Lecture 6 - MIPS design multi cycle

Gidon Rosalki

2025-05-04

1 Reminder: MIPS instruction formats

Assembly is a machine language expressed in symbolic form, using decimal and naming. Moving data to and from memory uses imm($rs) which refers to Memory[ea], where the effective address in memory is ea = imm + $rs. Branch and jump instructions refer to words, not bytes, and are calculated relative to the program counter PC + 4, and **not** the pc. There are 3 types of instructions, R-type, I-type, and J-type. R for register to register arithmetic, I for immediate, and J for jump.
These instructions operate on the processor as well, which is a state machine that includes the internal state (instruction pointer, registers, flags), the state in main memory, and changes its state based off the program.

2 Performance, frequency, and CPI

2.1 Timing of single cycle MIPS

The **critical path** is the path from PC, I-Mem, RegFile, ALU, D-Mem, RegFile. Thus we need

$$t_{\text{cycle}} \geq t_{\text{fetch}} + t_{\text{decode}} + t_{\text{execute}} + t_{\text{memory}} + t_{\text{writeback}}$$

even though some instructions need fewer stages, for example:

- The datapath for the R-type instructions is PC, I-Mem, RegFile, ALU, RegFile
- For branch is PC, I-Mem, RegFile, ALU, PC
- For jump is PC, I-Mem, PC

CPUs work at the rate of the clock cycle.

- The clock cycle is measured in nsec (10^{-9})
- The clock frequency, which is 1 clock cycle, is measured in GHz (10^9)
- IC - Instruction Count, the number of instructions executed in a program
- CPI - Cycles per Instruction. This is the average number of cycles per instruction (in a given program)
- IPC ($= \frac{1}{\text{CPI}}$): Instructions per Cycle

The CPU time (execution time) is the time required to execute a program:

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{Clock cycle}$$

Our goal is to minimise the CPU time, which can be done by minimising the clock cycle, the CPI, and the IC. This is measured by:

$$\text{Speedup} = \frac{\text{CPU TIME}_{\text{old}}}{\text{CPU TIME}_{\text{new}}}$$

There are other goals, not shown in this class, such as minimising power per operation.

To calculate the CPI of a program, we need

- IC_i: the number of times the instruction of type i is executed in the program
- IC: the number of instructions executed in the program

