Single Cycle MIPS Processor

Carl-Johannes Johnsen

Department of Computer Science
University of Copenhagen

March 17, 2017

In this lecture, we will be combining the core components into a single cycle MIPS processor.

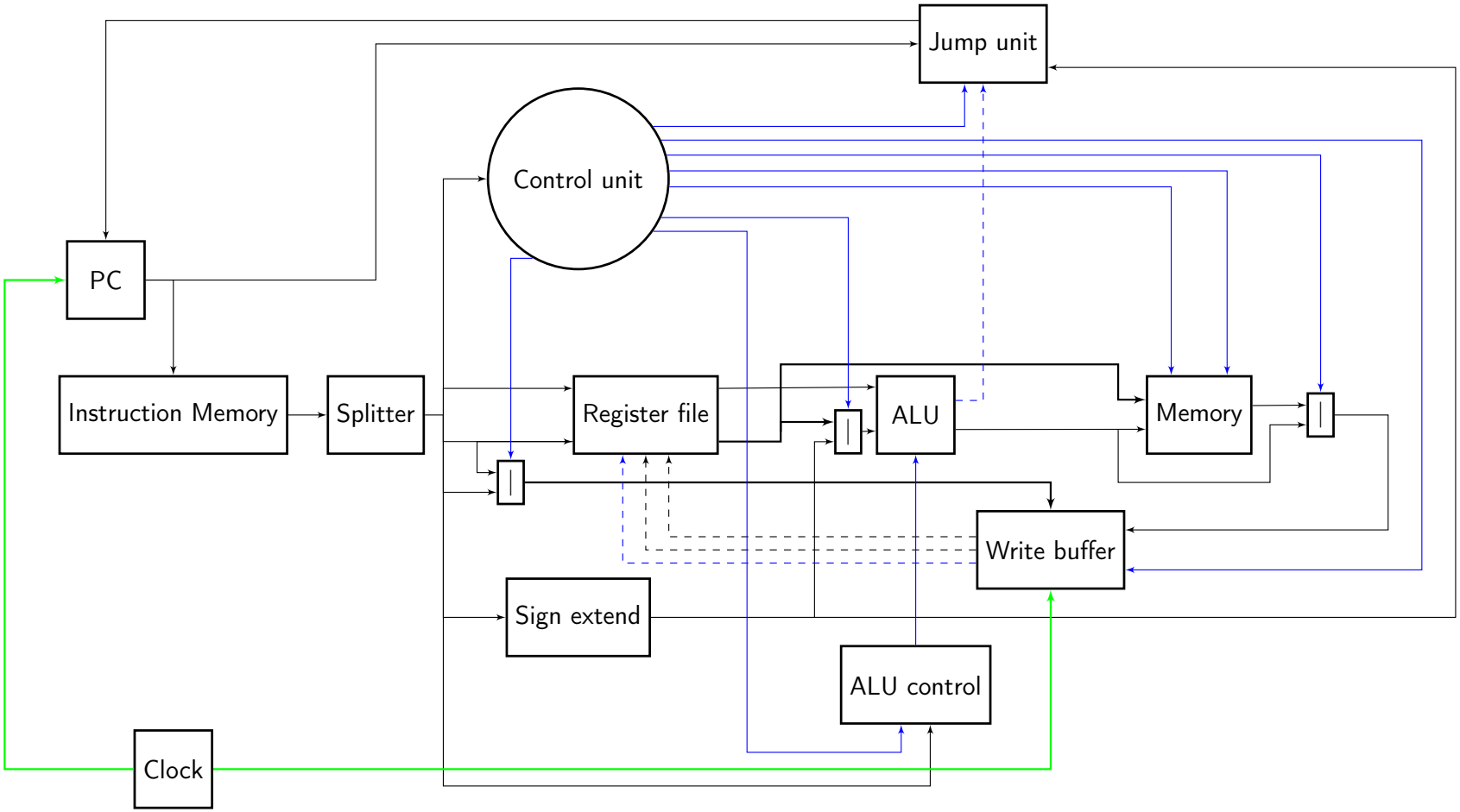
Then we will be looking at writing our first program, in order to verify that the processor works as intended.

Then we will be extending the processor, so that it can handle more instructions. Following each added instruction, we will be extending our initial program with the newly added instructions.

Finally, we will be writing two larger programs, and look into compiling them into hex values, that the processor can read.

Now that we have all the components and their busses, wiring up the processor is straightforward. We just need to declare the bus names, so that each process gets the bus with the corresponding name, and then SME will handle the wiring for us.

Note: as previously mentioned, the PC register and the Write Buffer should be clocked processes.


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

As mentioned before, the first single cycle MIPS processor should be able to handle `add`, `sub`, `and`, `or`, `slt`, `sw`, `lw` and `beq`. As such, the first program should consist of these.

