VHDL MIPS processor implementation

Auteurs :
Gabriel FREITAS OLIVEIRA

Last Update
July 4, 2020

Chapter 1

General Architecture

This document aims to describe a simple MIPS architecture processor that was implemented in VHDL using Quartus II. First, let's take a look on the requirements of the project.

- The processor must be a multi-cycle variant
- It must use a 16 bit architecture
- It must have 8 registers on cache, where register 0 is always 0 (also referenced as $ZERO)
- It must interact with the user with a 6-bit input and a 16- bit output
- It must execute these instructions
  - **ADD $DestReg, $Reg1, $Reg2** → $DestReg = $Reg1 + $Reg2
  - **ADDI $DestReg, $Reg1, Constant** → $DestReg = $Reg1 + Const
  - **NOR $DestReg, $Reg1, $Reg2** → $DestReg = $Reg1 NOR $Reg2
  - **AND $DestReg, $Reg1, $Reg2** → $DestReg = $Reg1 AND $Reg2
  - **BEQ $Reg1, $Reg2, Jump_to_line** → if($Reg1 == $Reg2) PC = LINE
  - **JUMP Jump_to_line** → PC = Line
  - **LOAD $DestReg, $RAM_ADDR** → $DestReg = $RAM_ADDR
  - **STORE $SrcReg, $RAM_ADDR** → $RAM_ADDR = $SrcReg
  - **SLT $RSet, $R1, $R2** → if($Reg1 < $Reg2) $RSet = 0xFFFF
  - **IN $DestReg** → $DestReg = IN
  - **OUT $SourceReg** → OUT = $SourceReg
  - **MOVE $DestReg, $SourceReg** → $DestReg = $SourceReg
  - **BGT $Reg1, $Reg2, Jump_to_line** → if($Reg1 > $Reg2) PC = LINE

Knowing that, I decided to separate the instruction in 4 parts: 4-bit OP (or instruction ID), 3-bit First Segment (Always for R1 address), 3-bit second segment (useg for R2 address and / or write address), and 6-bit last segment for miscellaneous. With that in mind, here is the proposed architecture:

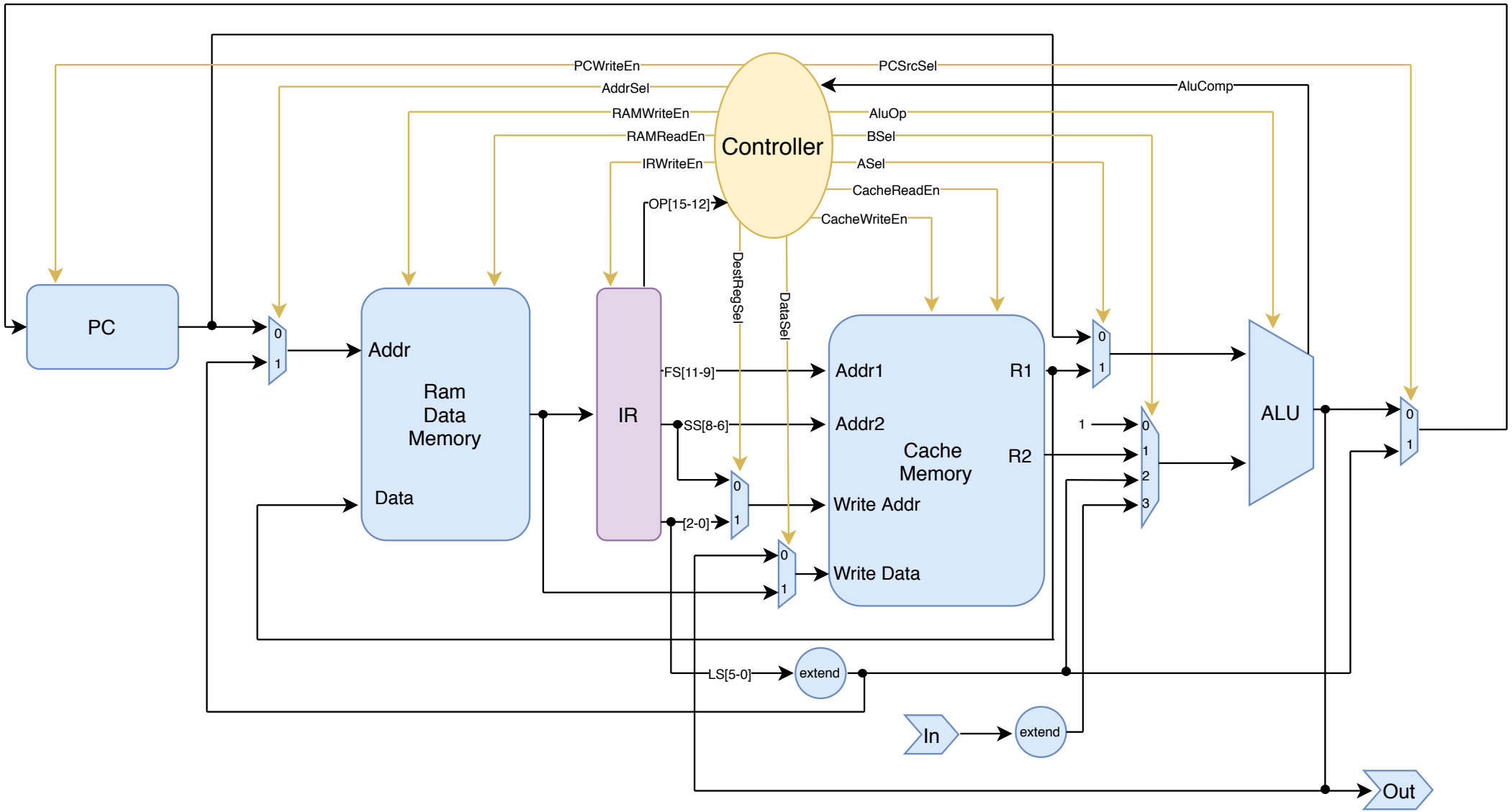

Figure 1.1: Processor Architecture

Every signal that does not have an end goes to / comes from the controller, that is responsible for enabling the correct signals in order to make the processor work (each memory component has a clock and a reset that have been omitted). The next step is to define our instructions based on the assembly instruction set (X = don't care). It is important to note that any instruction ID that is not represented in the table, will be considered as NOP.

Instruction Table				
Instruction	ID (binary)	Assembly Example	Arguments	Machine code
ADD	0001	ADD $R3, $R1, $R2	R3 → Destiny Register R1 → Source register 1 R2 → Source register 2	0001 001 010 000|011
ADDI	0010	ADDI $R3, $R1, 10	R3 → Destiny Register R1 → Source register 1 10 → 6 bit constant	0010 001 011 001010
NOR	0011	NOR $R3, $R1, $R2	R3 → Destiny Register R1 → Source register 1 R2 → Source register 2	0011 001 010 000|011
AND	0100	AND $R3, $R1, $R2	R3 → Destiny Register R1 → Source register 1 R2 → Source register 2	0100 001 010 000|011
BEQ	0101	BEQ $R1, $R2, 10	R1 → Equals operator 1 R2 → Equals operator 2 10 → Line to jump to	0101 001 010 001010
JUMP	0110	JUMP 10	10 → Line to jump to	0110 XXX XXX 001010
LOAD	0111	LOAD $R1, 10	R1 → Destiny Register 10 → RAM address	0111 XXX 001 001010
STORE	1000	STORE $R1, 10	R1 → Source Register 10 → RAM address	1000 000 001 001010
SLT	1001	SLT $R1, $R2, $R3	R1 → Register to be set R2 → Less operator 1 R3 → Less operator 2	1001 010 011 000|001
IN	1010	IN $R1	R1 → Destiny Register	1010 000 001 XXXXXX
OUT	1011	OUT $R1	R1 → Data Source	1011 000 001 XXXXXX
MOVE	1100	MOVE $R1, $R2	R1 → Destiny Register R2 → Source Register	1100 000 010 000|001
BGT	1101	BGT $R1, $R2, 10	R1 → Greater operator 1 R2 → Greater operator 2 10 → Line to jump to	1101 001 010 001010