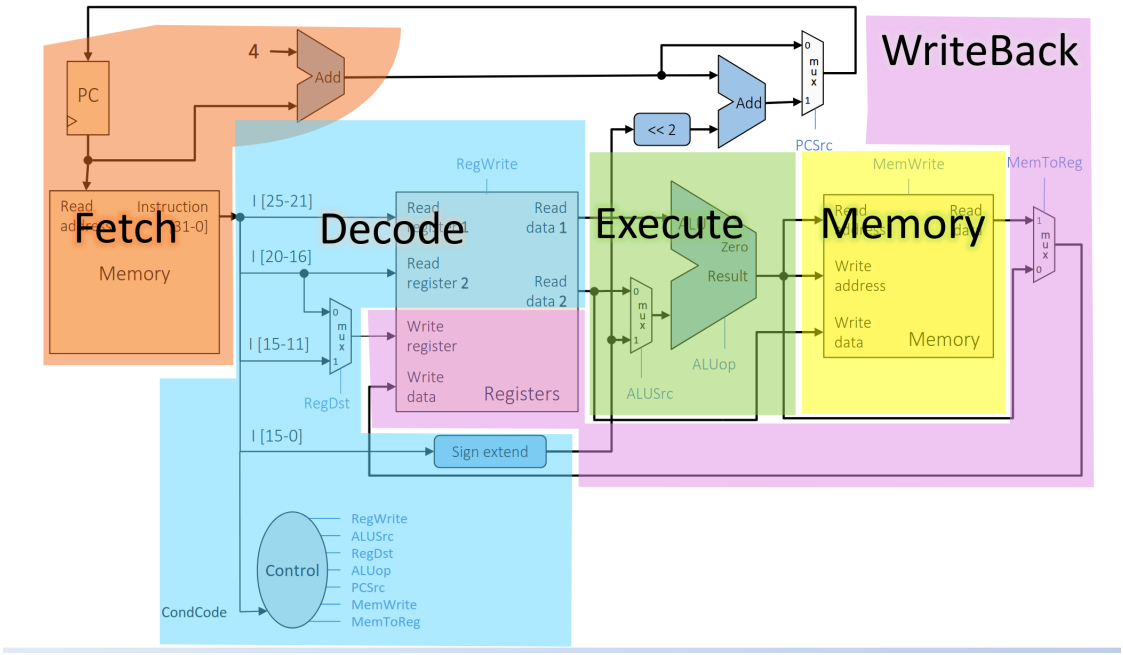

Figure 1: MIPS single cycle data path

This holds that

$$IC = \sum_i IC_i$$

we additionally have

- $F_i = \frac{IC_i}{IC}$: The relative frequency of the instruction i
- CPI_i: The number of cycles required to execute the instruction i (on average). For example $CPI_{\text{add}} = 1$, $CPI_{\text{mul}} = 3$.
- The number of cycles required to execute the program is given as

$$\#cyc = \sum_i CPI_i \times IC_i = CPI \times IC$$

- CPI:

$$CPI = \frac{\#cyc}{IC} = \frac{\sum_i CPI_i \times IC_i}{IC} = \sum_i CPI_i \frac{IC_i}{IC} = \sum_i CPI_i \times F_i$$

To compare performance, we first need to be able to measure it. We may measure performance through a number of mehtods:

- Real HW: This is fast, and easy to do long runs, but hard to analyse
- Simulators: This is slower, which limits the length of the run, and the data
- Analysis: What if... (e.g. Good old pen and paper, calculating it manually)

When measuring performance, we also have peak performance, measured in MIPS, MFLOPS. This is often not useful, due to being an unachievable / unsustainable level in practice. Additionally, CPI / IPC only has meaning on the same ISA, using the same compiler, so is not necessarily a good measurement for comparison. We generally use benchmarks to measure performance, since they can be more general. A benchmark is a program, that we can run on any hardware, and measure how quickly it runs, and so on. These are generally based off real applications, or representative parts of real apps. (See `www.spec.org` for examples).

2.2 Frequency

We need to ask ourselves how long a cycle should be. Consider the following tables:

Instruction class	Functional unit used by instruction				
R-Type	Fetch	REg Read	ALU	Reg write	
Load	Fetch	Reg read	ALU	Mem read	Reg write
Store	Fetch	Reg read	ALU	Mem write	
Branch	Fetch	Reg read	ALU		
Jump	Fetch				

Table 1: Required functional units

Instruction class	Instruction memory	Register read	ALU operation	Data Mem read or write	Register write	Total inst
R-type	200	50	100		50	400ps
Load	200	50	100	200	50	600ps
Store	200	50	100	200		550pse
Branch	200	50	100			350pse
Jump	200					200ps

Table 2: Timing table

The cycle time is fixed, and so we need to take the longest path (in our case, $600psec$). So, the frequency will be

$$\frac{1}{600 * 10^{-12}} = 1.67GHz$$

If we consider a variable cycle, then it depends on the cycle mix. Consider a cycle of 25% load, 10% store, 45% R-type, 15% branch, and 5% jump. Thus, the average CPU clock cycle would be:

$$600 * 0.25 + 550 * 0.1 + 400 * 0.45 + 350 * 0.15 + 220 * 0.05 = 447psec$$

resulting in an average frequency of $2.2GHz$. So how can we implement a variable cycle? We currently only know the instruction after we carried out the fetch, and it is very difficult to change the clock in the middle of the cycle, so performance is wasted, since we take the worst case.

In order to improve performance, assuming a given instruction set, improving the speed of a component is difficult and expensive, but we can change the critical path. We could also use a multicycle approach, where we break the execution into stages, where each stage only uses the required resources, and thus each stage can be executed faster. We save the results at the end of one stage to the beginning of the next, and we can reuse components to save on hardware.

