

Inf2C Computer Systems

Tutorial 5, Week 9

Solutions

1. Multi-cycle processor

Discuss the steps in executing the `jal`, the `lw` and the `add` instruction in the multi-cycle datapath presented in Figure 2.

Now assume that an access to the register file takes 6.5ns, an access to the memory takes 100ns and the ALU delay is 6ns. Assume also that the instruction's execution stalls when it waits for a resource to produce a result. If we were to optimize this simple processor, which should be the component we should spend our main design efforts on, what could we do to improve it?

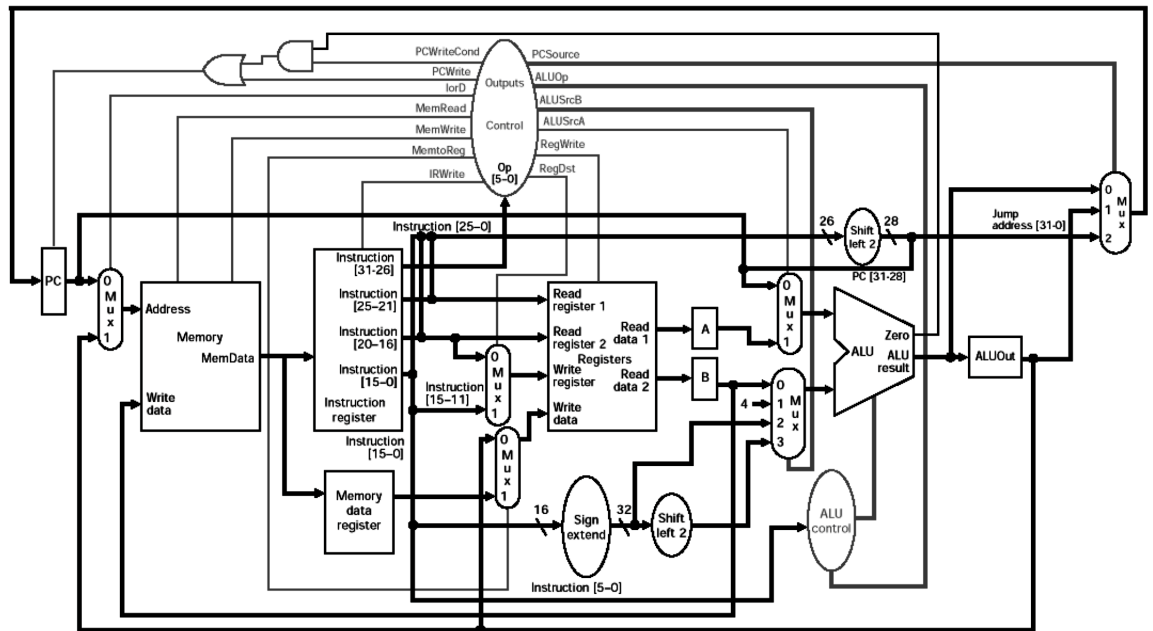


Figure 1: MIPS Datapath

Answer: The first two cycles are the same for all the instructions. During the first cycle, we fetch in the Instruction Register from the memory location indicated by the current PC, and we increment the PC by 4.

In the second cycle, we read the values in the register file registers addressed by bits 26-21 and bits 20-16 of the loaded instruction into registers A and B respectively. We also use the ALU to speculatively compute a branch destination address, though that is not needed by any of the instructions under consideration.

JAL:

JAL (jump and link) requires that we save the current program counter ($PC+4$, where PC is the address of the JAL instruction) to register 31. We can do so in the 3rd cycle with a couple of small modifications: we need the write register multiplexer extended to take 31 as a extra input and the write data multiplexer extended to take the current value in the program counter. At the same time we compute the jump location using the shift left by 2 of the lower 26 bits of the Instruction Register concatenated with the upper 4 bits of the current program counter. We then load this new value into the program counter at the end of the 3rd cycle.

LW:

The LW is a memory instruction, so in the third cycle we have to form the effective address. We use the ALU to add the A register contents to the sign extended Immediate value from the Instruction Register.

In the fourth cycle, we access the memory using the address computed in the previous cycle. The loaded data is stored in the Memory Data Register MDR.

In this final cycle, we load the destination register indicated by the bits 16 to 20 of the Instruction Register with the value in MDR.

ADD:

The ADD is a register-register operation. So in the third cycle, we compute the addition using the ALU of the contents of register A and register B.

In the fourth cycle we load the register indicated by the bits 11 to 15 of the instruction register with the computed value.

In the final section, note that the memory operation costs a lot more than the access to the register file and the ALU. Therefore one should try to improve that. The obvious way to do it is to use a cache to provide a fast access for the vast majority of the memory operations.

2. Adding a new instruction to the multi-cycle processor

Adapted from P&H 3/e.