It is important to note that the JUMP-like instructions and LOAD/STORE use absolute addresses, meaning that the value passed will be used as it is i.e. JUMP 10 will jump to line number 10. This approach is not ideal because it will limit the possibilities only to addresses 0 to 63, but this project aims to make a simple version of a processor, so the trade off is acceptable. The limitations of this kind of architecture are:

- Only holds a maximum of 16 different instructions
- Register bank can only have a maximum of 8 registers
- ADDI instruction can only assign values between 0 and 63, or -32 and 31
- Jump instructions can only jump to lines between 0 and 63
- LOAD and STORE can only get data in addresses 0 to 63

To test the processor, this simple code has been made

```
int main(void)
{
    int i, j, k;
    int input, output;

    i = 22;
    j = -2;
    k = 16;
    input = 0;
    output = 0;

    while (input != 2) {
        if (input > k)
            output += j;
        else
            output += i;
        scanf("%d", input);
    }
    printf("%d", output);
}
```

```
    ADDI $R1, $ZERO, 22 //i = 22
    ADDI $R2, $ZERO, -2 //j = -2
    ADDI $R3, $ZERO, 16 //k = 16
    ADDI $R4, $ZERO, 0 //input = 0
    ADDI $R5, $ZERO, 0 //output = 0
    ADDI $R6, $ZERO, 2 //temp = 2
LOOP:
    BEQ $R4, $R6, END //END if ==
    BGT $R4, $R3, IF //IF if >
    ADD $R5, $R5, $R1 //output+=i
    JUMP ENDIF //goto ENDIF
IF:
    ADD $R5, $R5, $R2 //output+=j
ENDIF:
    IN $R4 //scanf
    JUMP LOOP //While
END:
    OUT $R5 //printf
```

This code was chosen to use various different instructions, while only using 6 registers from the cache. Using Quartus simulation (Modelsim does not work in my computer, so I had to work with Quartus), I ran a timing simulation that takes 3us with a 50 MHz clock. The results can be verified in figure 1.2

Since input will always begin as 0, the output will be 22 -2*n where n is the times the IF part is executed. By counting the ALU compare high states, we can verify how many times the "IF" part was ran, in our case, 3 times (the last one is the BEQ to end the loop), thus output = 16. Using the simulation, I theorize a maximum clock of 80-90

MHz, but since this code was not tested on a real board, this number can vary a lot. The compilation was made using a Cyclone II EP2C5T144C8 containing 89 pins, while the debugging used Cyclone III EP3C120F484C7 containing 284 pins (every signal was outputted on the pins, thus needing a large amount of them). If you want more details, the next section will cover the component details.