Each cycle takes 2 - 5 cycles, and in each cycle, a single stage is performed. Thus we need

$$t_{\text{multi cycle}} \geq \max\{t_{\text{fetch}}, t_{\text{decode}}, t_{\text{execute}}, t_{\text{memory}}, t_{\text{writeback}}\}$$

Stages take different times, so in our case

$$t_{\text{multi cycle}} > \frac{1}{4} t_{\text{single cycle}}$$

Some instructions take less time, but lw takes more time. It is OK if the average CPI is less than 4.

Let us slowly build up the multi cycle CPU, starting with just R-type, then adding in lw, and then sw.

We can save time by reusing the ALU. While the earlier parts of the processor cycle are performing their parts, we may simultaneously use the ALU to increment the program counter by 4.

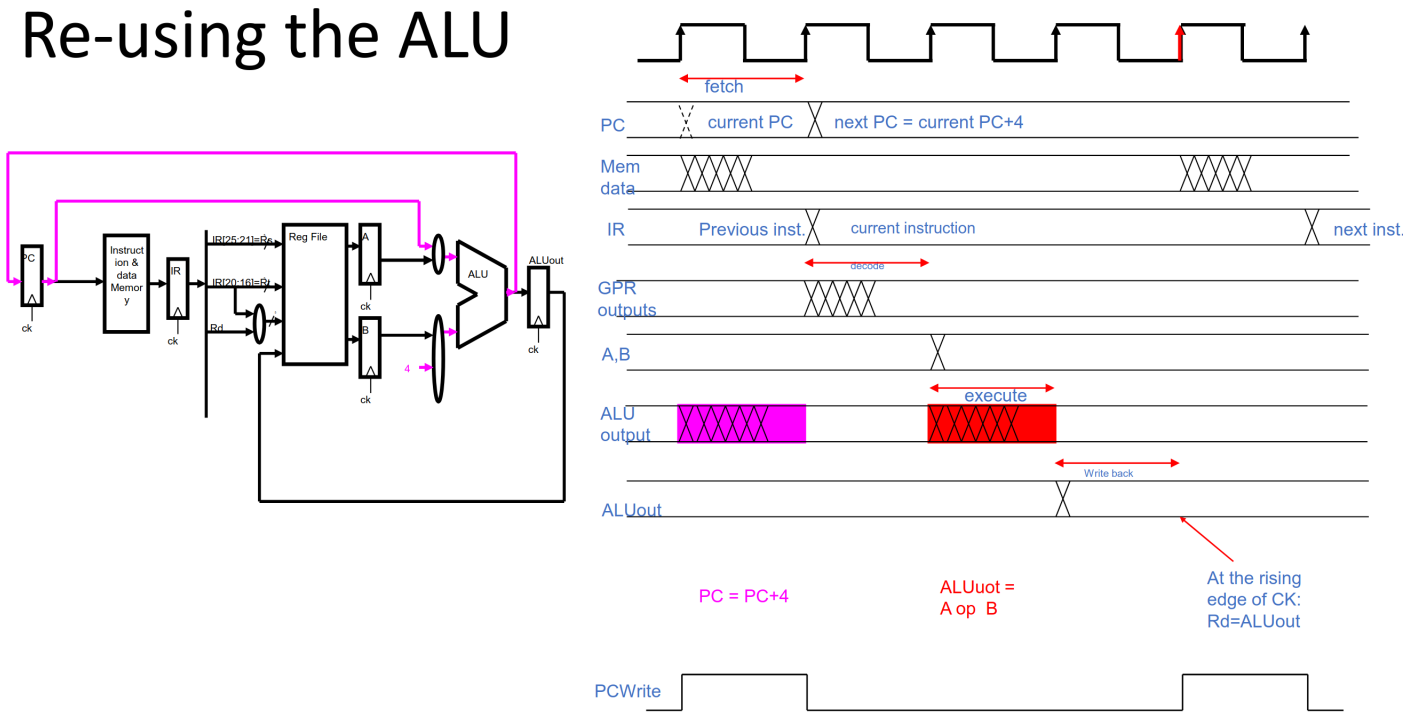

Figure 2: Reusing the ALU

To add in the load word instruction to our CPU, we need to add the following data path:

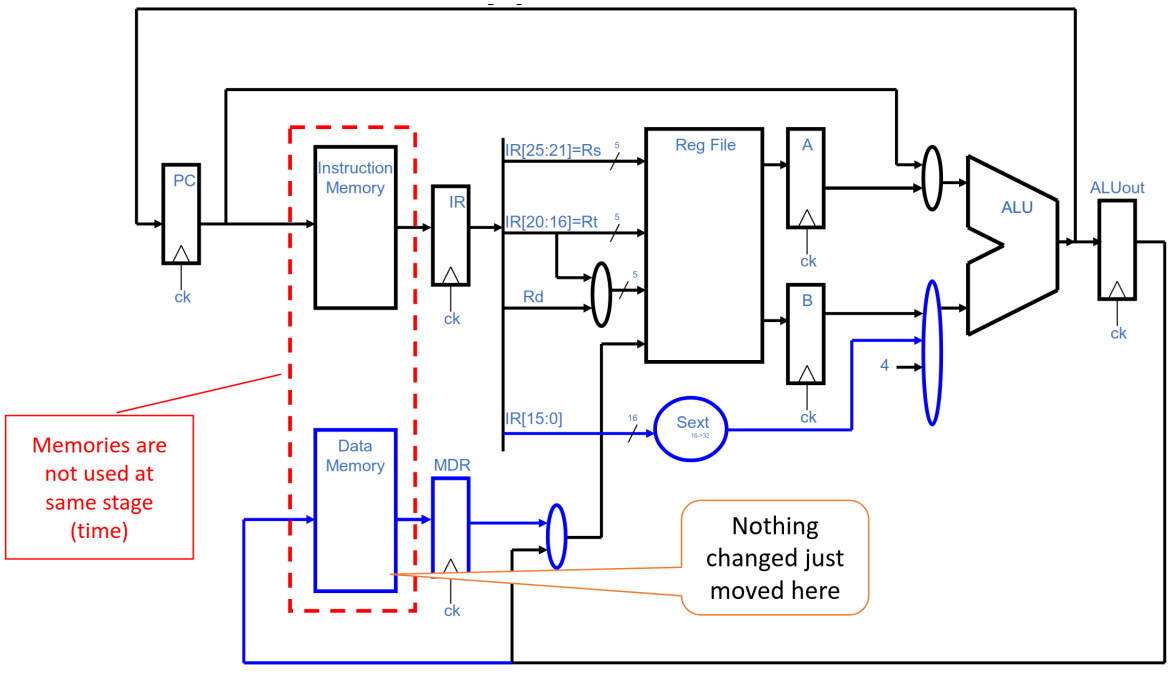

Figure 3: Data path for the r-type, and lw CPU

Let us also note the following, the instruction memory, and the data memory are not used at the same time, the instruction memory is only used in the first part of the CPU cycle, where the data memory is only used in the 4th part, so we can combine them into a unified memory. To add in sw we can do the following:

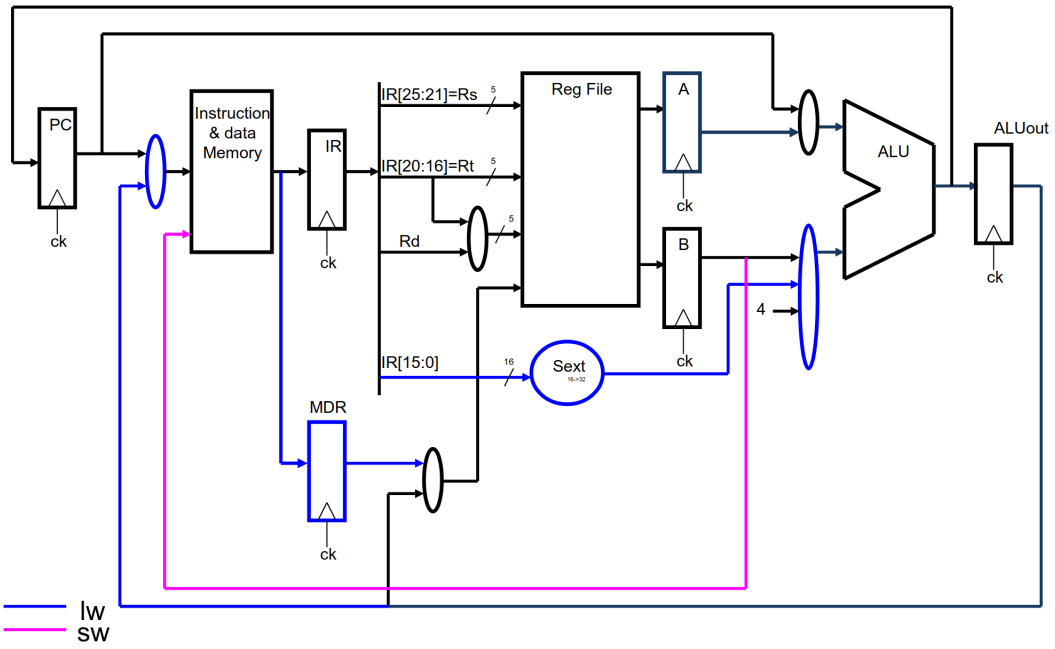

Figure 4: Data path for the r-type, lw, and sw CPU

To add in branch we need to add in the following data path, since we need to add in the sign extending on the immediate, and so on.

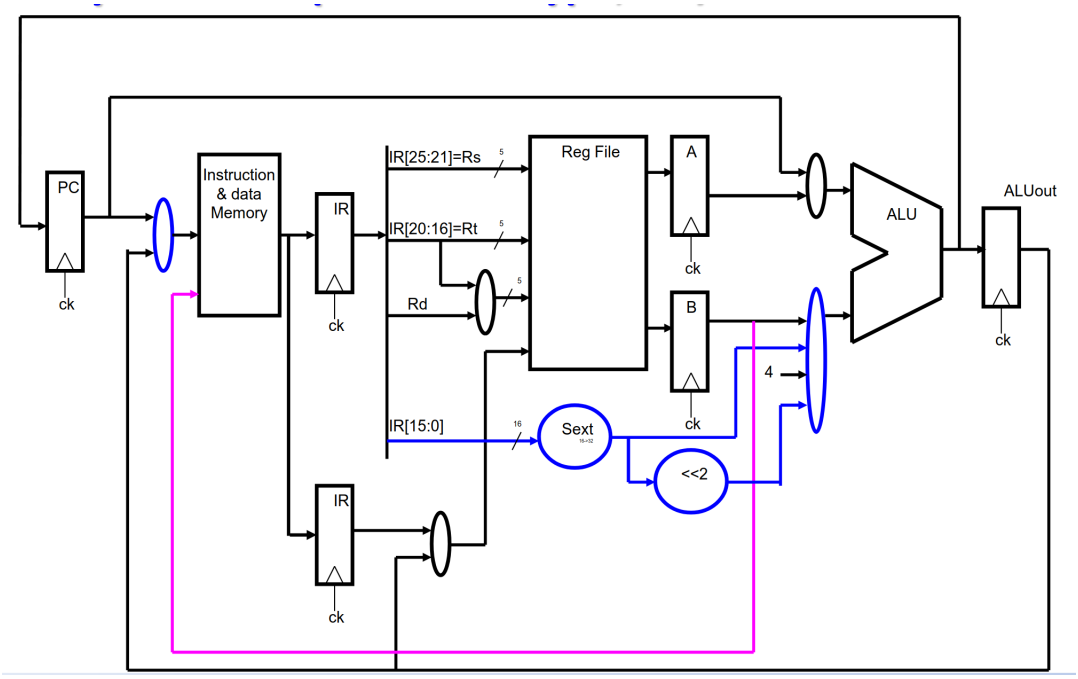

Figure 5: Data path for the r-type, lw, sw, and branch CPU

We can improve performance here by pre-calculating the branch target. We use the decode state in which the ALU is idle to compute the branch target address, and then store it for 1 cycle until the branch decision is known (we can use ALUout for this purpose).

