Core Components

Carl-Johannes Johnsen

Department of Computer Science
University of Copenhagen

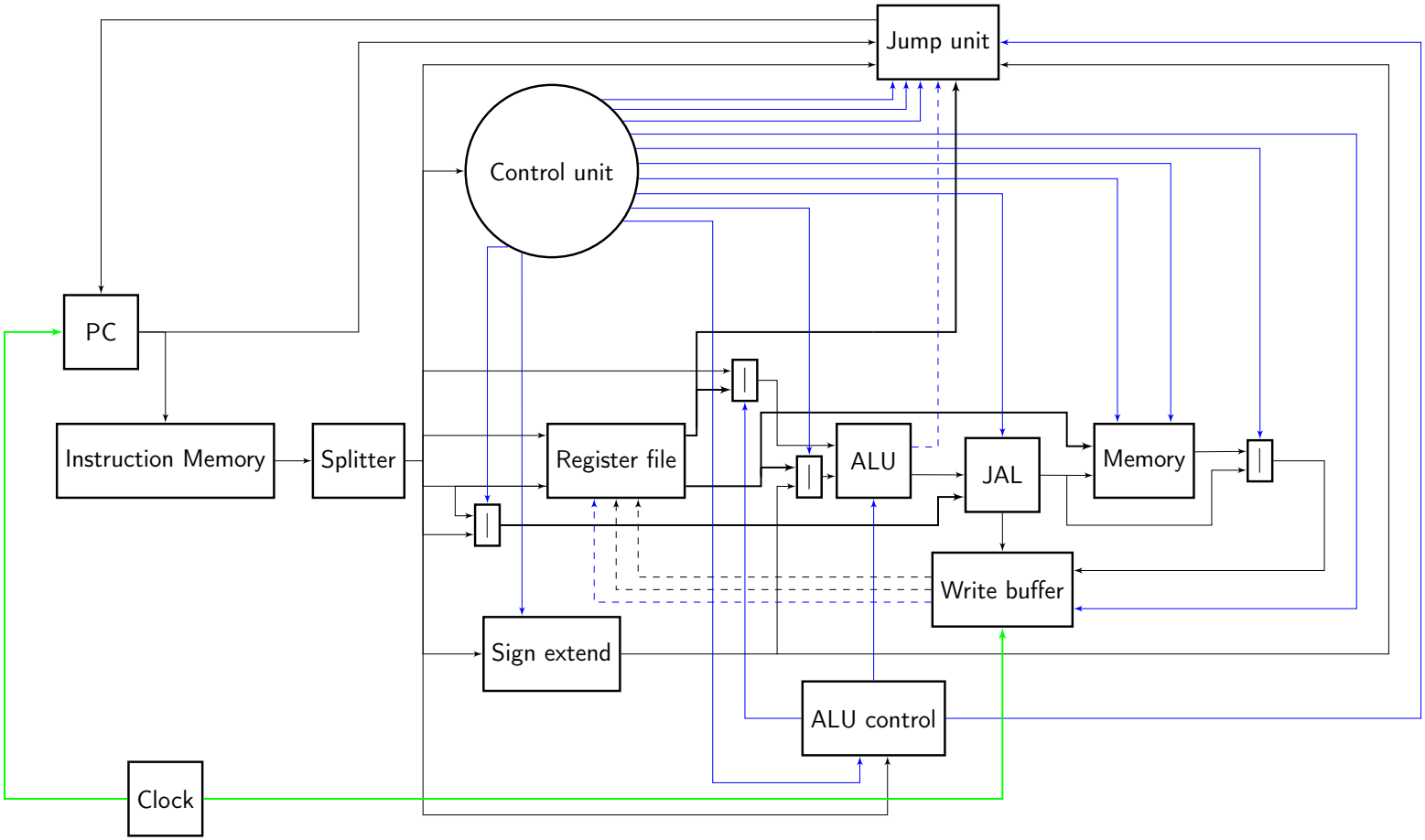
March 19, 2017

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

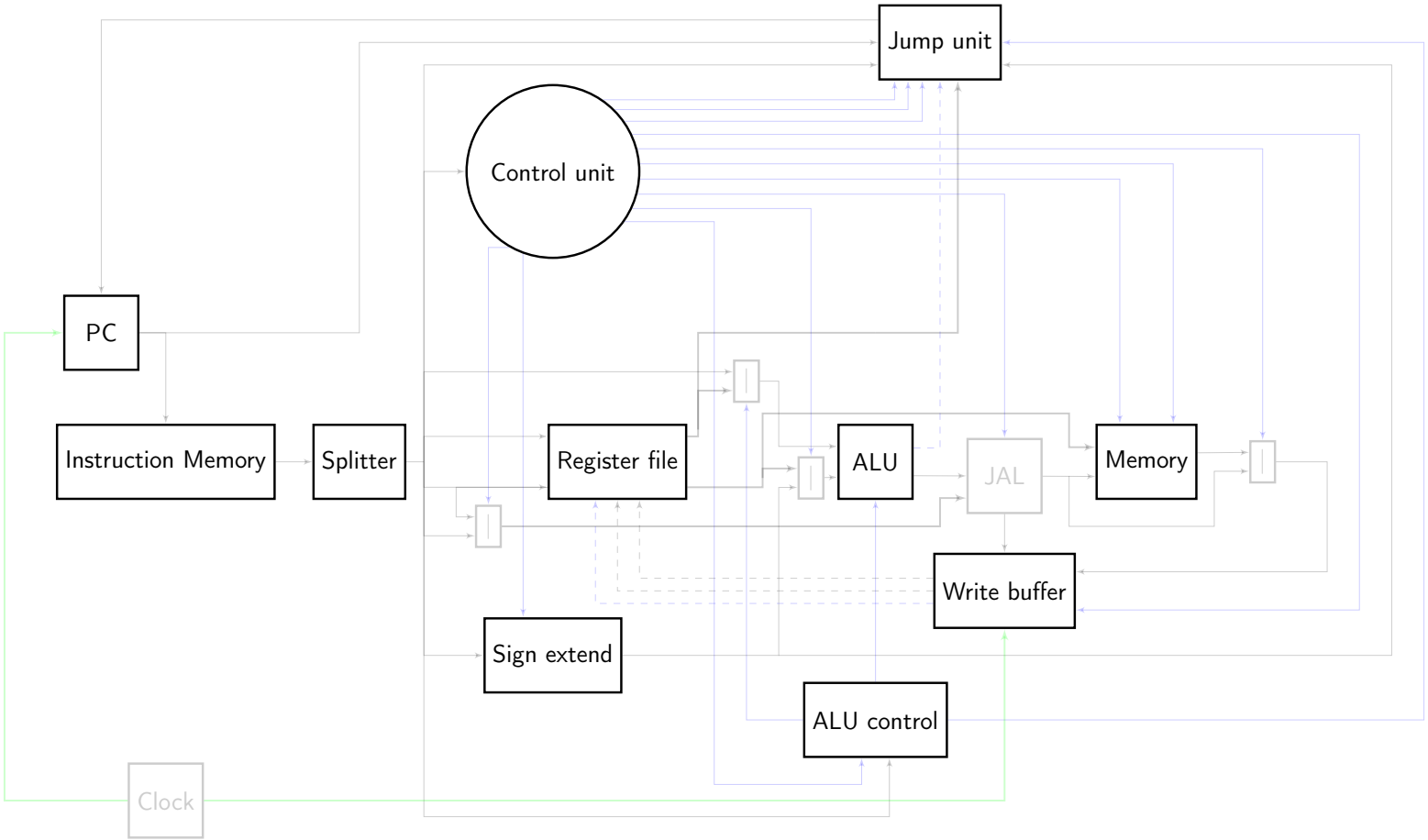
By combining these components, we should be able to construct a single cycle MIPS processor.

