Program 3

Here’s where we implement pipelining and have to deal with the hazards thereof. Since the processor does many operations at the same time (electrons are always flowing through the transistors), pipeline registers and multiplexors are used to maintain order and produce predictable results. Writing a simulation of the processor can be a bit daunting because a computer program works in serial while the circuits on the processor all work in parallel. In other words, the five stages of a basic MIPS processor are operating simultaneously on the chip, while they have to operate in some order in your computer program.

Instructions proceed in stages from left to right: Instruction fetch – instruction decode – execute – memory access – writeback. Nevertheless, a sensible way to write a simulation is to process the stages in reverse order. You would write five methods, one for each stage in the pipeline, but call them in the order Writeback-Memory-Execute-Decode-Fetch. Of course, for the first several cycles, stages at the end of the pipeline wouldn’t be doing anything useful, but once an instruction has migrated from fetch through the rest of the pipeline, each stage will be doing something each time it is called. At the end of the program, of course, stages on the left will stop doing anything useful and the last stage that does anything significant would be the writeback stage.

Pseuocode for this might look like:

int nCycle = 0;

while (not done)

{

Writeback();

Memory();

Execute();

Decode();

Fetch();

nCycle++;

}

Pipeline registers are very important in synchronizing the operation of the stages. They hold the initial state for a stage, and they record the final state when a stage is done operations. One of the reasons we call our stages “backwards” is so that the destination pipeline register can be written following a stage’s execution without overwriting the initial conditions of another stage that hasn’t executed yet this cycle. Of course, what information you persist in the pipeline register is up to you and is part of understanding how the processor works.

Useful in this discussion will be the multiplexor settings for each instruction—which are the same as in Program 2.

|  |  |  |  |
| --- | --- | --- | --- |
| **Mux** | **Produced** | **Used** | **Comments** |
| Jump | ID | EX | 1 = this is a j instruction; 0 otherwise |
| Reg Write | ID | ID | 1 = this instruction writes back to the register file |
| ALU Src | ID | EX | 0 = Register read data B, 1 = sign extender |
| Forward A | EX | EX | 0 = A data from ID/EX.Rs, 1 = data from MEM/WB, 2 = data from EX/MEM |
| Forward B | EX | EX | 0 = B data from ALUSrc, 1 = data from MEM/WB, 2 = data from EX/MEM |
| Mem Read | ID | MEM | 1 = load instruction |
| Mem Write | ID | MEM | 1 = store instruction |
| WB Sel | ID | WB | 1 = write back from ALU; 0 = write back from memory |
| Rd/Rt Sel | ID | ID | 0 = R-type instruction; 1 = not R-type |
| BEQ Branch | ID | EX | 1 = beq instruction |
| ALU Control | ID | EX | 0 = add; 1 = subtract; 2 = load upper; 3 = or |

The following diagram shows an overall view of the processor circuits and the execution process. This diagram is not as detailed as some which will follow, but it gives an overall view and shows the use of the pipeline registers.