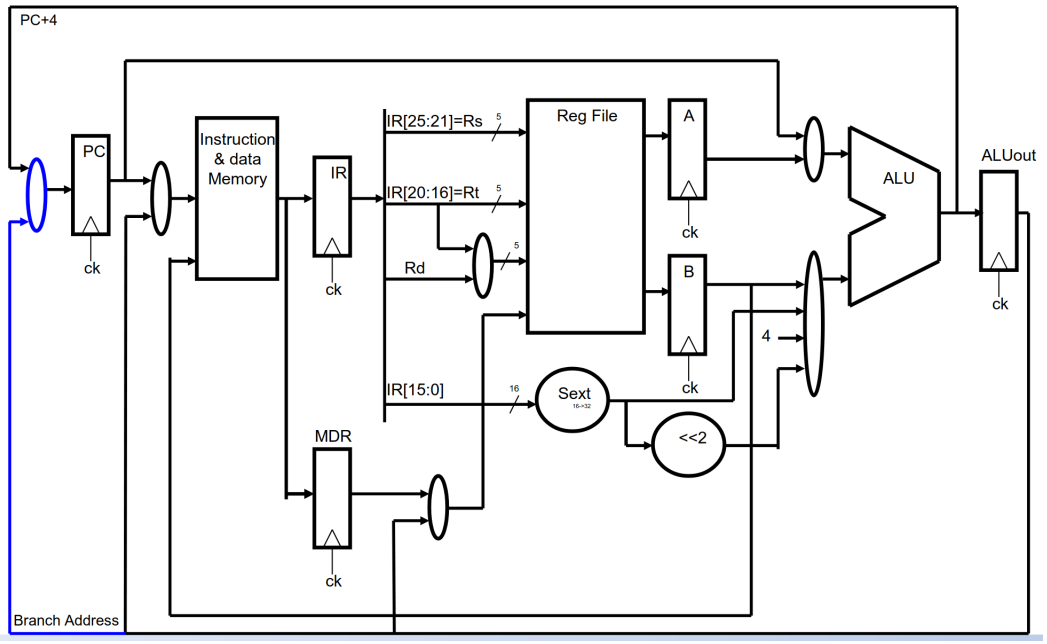

Figure 6: Optimised data path for the r-type, lw, sw, and branch CPU

To add in the jump command, all we need is to remember that it just changes the PC as follows:

$$PC = PC[31:28]|IR[25:0] << 2$$

and so may use the following structure:

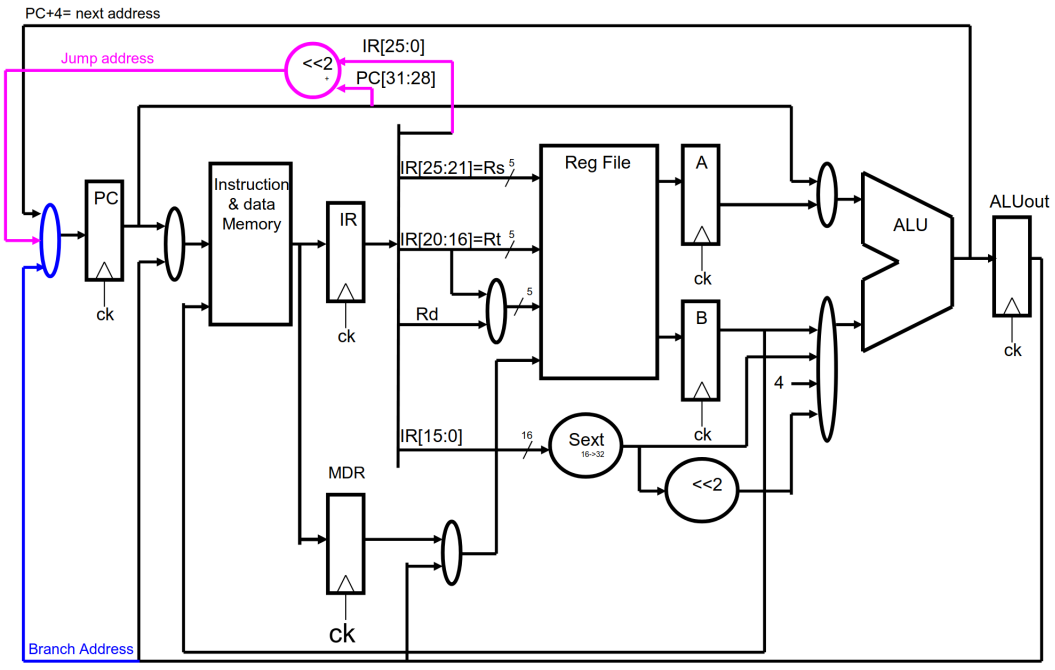

Figure 7: Data path for the r-type, lw, sw, branch, and jump CPU

This final CPU we have created may be considered as the following state machine:

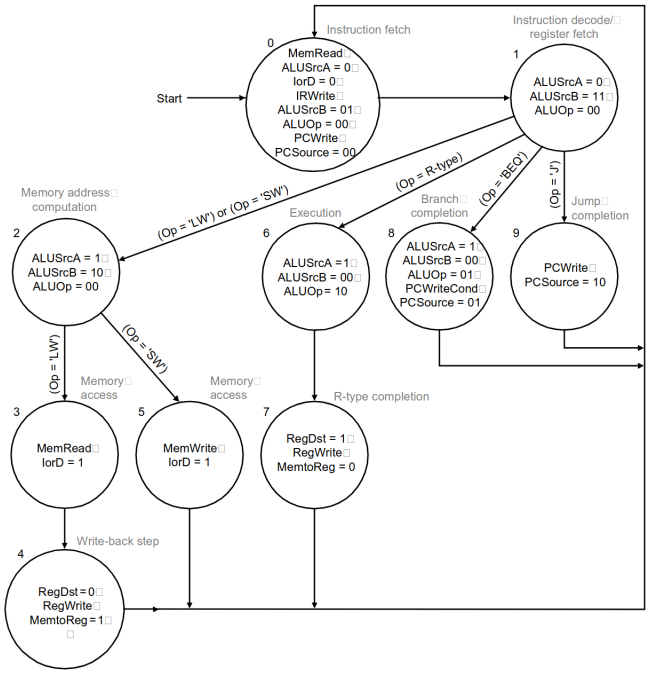

Figure 8: CPU state machine

So in summary, our new CPU stages for the multicycle variant are as follows:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	0 IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	1 A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B 6	ALUOut = A + sign-extend (IR[15-0]) 2	if (A ==B) then PC = ALUOut 8	PC = PC [31-28] II (IR[25-0]<<2) 9
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut 7	Load: MDR = Memory[ALUOut] 3 or 5Store: Memory [ALUOut] = B		
Memory read completion		4Load: Reg[IR[20-16]] = MDR		

Figure 9: CPU state machine

3 Summary: Single cycle and Multi cycle

In a single cycle CPU, each instruction is executed in one clock cycle, and the cycle time is determined by the longest path (of a complete instruction). In a multi cycle CPU each instruction is executed in multiple clock cycles (stages), and the cycle time is determined by the path of the longest stage. There are a variable number of cycles for each instruction, and this enables reuse of idle hardware. The performance is given by

$$CPI \times \#instructions \times t_{\text{cycle}}$$

where the CPI is fixed for single cycle MIPS, and application dependent for multi cycle MIPS, the number of instructions is application and architecture dependent, and the cycle time is implementation (uArch) dependent.