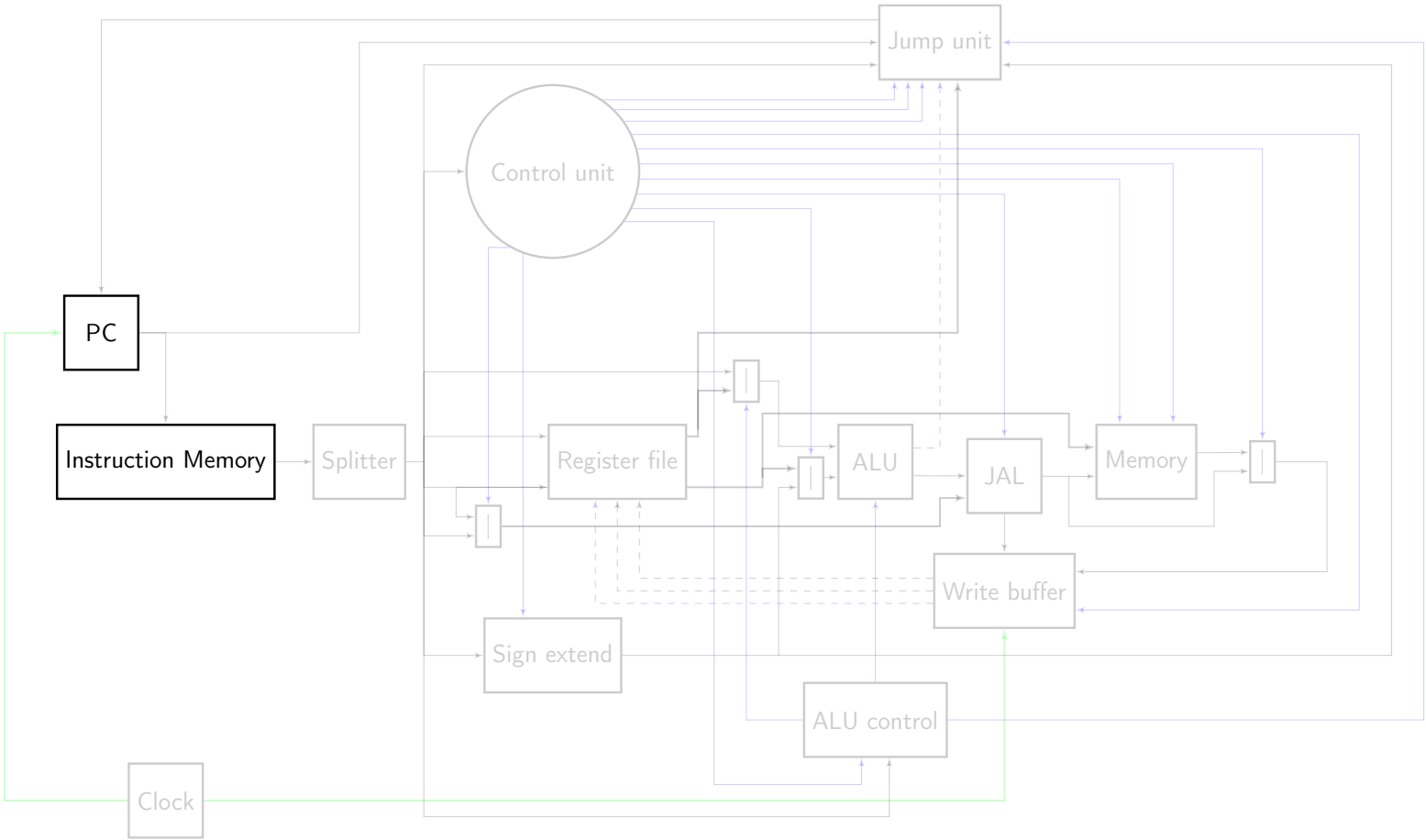
When we talk about testing, we should test in the samme manner as with the logic circuits, i.e. construct a tester process, which sends inputs to the components, and verifies that the output is as expected.

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

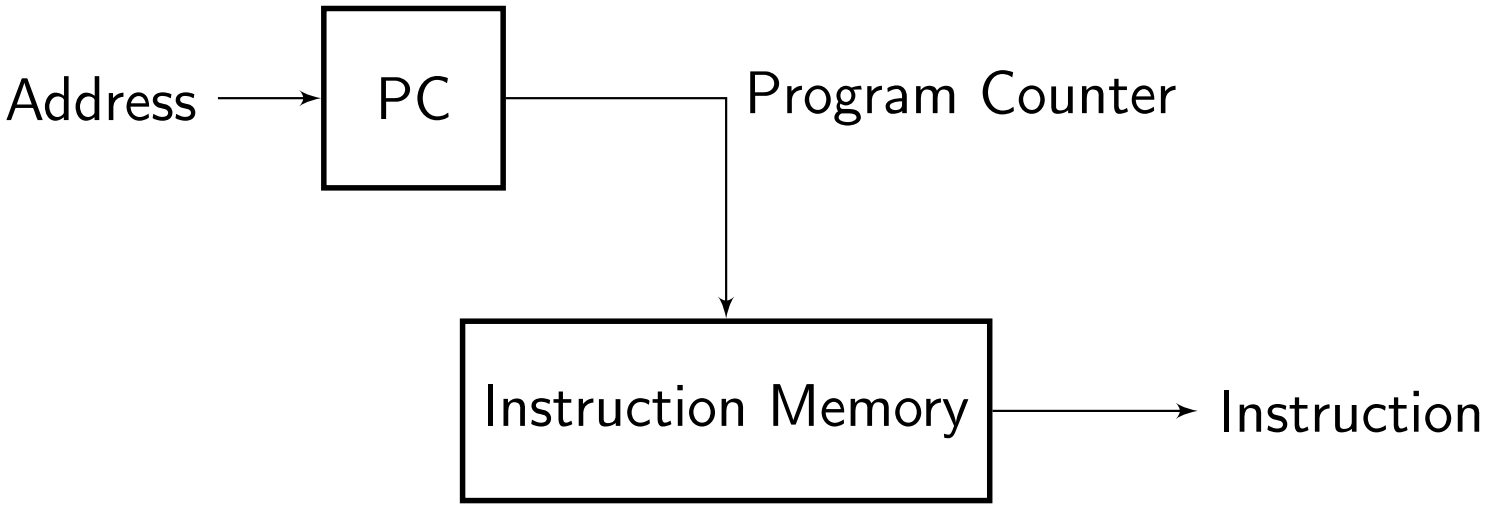
- Program Counter

and it produces one output:

- Instruction

We are also going to need a register to hold the current address, called the Program Counter (PC). This register outputs to the Instruction Memory, and has a single input:

- Address

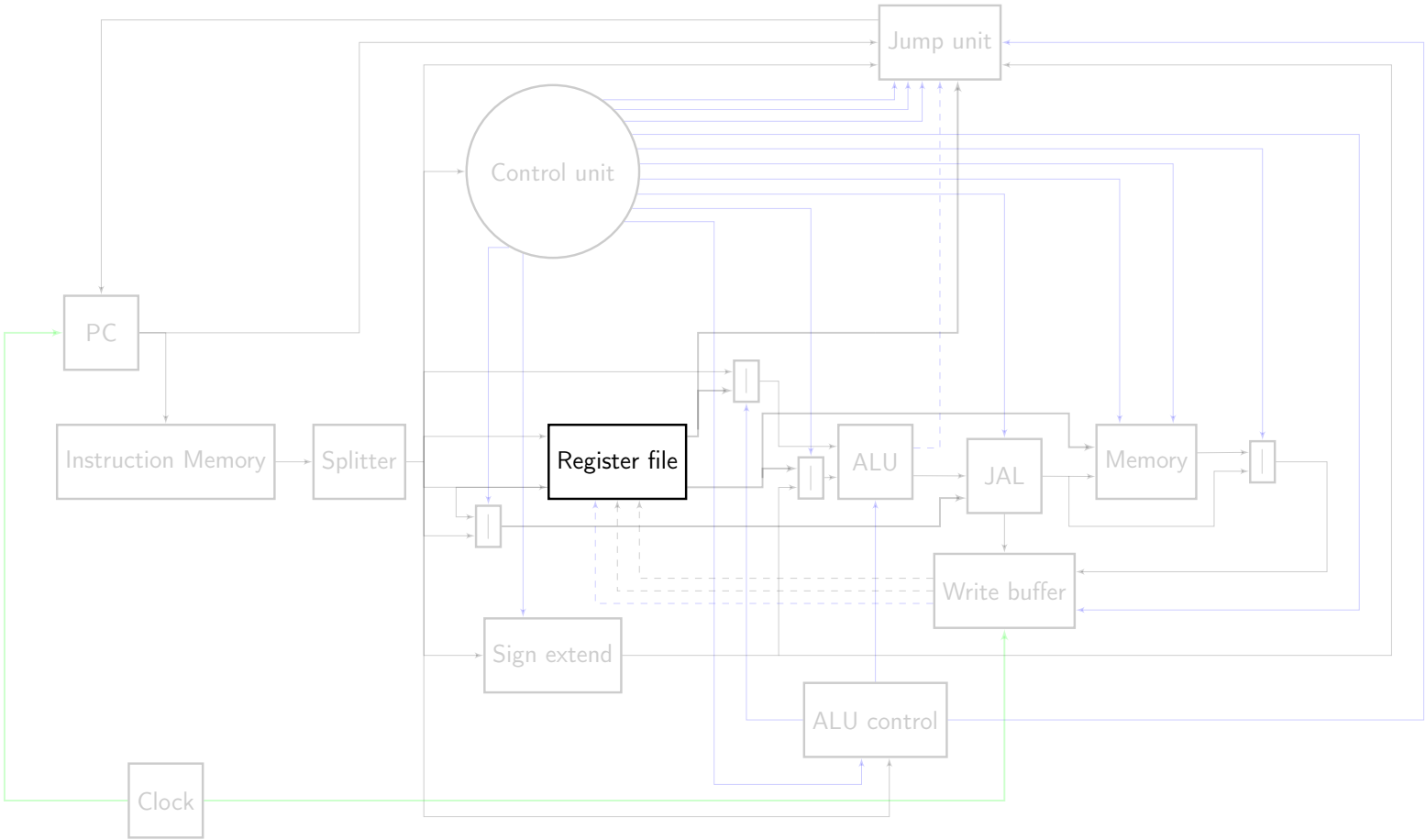

Address
PC
Program Counter
Instruction Memory
Instruction

Both the PC and the Instruction Memory should be SME processes. The input bus for the PC, the Program Counter bus and the instruction bus, should all contain an `uint` value.