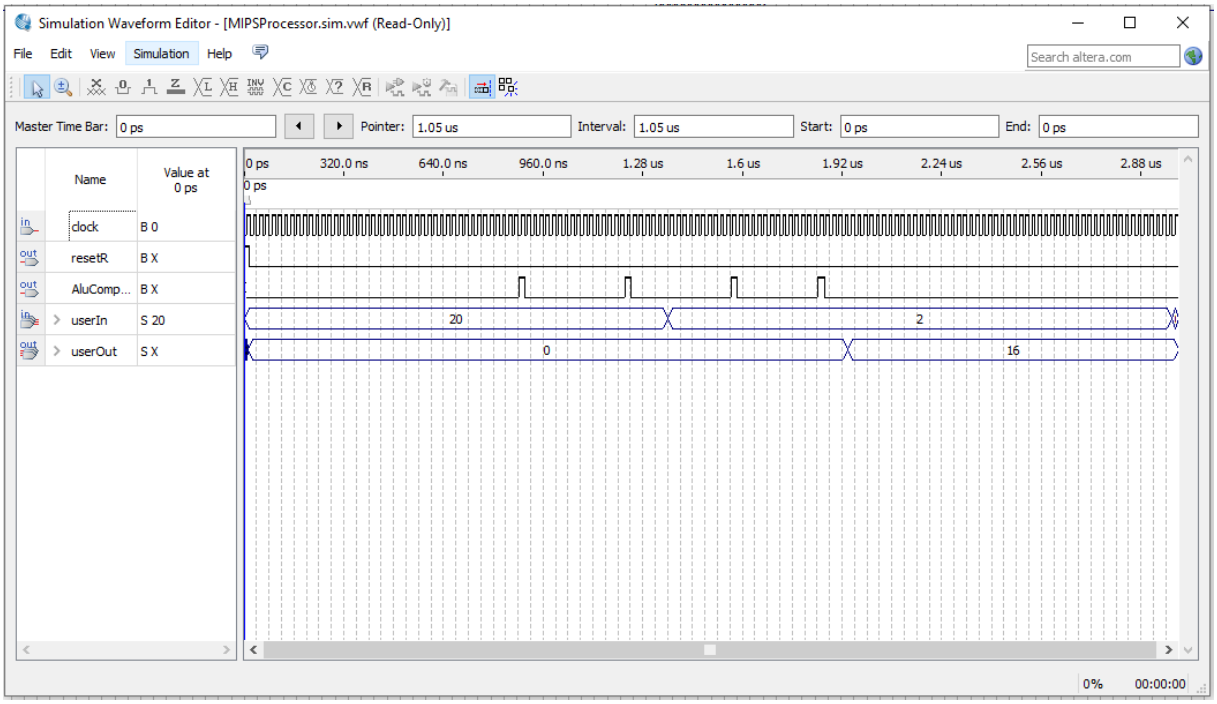

Figure 1.2: Processor Results

Chapter 2

Component details

This section will give some details about the components that make the processor

2.1 Registers

All registers in this project (Instruction Register, PC register, and Out register) have an asynchronous reset, a clock, an input, and an output. All of them are sensible to the rising edge of the clock, when they get updated if write control bit is enabled

2.2 RAM

The RAM is initialized with the instructions already in place (addresses 0 to 13), and data memory should be loaded / stored in addresses outside of it. Like registers, it has an asynchronous reset, is sensible to the rising edge of the clock, when the register indicated by the address input get updated if the write enable control bit is enabled. The difference is that the output will be maintained as long as the read control bit is disabled, and update on the next control clock, reading the register indicated by the address input. To not use all the board's resources only the 4 LSBs are kept from the PC, creating 16 registers (addresses 14 and 15 are initialized as 0). During reset, RAM will output the value in register 0.

2.3 Cache

Like RAM, it has an asynchronous reset, is sensible to the rising edge of the clock, when the register indicated by the address input get updated if the write enable control bit is enabled, and the outputs will be maintained as long as the read control bit is disabled. When the read control bit is enabled, in the next rising edge, both outputs will be updated. All registers are initialized with 0, and the register in the 0 address

(aka $ZERO) is write-protected by hardware. During reset, cache will output the value in register 0 for both outputs.

2.4 Signal extend

The signal extend is an asynchronous component that extends its input from 6 bits, to 16 bits. This component has a mode input, that decides whether it will make a zero extend (filling the MSB as 0 if mode is '0'), or a sign extend (filling as '1' if the 6th bit is 1, keeping the sign, if mode is '1').

2.5 ALU

The ALU is another asynchronous component that receives 2 numbers and 1 mode, and outputs AluRes as result, and AluComp as compare (if applicable). The following table describes every operation possible in the ALU. Note that the less than operation also sets the result as 0xFFFF, that is required by the SLT instruction.

ALU operation Table			
Operation	ID (binary)	Result	Compare
ADD	001	A + B	0
NOR	010	A NOR B	0
AND	011	A AND B	0
Equals	100	0	1 if A == B, 0 otherwise
Less than	101	0xFFFF if A < B, 0 otherwise	1 if A < B, 0 otherwise
Greater than	110	0	1 if A > B, 0 otherwise
others	—	0	0

2.6 Controller

Now the last and most important component, the controller. It's job is to set every control signal in the processor so that the processor does the correct instruction. The first state is the Reset, that resets every memory element to 0, and the case of RAM, inserting the instructions. Then, it goes to the Fetch state where it uses the ALU to calculate PC + 1, and enable writing in the PC and instruction registers. The next state is the Decode one, where the controller uses the operation ID (OP) to decide which instruction to execute. Even though every single instruction has its own state, the controller figure will simplify it by 3 types, the R-Type (ADD, ADDI, NOR, AND, SLT, IN, OUT, and MOVE), does pretty much the same thing, reads from the cache, does an operation using the ALU, and writes back into the cache. The J-type that are instructions that change the PC (BEQ, JUMP, BGT), since the PC is only updated in

the rising edge, we need a waiting state to change the PC before going back to Fetch (where PC is written by PC + 1). Lastly, we have the LS-type (load and store) that first load the data (cache or RAM), then in the next cycle, write it on RAM or cache.