You wish to add a new instruction `jmt` (jump to memory target), which reads the value stored in the specified memory address and then jumps to it. The instruction format is as follows:

jmt n(r1)

where n is the offset in bytes and register $r1$ stores the memory address. Explain if any changes would be required to the existing data and control path of the multi-cycle processor as shown in figure 1. Discuss the steps involved in executing the `jmt` instruction.

Answer: `jmt` has the same format as the `lw` instruction. However, the `rt` field in the instruction is unused because data (jump address) read from memory is put directly into the PC instead of a specified register. The multiplexer connecting to the PC will get the value from the MDR as an extra input. The select signal/PCSource will be modified to select this input whenever a `jmt` instruction is encountered.

The first four cycles will be the same as `lw`. In the final cycle, the modified multiplexer selects the MDR output with PCSource=11 and loads it into the PC register with PCWrite=1.

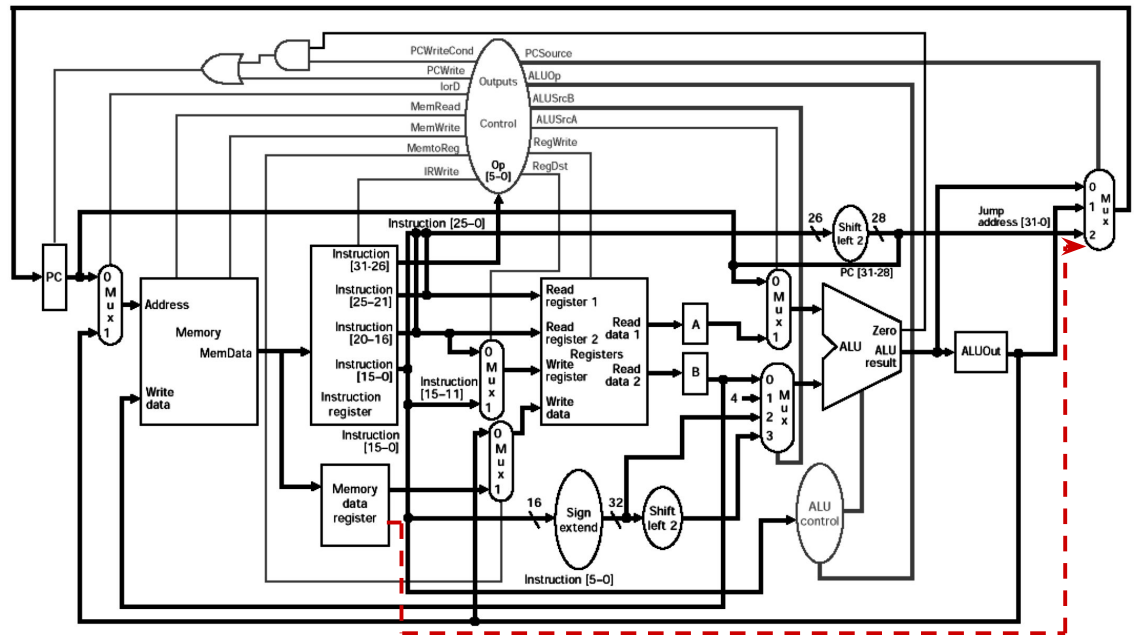


Figure 2: MIPS Datapath for `jmt`

3. Direct-mapped cache

Consider a simple direct-mapped cache with 4 lines, each containing 2 bytes of data and a tag field of appropriate size for 6-bit addresses. Assume that the cache is initially empty and that memory is byte-addressable.

For the following pattern of cache accesses:

Access Number	Address
1	1 0 1 1 0 0
2	0 0 1 0 1 1
3	0 1 1 0 0 1
4	0 1 0 0 0 1
5	1 0 1 1 0 0
6	0 1 1 0 0 1
7	1 0 0 1 1 0

draw up a table of form

Access Number	Tag	Index	Line 0	Line 1	Line 2	Line 3	Hit/Miss
1							
2							
3							
4							
5							
6							
7							

that summarises the contents of the cache *after* each access. In the *Tag* and *Index* columns, write the decimal values of the tag and index components of each access address. Then, in each row, if Line i is occupied with the memory block with tag t , write the decimal value of t in the column for Line i , and record in the Hit/Miss column an H or an M to indicate whether the access is a hit or a miss. If a line is empty after some access, leave the entry for that line and access number blank.

Answer: Each 6-bit address splits into a 3-bit tag, 2-bit index and 1-bit offset. The desired table is

Access Number	Tag	Index	Line 0	Line 1	Line 2	Line 3	Hit/Miss
1	5	2			5		M
2	1	1		1	5		M
3	3	0	3	1	5		M
4	2	0	2	1	5		M
5	5	2	2	1	5		H
6	3	0	3	1	5		M
7	4	3	3	1	5	4	M

Note how line replacement is insensitive to how recently a line has been accessed. E.g. access 4 removes the line accessed immediately before it.