The PC should have a single `uint` value holding the current address. On each clock tick, it should output its stored value, and store the new input value. It should also be clocked, as this is the starting point of every instruction.

The Instruction Memory should have an `byte` array, as this will make indexing from the address simpler. Usually the Instruction Memory and the Memory Unitshare the same address space. However, they each have their own for simplicity. On each clock tick, the Instruction Memory should read the byte at the given address, along with the following three bytes. Then it should take the four bytes, and pack them together into an `uint`, and put it on the instruction bus.

Testing the Instruction Memory is very trivial. The program should be hardcoded into the `byte` array. The tester process should then send values to the PC, and verify that the output on the instruction bus, matches the value at the given address in the memory.


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.

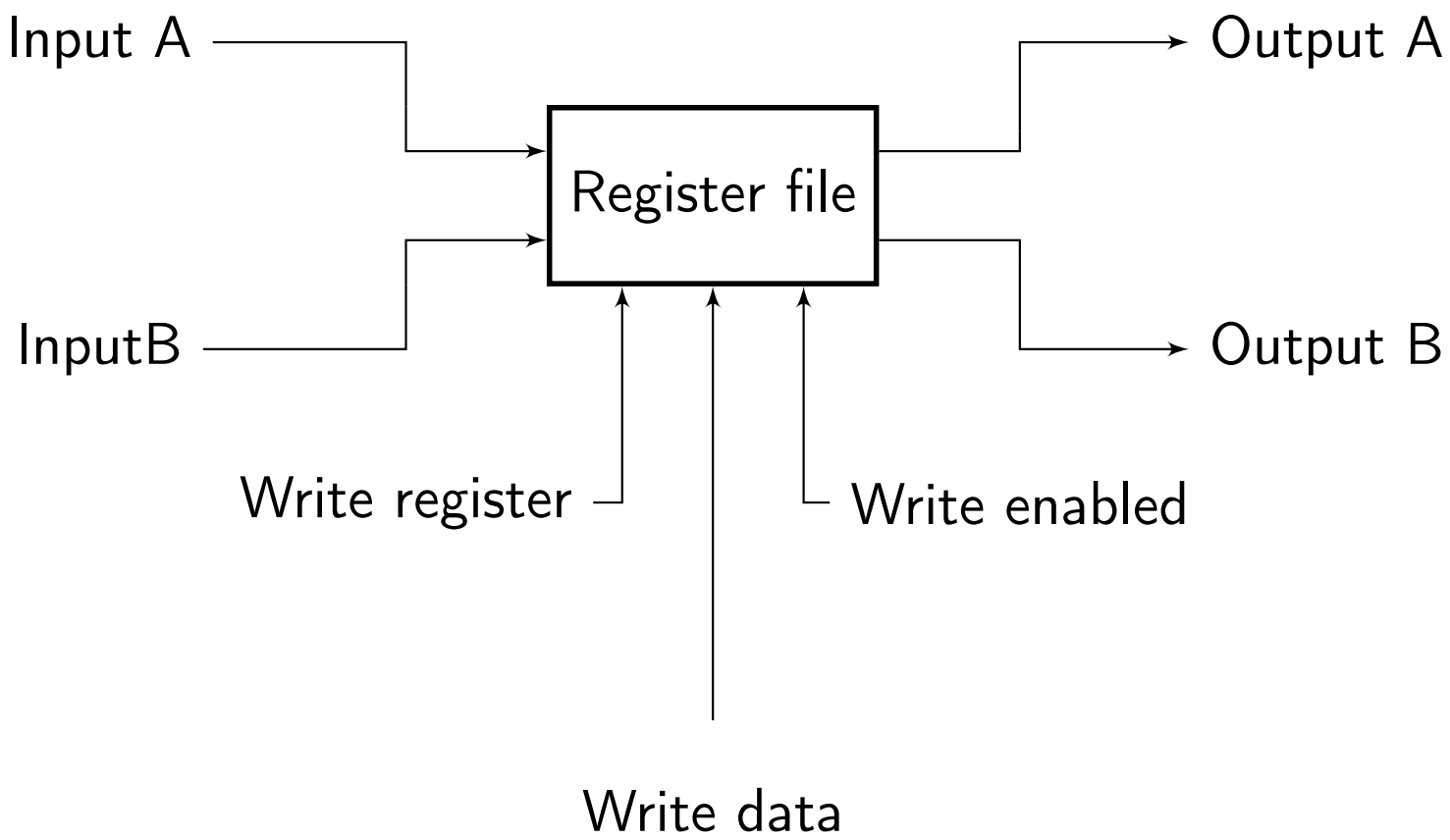
There are 32 registers in a 32-bit MIPS processor, each holding a 32-bit value. The registers are divided into groups based on their usage. This does not matter from a hardware perspective, except for register 0, which is immutable and always 0.

The register file has 5 inputs:

- Read address A
- Read address B
- Write enabled (`RegWrite`)
- 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 `uint` array. It should have the inputs and outputs as specified. The address busses should all hold a `byte` value. The output busses and the Write Data bus should all hold an `uint` value. And the Write Enabled bus should hold a `bool` value.

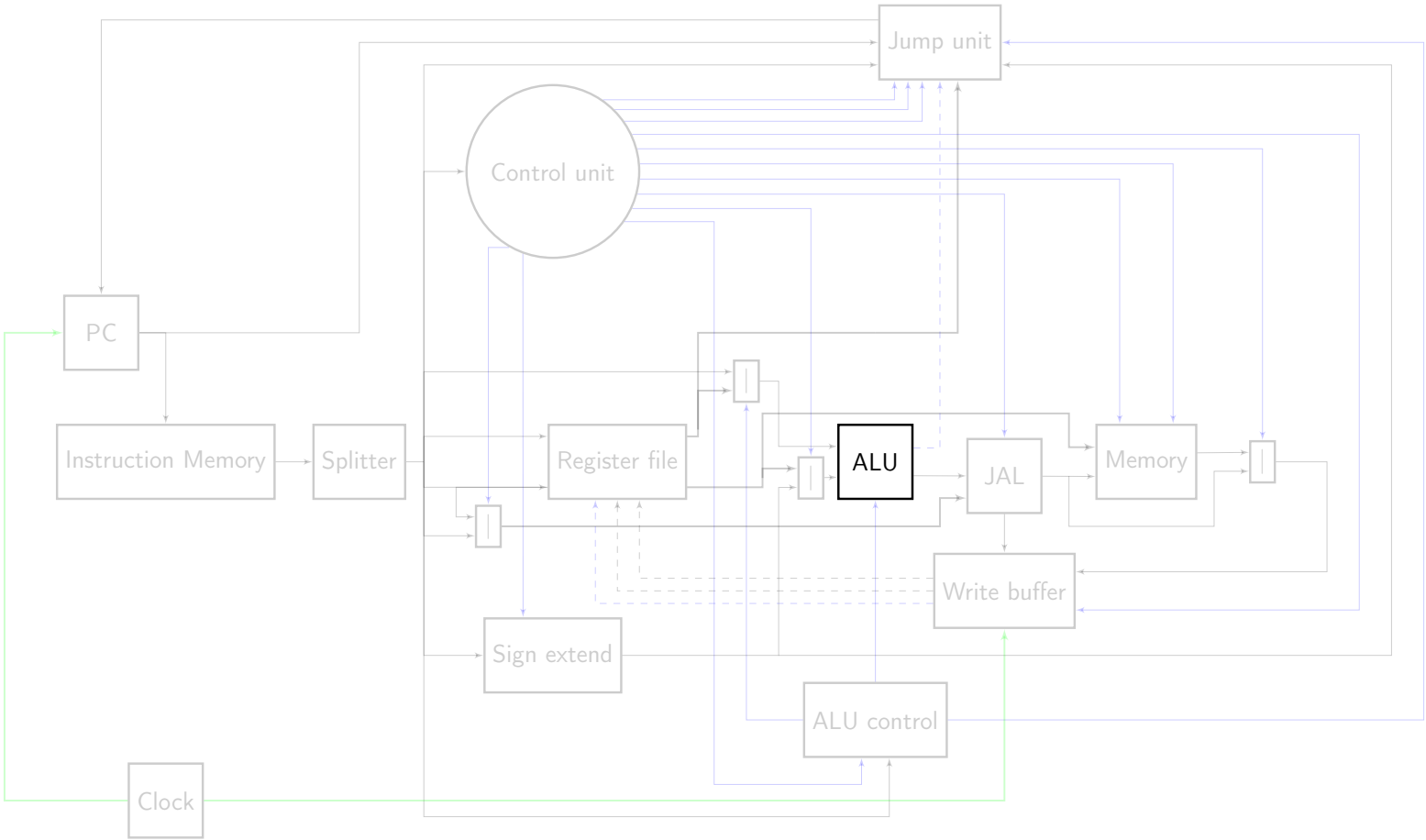
Upon receiving input, it should write the write data into the array at the given write address, if the `RegWrite` flag has been set. Then it should output the value at Read Address A onto Output A, and the analogous for Read Address B and Autput B.