Address	Instruction	opcode	rs	rt	rd/imm	shmt	funct	hex
0x00	add $3 $1 $2	0x00	0x01	0x02	0x03	0x00	0x20	0x00221820
0x04	sub $4 $3 $2	0x00	0x03	0x02	0x04	0x00	0x22	0x00622022
0x08	and $5 $3 $1	0x00	0x03	0x03	0x05	0x00	0x24	0x00612824
0x0C	and $6 $3 $1	0x00	0x03	0x03	0x06	0x00	0x25	0x00613025
0x10	slt $7 $6 $5	0x00	0x06	0x05	0x07	0x00	0x2A	0x00C5382A
0x14	sw $6 0x0($0)	0x2B	0x00	0x06	0x0000	-	-	0xAC060000
0x18	lw $8 0x0($0)	0x23	0x00	0x07	0x0000	-	-	0x8C070000
0x1C	beq $5 $4 0x4	0x04	0x05	0x04	0x0004	-	-	0x10A40004
0x20	add $9 $8 $6	0x00	0x08	0x06	0x09	0x00	0x20	0x01064820
0x24	add $10 $8 $6	0x00	0x08	0x06	0x0A	0x00	0x20	0x01065020

Inserting this program into the Instruction Memory is straightforward: just create an `byte` array, which is initialized to the hex values from the instruction column in the previous table.

Since we do not have any way of feeding values into the Register File at this moment, we are going to hardcode registers 1 and 2 with the values 5 and 2 respectively.

When the program has finished, the contents of the register file should be as this table:

Address	0	1	2	3	4	5	6	7	8	9	10
Value	0	5	2	7	5	5	7	1	7	0	14

We start by implementing the simple remaining R format instructions:

- `addu`
- `subu`
- `xor`
- `nor`
- `sltu`

The only modification that we need to do, is to extend the ALUOperation `enum`, and then add the remaining cases in the ALU Control and in the ALU.

The next instruction we want to add is the `ori` instruction, as we can then feed initial values into the Register File, and thus remove the hardcoding in the Register File. As we are extending the processor to handle the `ori` instruction, we should also add support for these instructions, as it requires the same amount of work:

- `addi`
- `addiu`
- `slti`
- `sltiu`
- `andi`
- `xori`

For the logical immediates, we need to ensure that the Sign Extend does not sign extend, but rather zero extends. To do this, we add an additional control signal in the Control Unit: `LogicalImmediate`. If this flag is set to `1`, the sign extend should not sign extend.

Then we just need to extend the opcode and the ALUOp `enum`, and add the remaining `case`s in the Control Unit and the ALU Control.

We are going to introduce another format: the J format. This format is used for executing the `j` instruction.

The first step is to extend the Splitter, as it should send the lower 26 bits to the Jump Unit.

Then the opcode `enum` should be extended, and the `case` for `j` should be added to the Control Unit. The Control Unit should have an additional control signal: `jump`, which should go to the Jump Unit.

Finally, the Jump Unit should be extended to handle the `j` instruction.

It should start by taking the 26 bits from the Splitter, and left shift by 2, just as we did with branching. Then it should take the 4 most significant bits of the incremented PC, and prepend them to the shifted address. Finally, we add a multiplexor, which takes the computed jump address, the previous output address and the `jump` signal as inputs. If the control signal is `0`, then the original output should be put on the output bus, and the jump address otherwise.

Then we are going to add the remaining branching instructions:

- `bne`
- `blez`
- `bgtz`

To implement the `bne` instruction, we are going to add another signal from the Control Unit to the Jump Unit: `bne`. Then we should split the Zero signal into two, and add a `NOT` gate, where one of the Zero signals should go to. Then we add a multiplexor, which takes Zero, not Zero and the `bne` signal as input. If the `bne` signal is `0`, then the output should be the original Zero, and the not Zero otherwise. This output should go into the `AND` gate, where the Zero signal was before.

Then we are going to add the remaining jump instructions:

- `jr`
- `jal`
- `jalr`

As these are useful when writing the larger programs later.

We start with `jr`. The instruction is in R format, so we do not know it is `jr`, until it has reached the ALU Control. As such, we are going to need a control signal from the ALU Control to the Jump Unit. We are also going to forward Output A from the Register File to the Jump Unit, as this is the address that the processor should jump to in the `jr` instruction.

The Jump Unit should not compute the new address in the same manner as with the `j` instruction. This is due to the registers being a full 32 bit, and thus can contain the whole address space.

The Jump Unit should also have a multiplexor, controlling whether the jump address should be the immediate value, or if it should be the value from the instruction.

For the `jal` instruction, we are going to need an extra unit following the ALU.

In the case of a `jal` instruction, we should store the PC+4 address in register `31` (which is called the `$ra` register). There should be an additional control signal from the Control Unit: the `jal` signal. The new JAL Unit should take three inputs: the ALU Result, the PC+4 and the `jal` control signal. It should produce two outputs: the Write Address for the Write Buffer, and the value to store. If the `jal` signal is `1`, the JAL Unit should output the PC+4 on the value bus, and `31` on the address bus. Otherwise it should output the regular ALU Result, and the regular Write Address.

Then we are going to add the shift instructions:

- `sll`
- `slr`
- `sra`
- `sllv`
- `srlv`
- `srav`

The shifting itself is performed in the ALU, and modifying the ALU to handle these is straightforward.