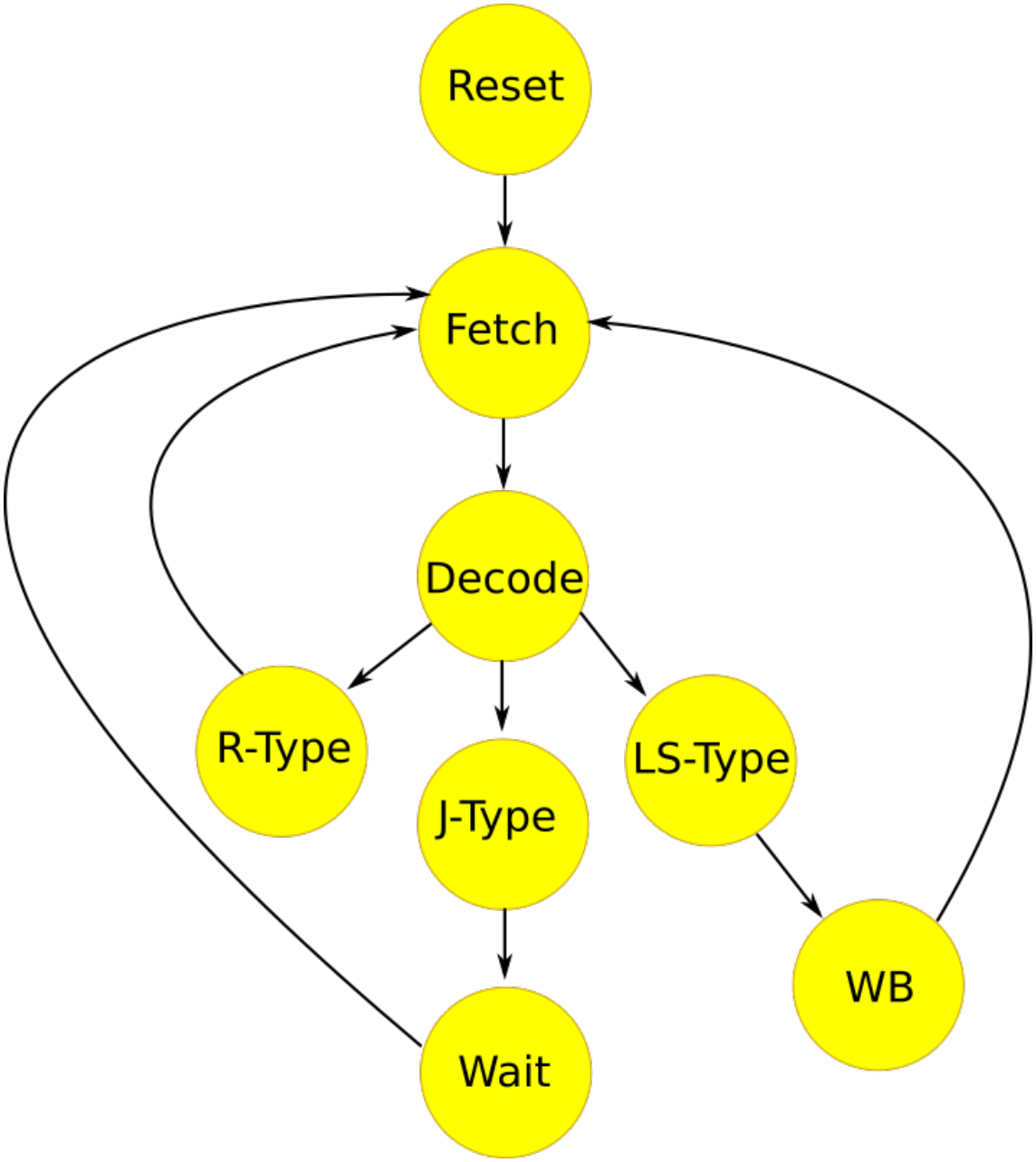

Figure 2.1: Controller