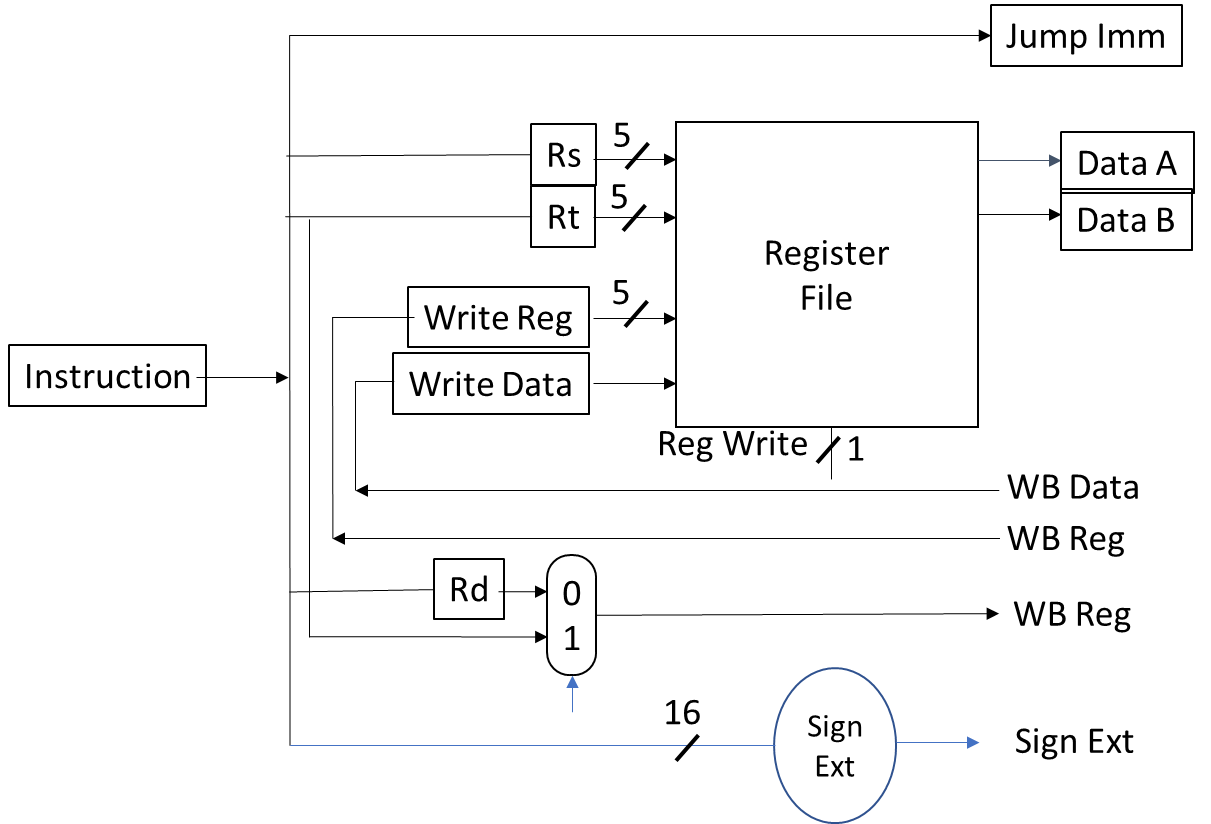


From Patterson & Hennessey, 5ed, p 308 & ff.

Let’s describe what each stage does.