The problem is that in an R format instruction, which the shift operations are, the shift amount (`shamt`) is stored in its own field within the instruction. As such, the splitter should extract these 5 bits, and send them to a new multiplexor, which also takes Output A from the Register File.

As with the `jr` instruction, we do not know it is a shifting instruction until it reaches the ALU Control. So to control the new multiplexor, we need a Control signal from the ALU Control, indicating whether the multiplexor should output either the `shamt` or Output A from the Register File.

Then we are going to add the multiplication and division instructions:

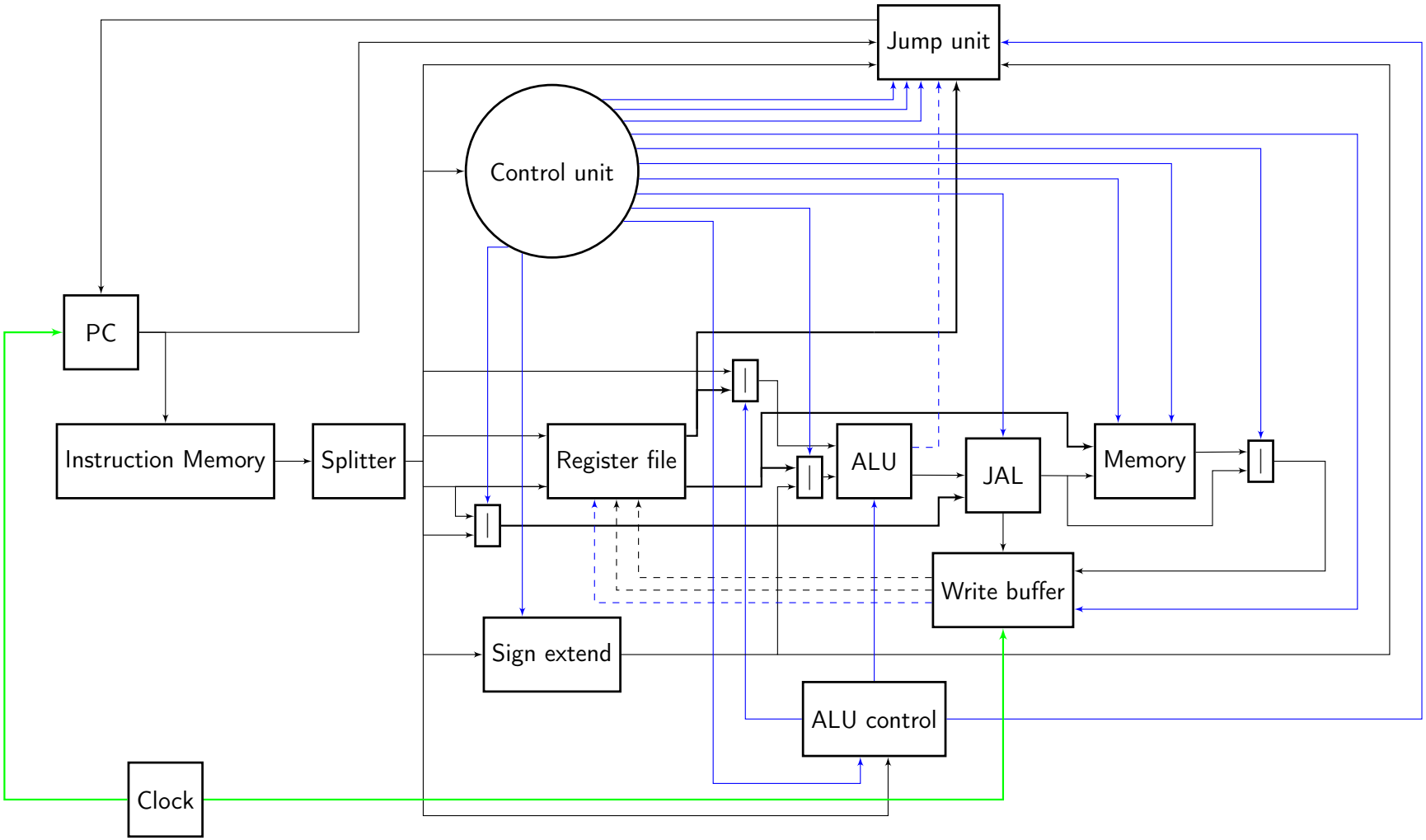
- `mult`
- `multu`
- `div`
- `divu`

All of these instructions put their result in two special registers: HI and LO. As such, to get the results from them, we are also going to need the following instructions:

- `mfhi`
- `mthi`
- `mflo`
- `mtlo`

We start by adding the special registers. Since they are performed in the ALU, we might as well put them there. Then, when we are doing the computation, we should just put the values there, and since the instructions do not write to registers, touch memory or change the PC register, it does not matter what is put on the ALU Result or Zero busses.

The instructions handling the moving to and from the HI and LO registers are simple to implement: just either input or output the corresponding register, to or from the ALU.


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