are word sized, i.e. in the 32-bit processor, the word size is 32. It takes four inputs:

- Address
- Data
- `MemRead`
- `MemWrite`

and produces one output:

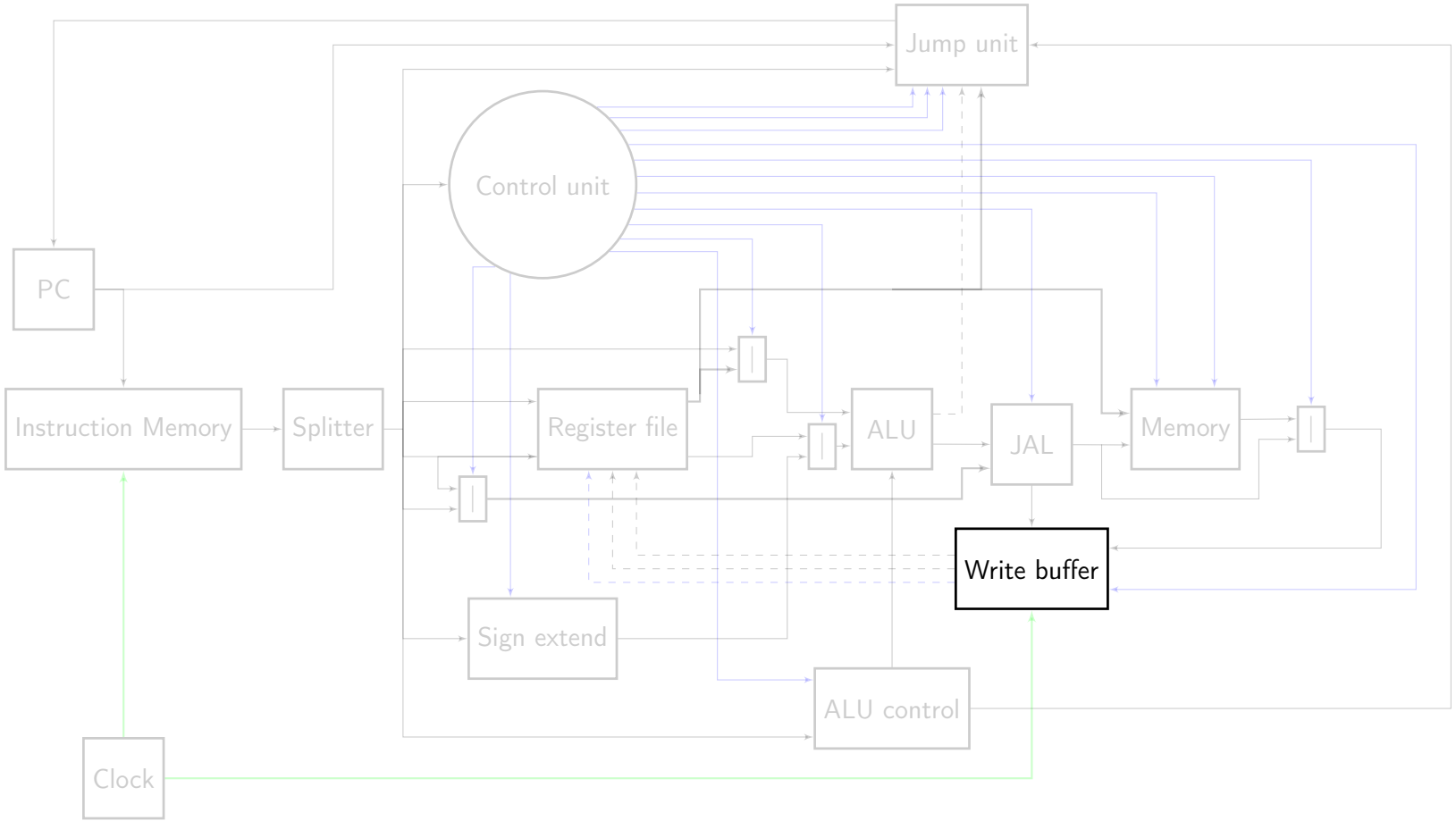
- Read Data


MemRead
MemWrite
Address
Memory
Read Data
Data

The memory should be implemented as a `byte` array, as this makes it easier later on when running programs on the processor. It should pack 4 bytes into an `int` value.

Upon receiving all of its inputs, the SME process should first check if it should read, in which case it should just output the value stored at the address of the Address bus. Then it needs to check whether or not it should write, in which case it should write the value from the Data bus into the memory at the given address.

The Memory is tested in the same manner as the Register File.

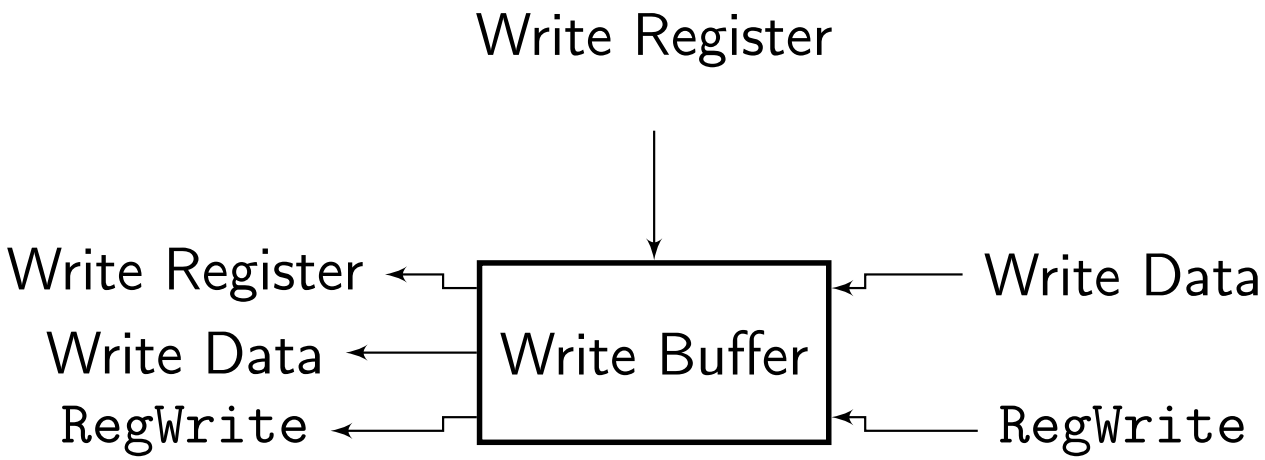

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The final stage of the processor is the Write Back. There is usually nothing special in this stage in the Single Cycle MIPS processor. However, we are not allowed to make unclocked cycles in SME, and there is an unclocked cycle from the Register File, through the ALU and Memory and back to the Register File.

To solve this, we introduce a Write Buffer. It is a clocked process, which holds all the values that the Register File needs for writing. Since it is clocked, we have removed the cycle. It takes three inputs:

- `RegWrite`
- Write Register
- Write Data

and produces the exact same as output.


Write Register
Write Register
Write Data
RegWrite
Write Buffer
Write Data
RegWrite

Implementing the Write Buffer in SME is straightforward. Construct a process that holds 3 values, and on each clock, output the values in the hold, and store the values from the input.

This is tested in the same matter as the previous components.