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

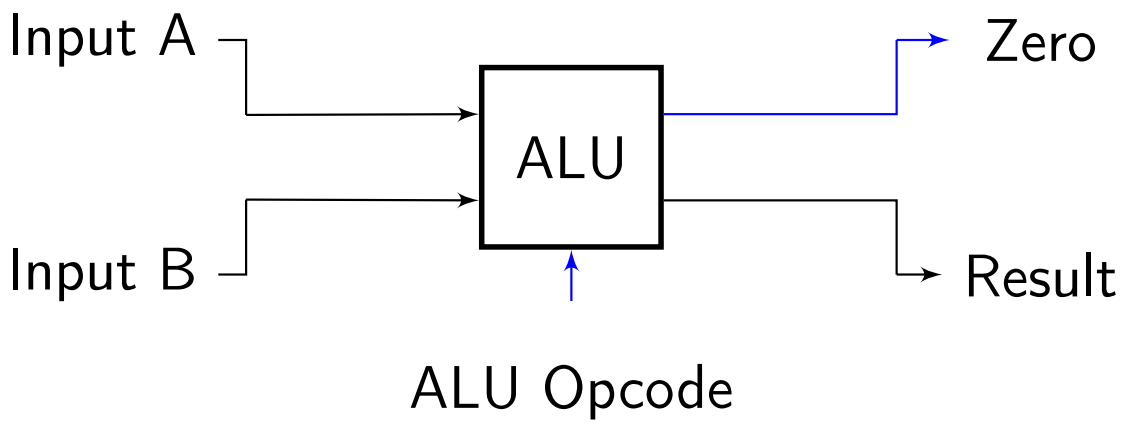
The ALU (Arithmetic Logic Unit) is the part of the processor, which makes the actual computation.

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.

To implement the ALU, we construct a process in SME. It should have the inputs and outputs as specified. The input busses and the result bus should hold an `uint` value. The ALU Opcode should hold a `byte` value, and the Zero bus should hold a `bool`.

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`

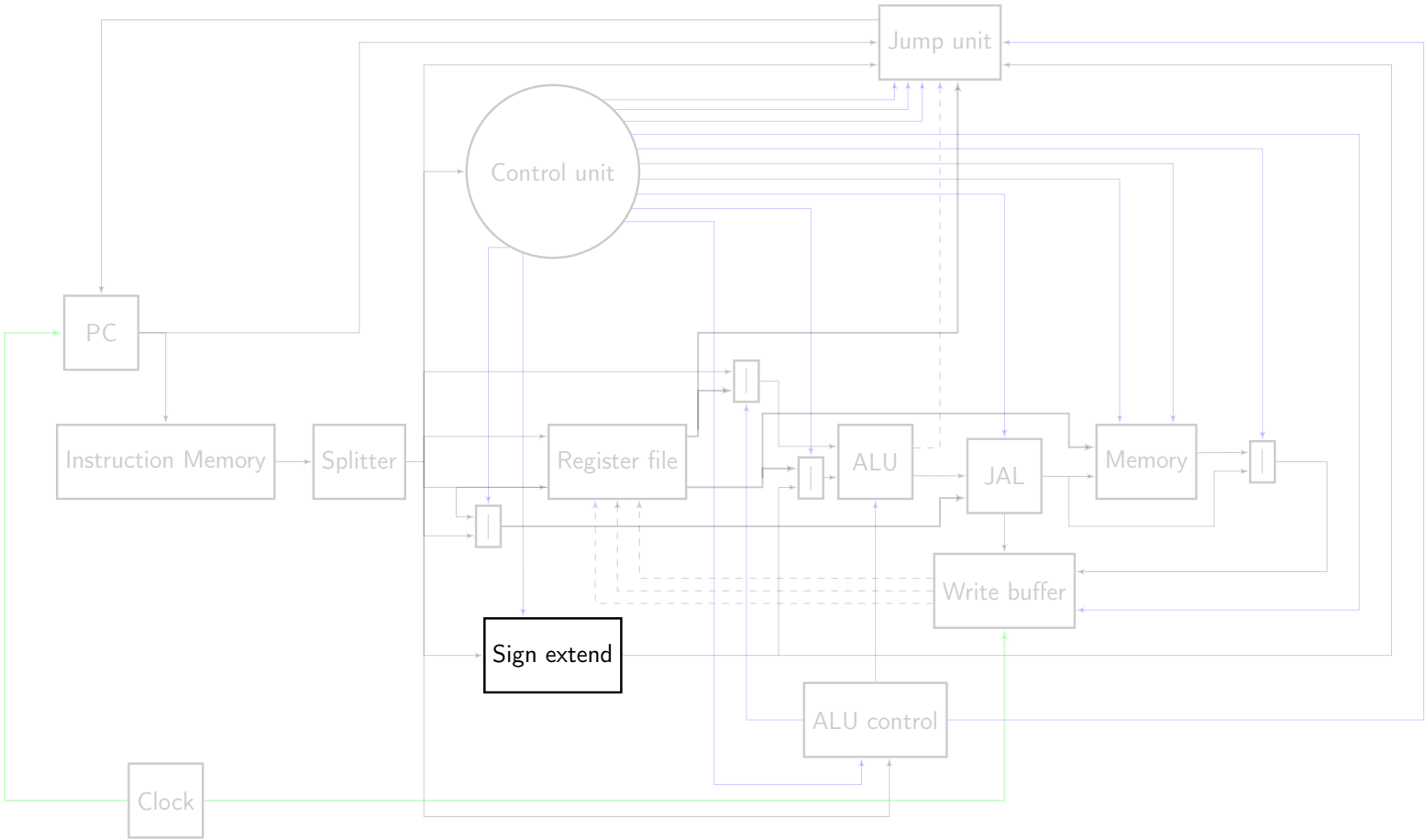
We follow the procedure in the book, and make sure the ALU support the following operations:

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

We will be adding more operations later on.

Testing the ALU is like the Register File, very trivial. Construct a tester process, which:

- Sends values on the Input A, Input B and ALU Opcode busses
- Reads the values from the Result and Zero Busses
- Verifies that the read values are as expected

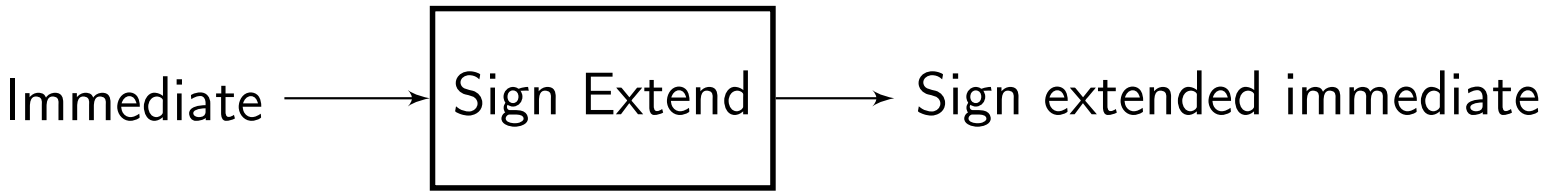

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The Sign Extend is used for extracting the 16-bit value from the instruction, called the Immediate. 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:

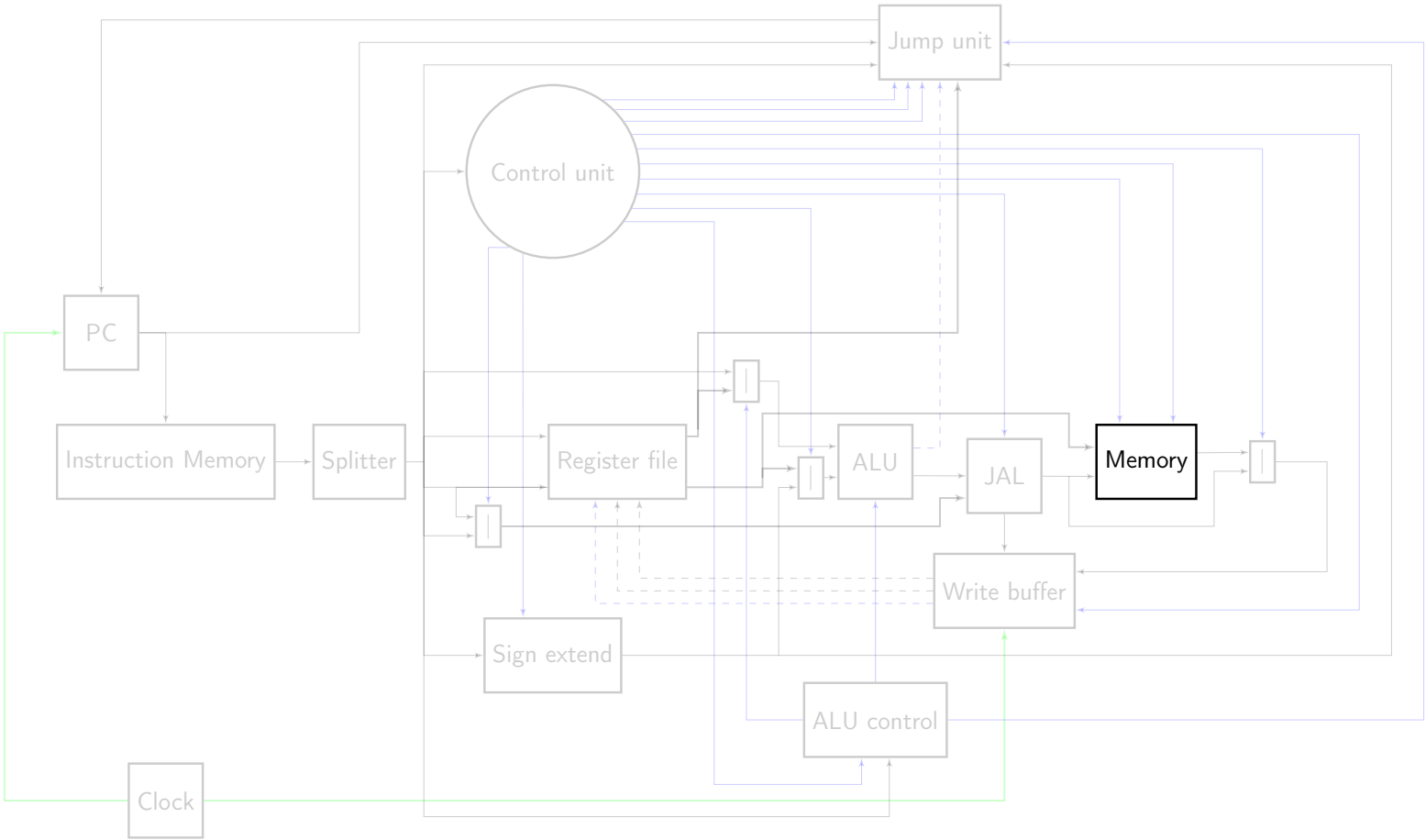
- Sign extended immediate


Immediate
Sign Extend
Sign extended immediate

To implement the Sign Extend, we construct an SME process. The input bus should contain a `short` value, and the output bus should contain a `uint` value.

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.

Testing the Sign Extend is very trivial: just check that the value on the output bus is the same as the one sent on the input bus. 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

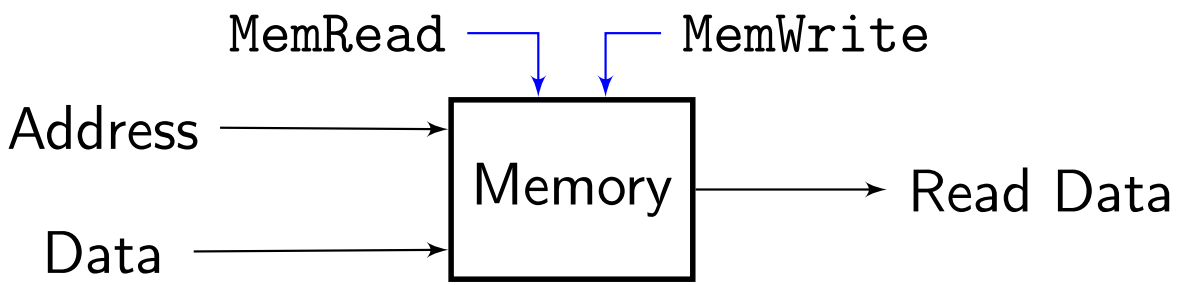
The Memory Unit is the main memory of the processor. It can either be written to or read from. The addresses for the memory are byte addresses and word aligned, i.e. in the 32-bit processor, the word size is 32 and the addresses should as such be dividable by 4.

It takes four inputs:

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

and produces one output:

- Read Data


MemRead
MemWrite
Address
Data
Memory
Read Data

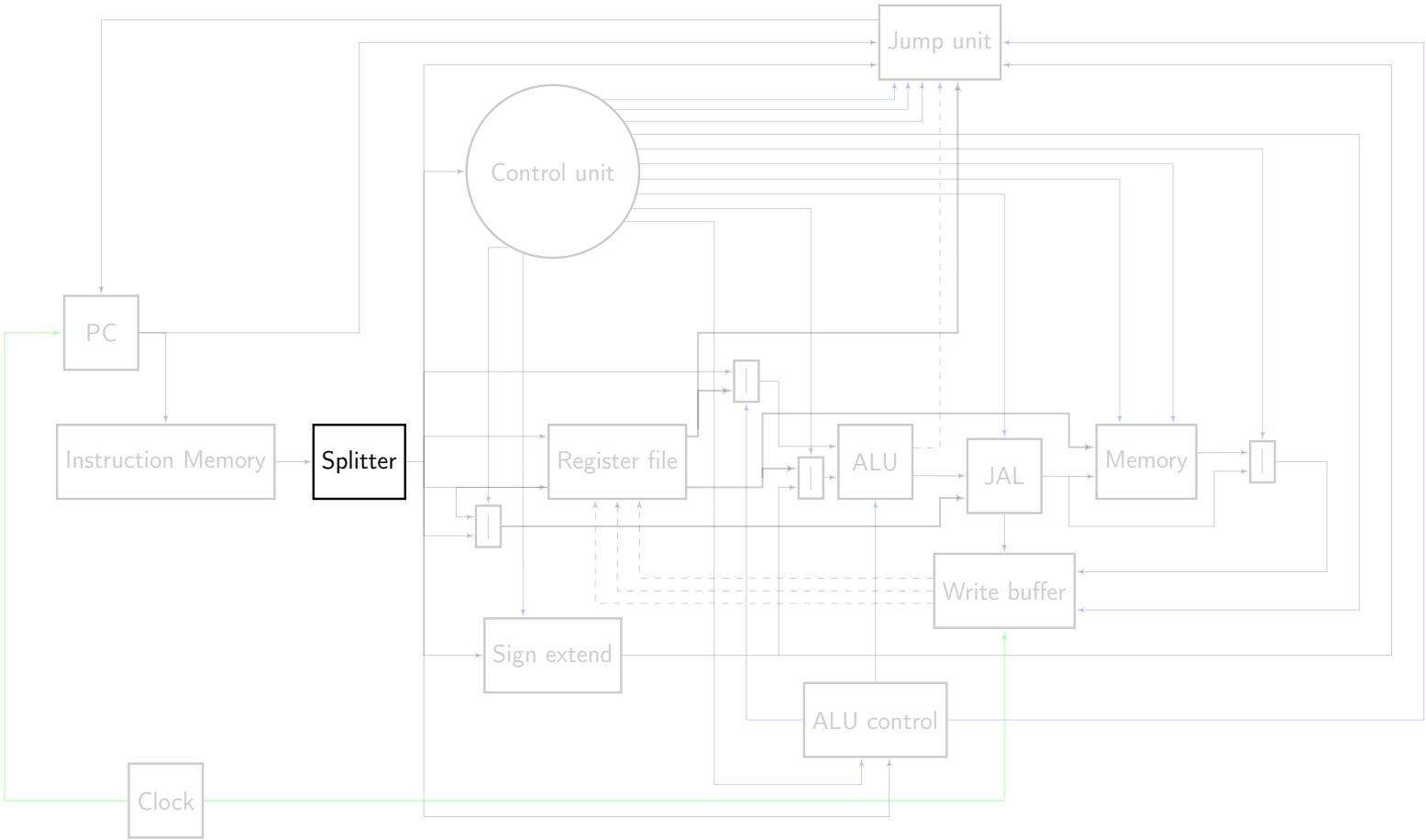
To implement the Memory Unit, we should construct an SME process. The Address bus, the Data bus and the Output bus should all contain an `uint` value. The two control busses, `MemRead` and `MemWrite`, should both contain a `bool` value.

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

Otherwise, 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, i.e. by inserting some values into memory, and checking if the same values can be fetched again.