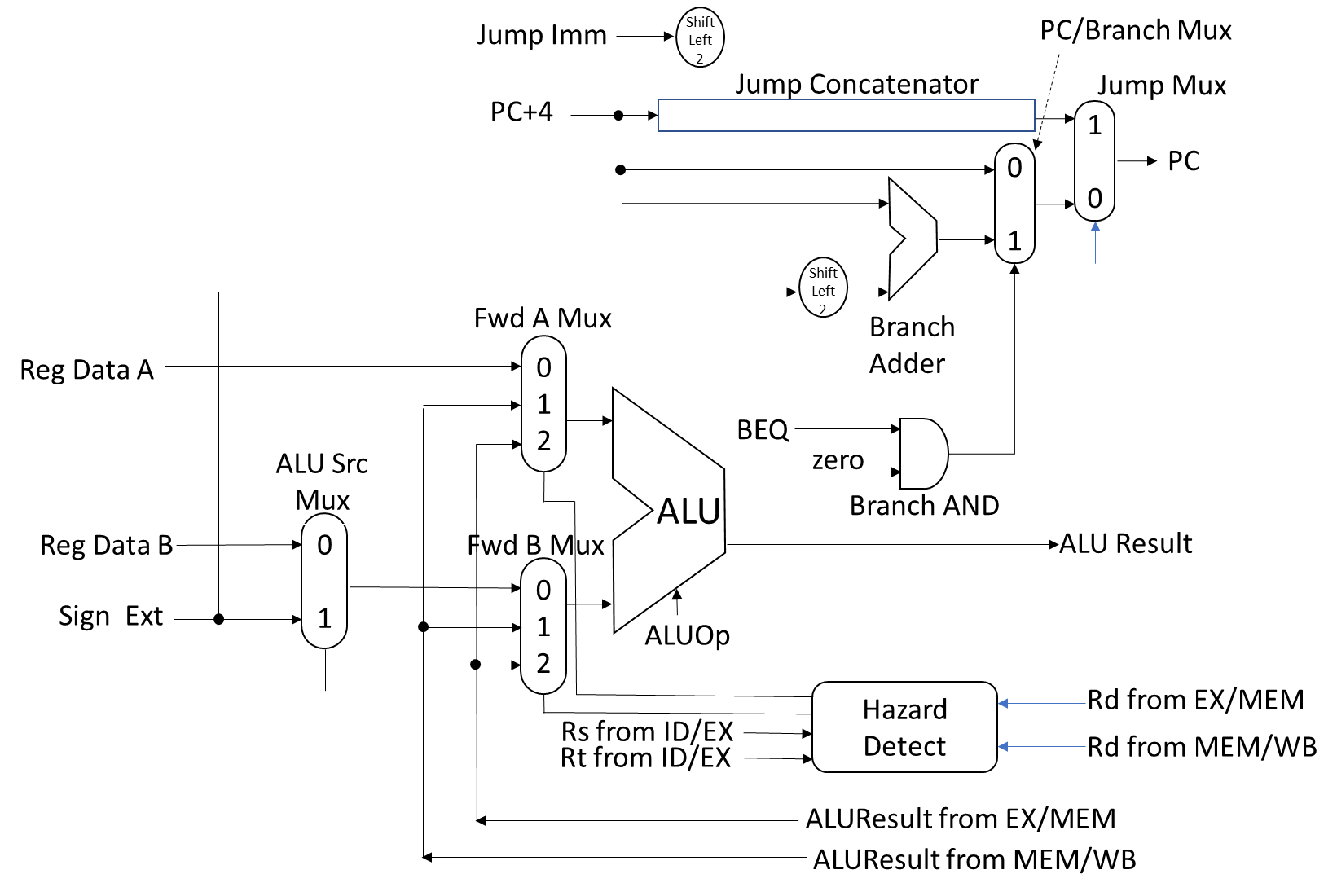
**Instruction Fetch**

This is probably the simplest stage. Here you would take the contents of the Program Counter, obtain the instruction at that location, write that instruction to the IF/ID pipeline register and increment the PC by four.

**Instruction Decode**

In this stage we take apart the instruction as recorded in the IF/ID pipeline register the way we did in Program 2 and write results to the ID/EX pipeline register. Additionally, we access the register file and obtain the values of Rs and Rt, writing those to the ID/EX pipeline register as well. When this phase is done, all of the information we need to process this instruction in subsequent stages should have been written to the ID/EX pipeline register.