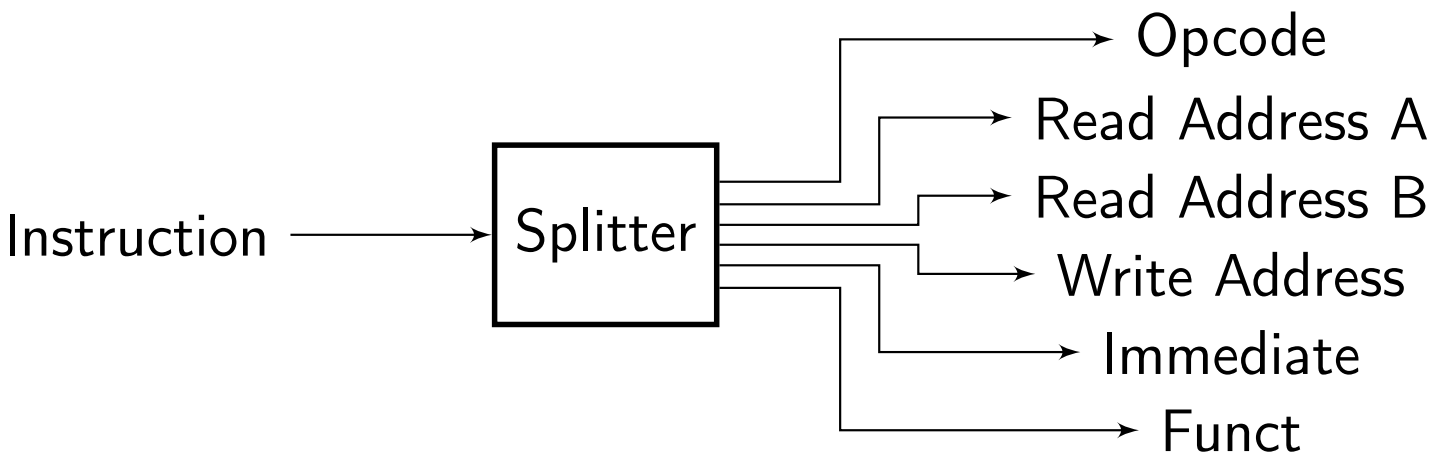

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

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

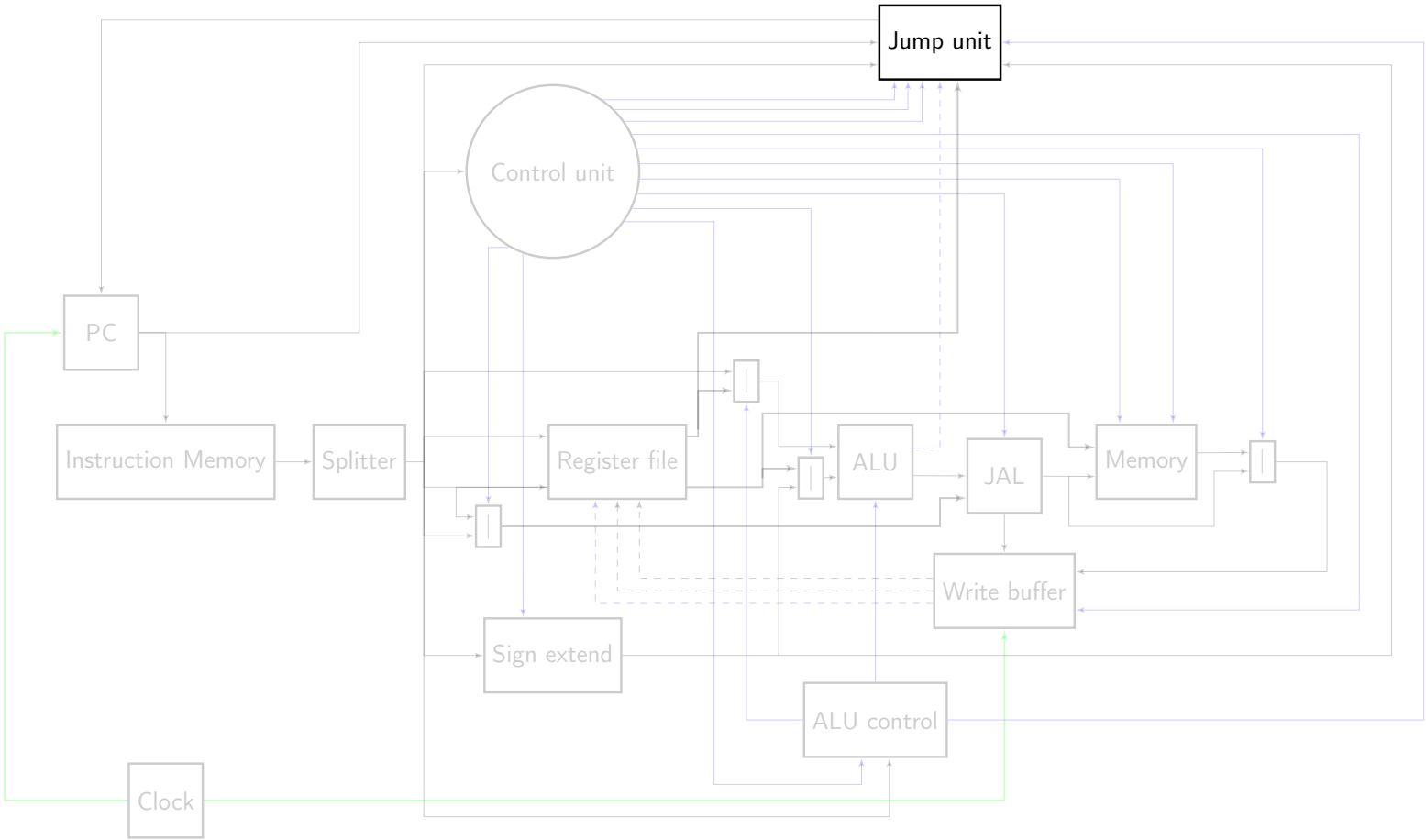
and it produces 6 outputs (bit numbers are inclusive):

- 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 contsruct an SWE process, which takes the instruction, and use C# bit hacking to extract the bits that we need, and send the result out on the corresponding busses.

Testing the Splitter requires no special procedures.

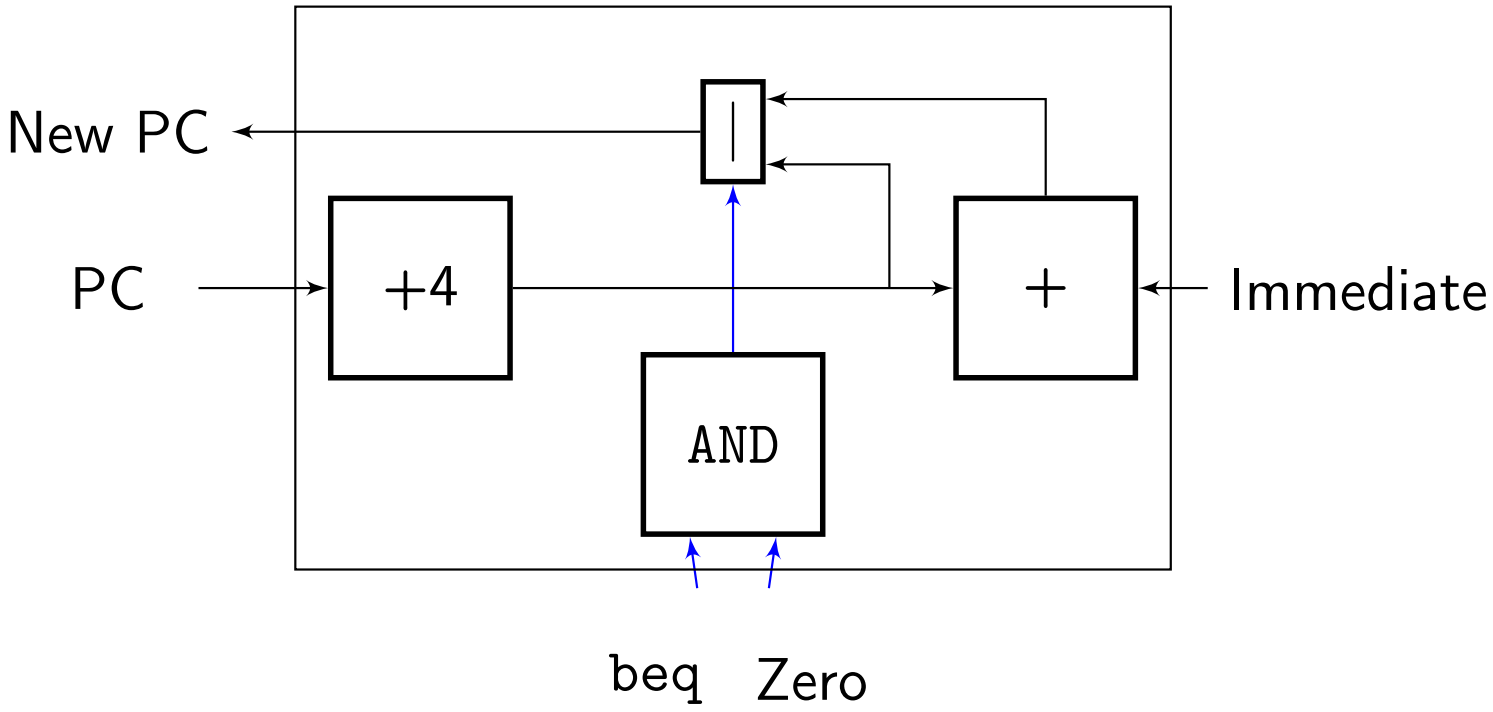

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The Jump Unit is the one controlling which instructions to load next. It takes four inputs:

- Sign extended value from the Sign Extend
- Zero signal from the ALU
- The beq signal
- The PC

and produces one output:

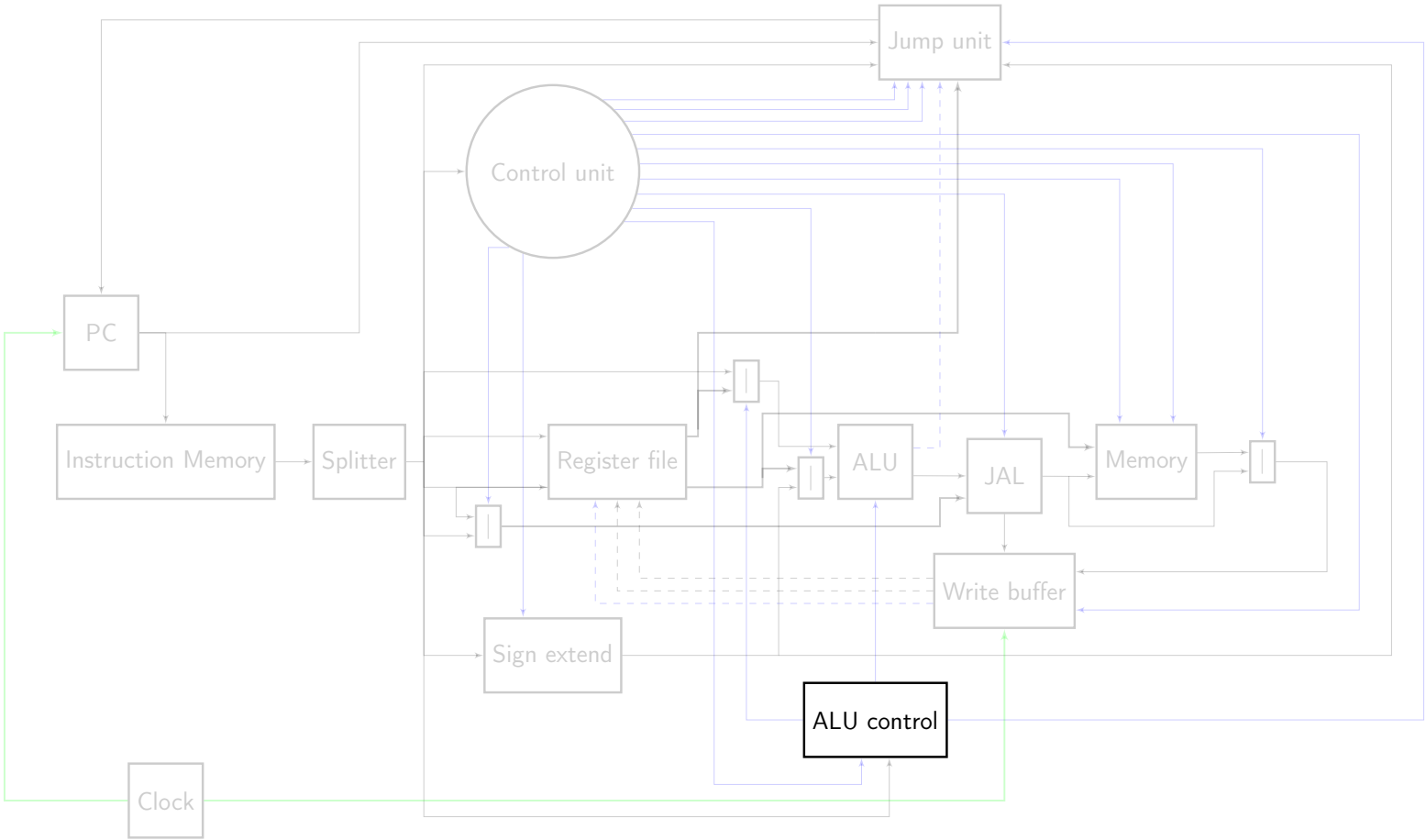
- The new PC


New PC
PC
+4
AND
+
Immediate
beq
Zero

The Jump Unit is responsible for handling branching and jumping instructions. In the initial single cycle MIPS processor, we are only aiming at implementing normal program traversal and the branching instruction `beq`.

As shown in the previous figure, it can be done with two adders, one `AND` gate, and a multiplexor.

Testing the Jump Unit should be straightforward.

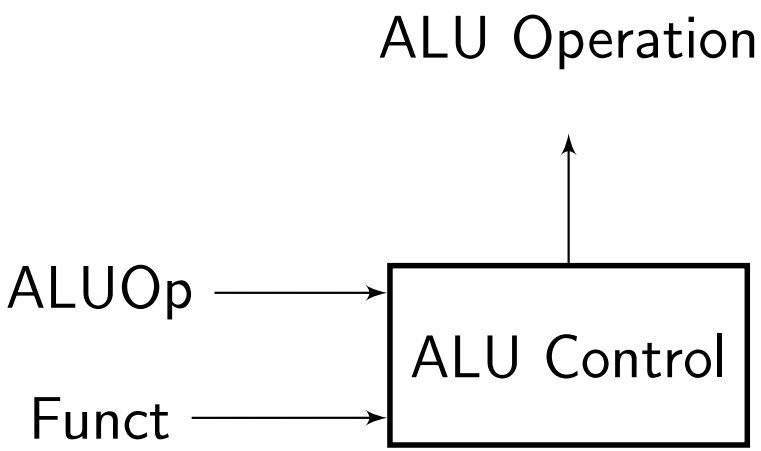

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

The ALU Control is used for generating the ALU Operation 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` code, to identify whether or not the instruction is in R format, in which case it should look at the `funct` code.

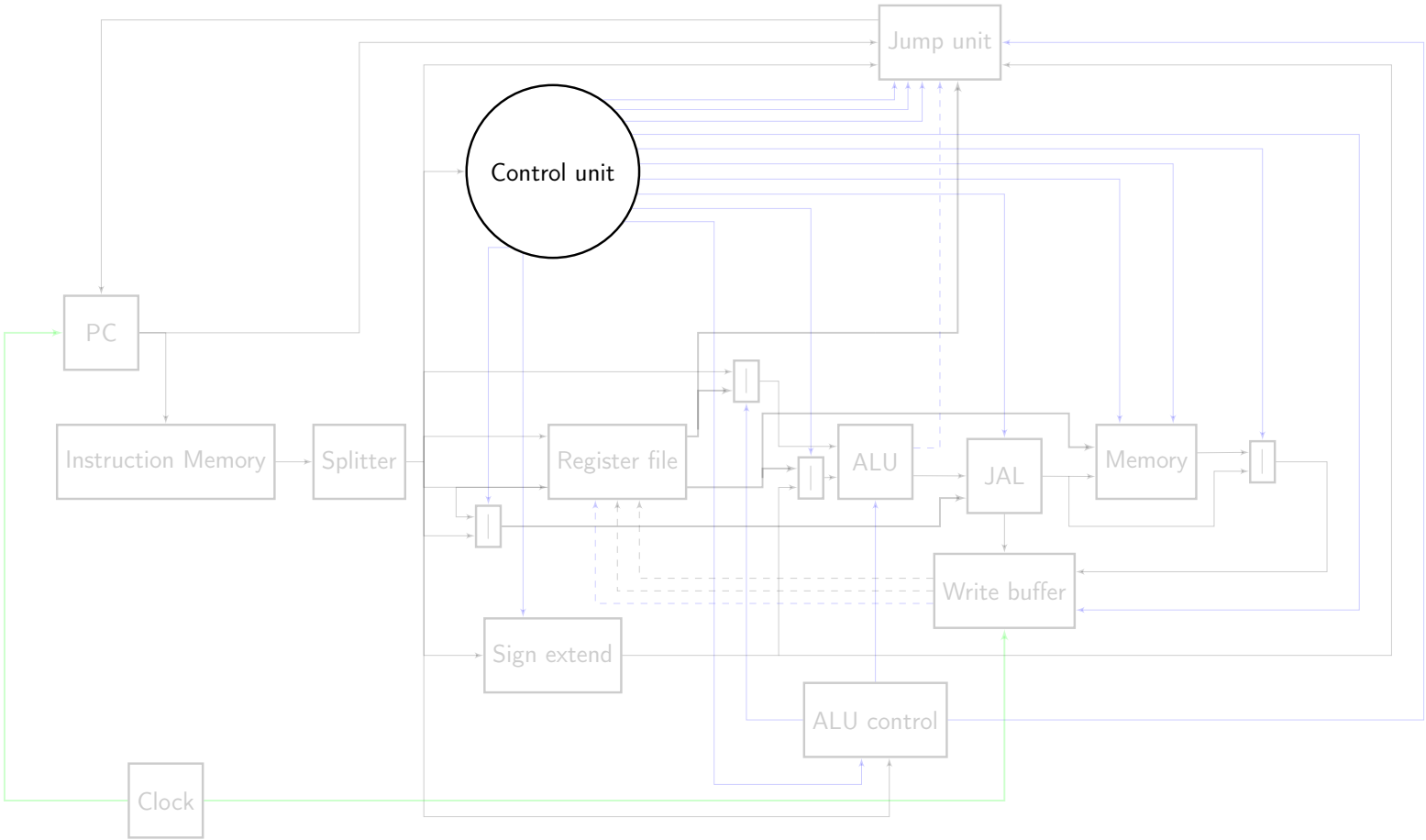
In either case, it outputs on the ALU Operation bus, which operation the ALU should perform.

We construct an SME process, which should start by looking at the `ALUOp` code, and have an if on whether or not it is in R-format.

If it is in R format, it should have a `switch` on the `funct` input. If not, it should have a `switch` on the `ALUOp` code. 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 ALU Control should be tested for all possible combinations of `ALUOp` and `funct` codes, as there should not be too many of either.

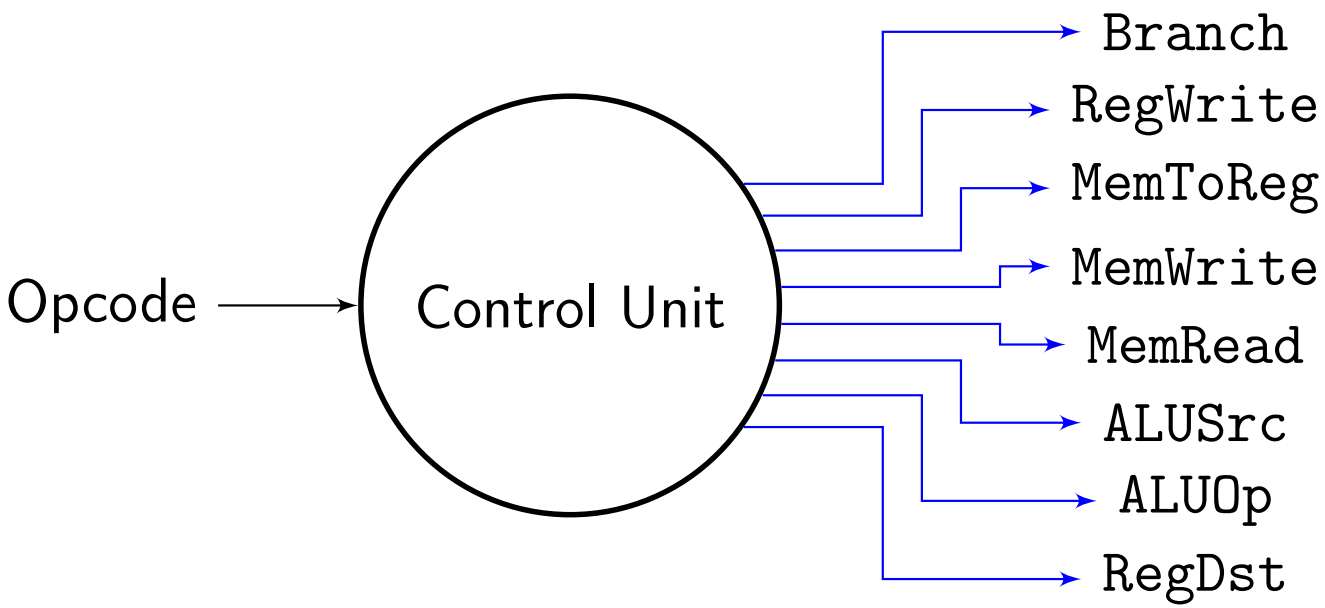

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

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.

To implement our first set of instructions, we are going to need the following flags:

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

Each of the control flags goes to their respective part of the processor.

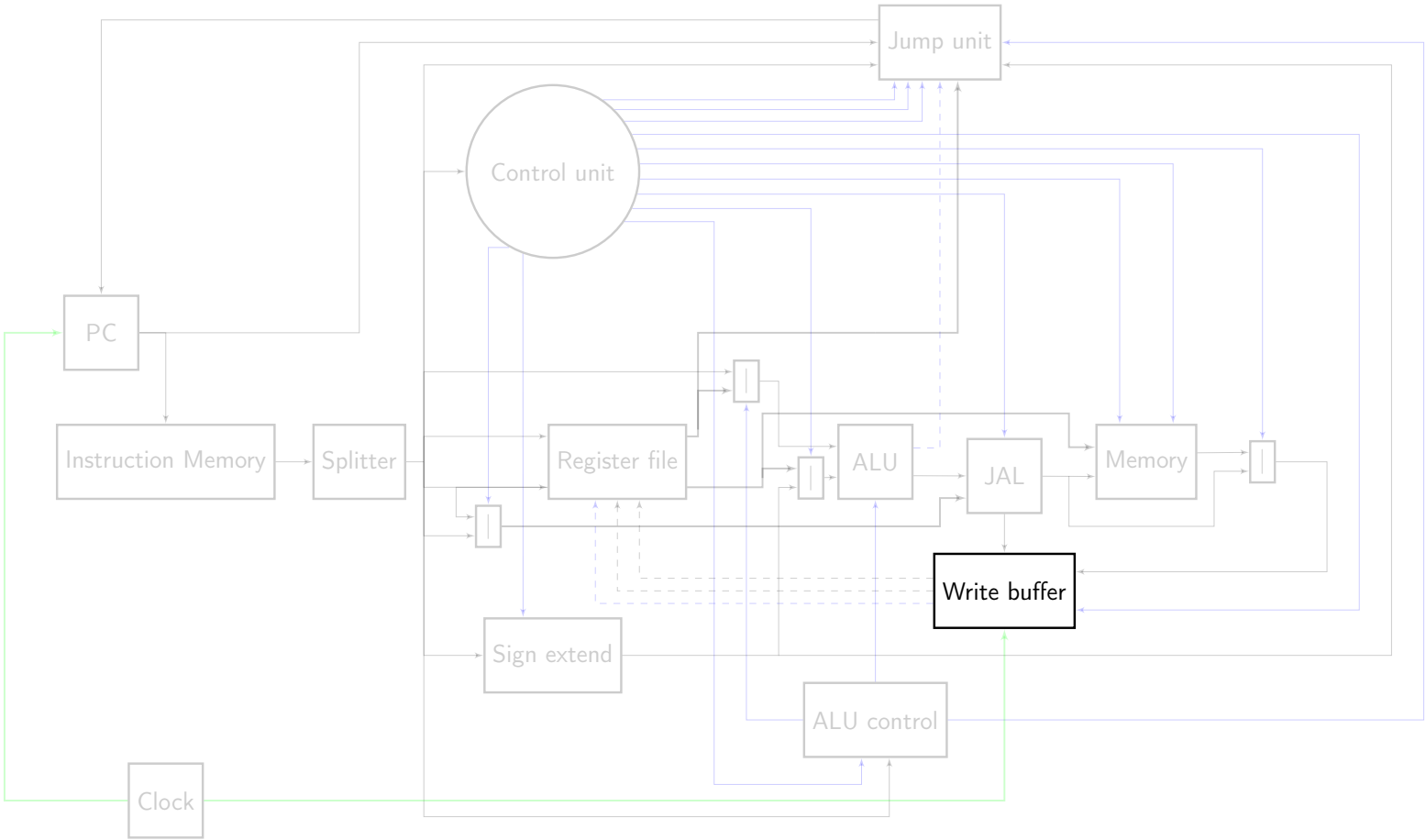

Opcode
Control Unit
Branch
RegWrite
MemToReg
MemWrite
MemRead
ALUSrc
ALUOp
RegDst

As with the ALU Control, the book describes the logic needed to implement the Control Unit. However, it is not trivial to extend, and has been generated.

As such, we construct our own Control Unit. It should have a `switch` on the input opcode, and in each case, 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.

Testing the Control Unit should try all possible inputs, as there should not be too many that the Control Unit can actually handle.

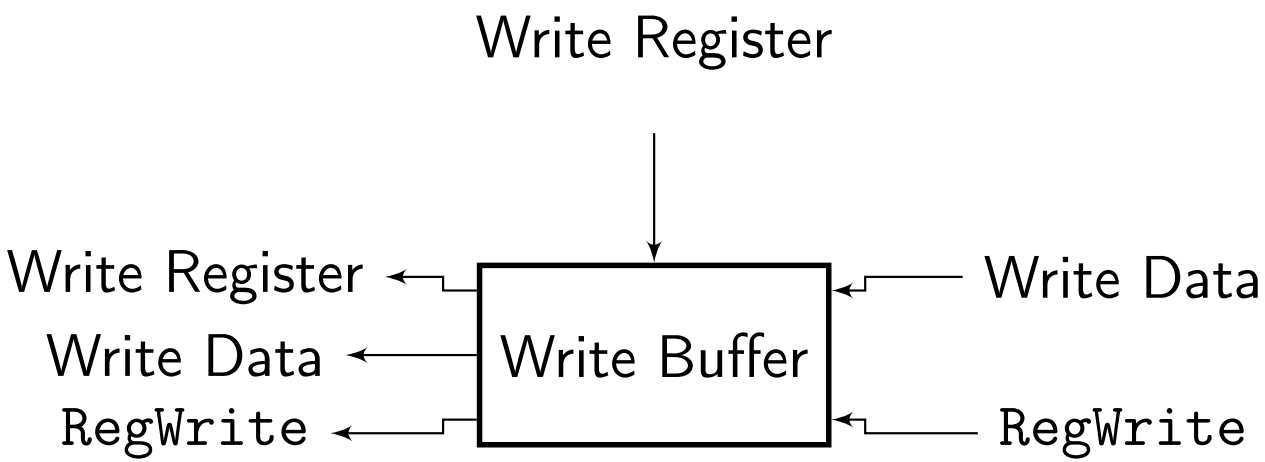

Jump unit
Control unit
PC
Instruction Memory
Splitter
Register file
ALU
JAL
Memory
Write buffer
Sign extend
ALU control
Clock

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.

Testing the Write Buffer is very trivial: verify that it outputs its input from the previous clock tick.