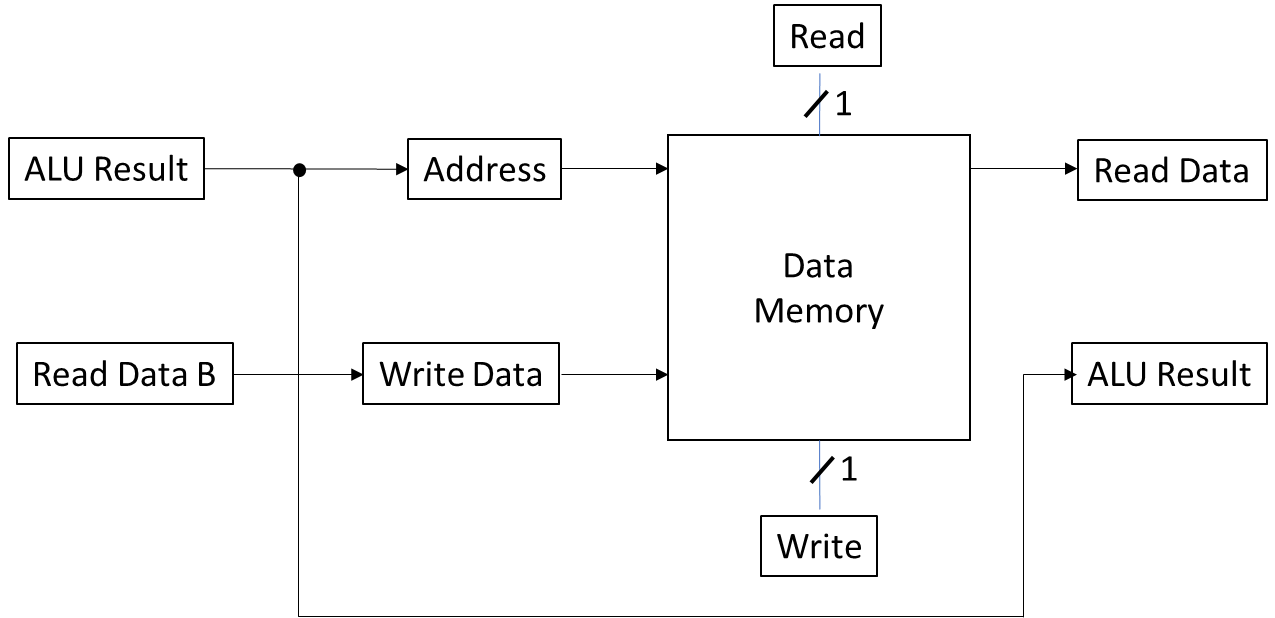
**Execute**

 This stage is the most complicated. In general, the purpose of this stage is to operate the ALU, but in doing so, a lot of other things happen. In order to obtain the correct inputs for the ALU, we need to know what the mux settings are for the ALUSrc mux and for the ForwardA and ForwardB muxes. The ALUSrc mux signal should have been written to the ID/EX pipeline register during the ID stage. The signals for the ForwardA and ForwardB muxes are computed in the EX stage by examining the relationships between the Rs and Rt of the current instruction and the Rd of the instructions in the next two (Memory, Writeback) stages. Inputs for the ForwardA and ForwardB muxes come from three different pipeline registers.

In the execute stage we also write to the PC if we are processing a branch instruction and the branch is taken. So if this is a BEQ instruction and the ALU zero line is true, the branch target (computed in this stage) is written to the PC. If this happens, of course, you will have to take care of flushing the pipeline (what was in the IF and ID stages) because even though we started executing those instruction, we can’t complete them because the program took a different direction.

In the case of a jump instruction, the jump address appears on input 1 of the jump mux. The selector for that mux sends the jump address back to the PC. As is with the case of the branch, if the jump address is written to the PC then the pipeline must be flushed and fetching must restart with the new PC.

**Memory Access**

 If we’re processing a load or store instruction, we read or write values from or to a memory object we’re simulating. Otherwise we transfer the ALU output value from the EX/MEM pipeline register to the MEM/WB pipeline register. This stage is not very complicated. For the purpose of this program, there is only one value read from memory. Set that value to 20 (0x14).

**Writeback**

Here we use the WBMux signal which should have been passed stage-to-stage to us through the pipeline registers to select either the Data memory output or the ALU output to be written to the Register File. The register file is updated, if necessary.

The main execution loop would simply cycle through the sequence Writeback(), Memory(), Execute(), Decode(), Fetch() until some sort of termination signal occurs.

**Hazard Determination.**

Data hazards occur in the Execute stage, and are based on the result registers of the previous two instructions. A way of expressing hazard determination is as follows:

EX stage Hazard:

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

MEM stage Hazard:

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and (MEM/WB.RegisterRd = 1D/EX.RegisterRt)) ForwardB = 01

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd != ID/EX.RegisterRs)) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd != ID/EX.RegisterRt)) and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

For reference, here is the program you will be running:

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | **Instruction** | **Mnemonic** | **Arguments** |
| 0x00400000 | 0x3c011001 | lui | $at, 0x1001 |
| 0x00400004 | 0x34300000 | ori | $s0, $at, 0x0000 |
| 0x00400008 | 0x8e080000 | lw | $t0, 0($s0) |
| 0x0040000c | 0x20090003 | addi | $t1, $zero, 3 |
| 0x00400010 | 0x11200004 | beq | $t1, $zero, 0x0004 |
| 0x00400014 | 0x01094020 | add | $t0, $t0, $t1 |
| 0x00400018 | 0x2129ffff | addi | $t1, $t1, -1 |
| 0x0040001c | 0xae080000 | sw | $t0, 0($s0) |
| 0x00400020 | 0x08100004 | j | 0x00400010 |
| 0x00400024 | 0x2002000a | addi | $v0, $zero, 10 |
| 0x00400028 | 0x0000000c | syscall |  |

Hazards are shown in the highlighted colors. While he ForwardA and ForwardB Mux signals can be computed before the EX stage, the values that may be forwarded are not available until the branch instruction is in the EX stage.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction | Op | Fct | Rd | Rs | Rt | Jump | Reg Write | ALU Src | Fwd A | Fwd B | Mem Rd | Mem Wr | WB Sel | Rd/Rt Sel | beq | ALU |
| add | 000000 | 100000 | 2 | 0 | 1 | 0 | 1 | 0 | \* | \* | 0 | 0 | 1 | 0 | 0 | 0 |
| addi | 001000 | - | 0 | 1 | 16 | 0 | 1 | 1 | \* | \* | 0 | 0 | 1 | 1 | 0 | 0 |
| beq | 000100 | - | 0 | 16 | 8 | 0 | 0 | 0 | \* | \* | 0 | 0 | - | 1 | 1 | 1 |
| j | 000010 | - | 0 | 0 | 9 | 1 | 0 | - | \* | \* | 0 | 0 | - | - | 0 | - |
| lui | 001111 | - | 0 | 9 | 0 | 0 | 1 | 1 | \* | \* | 0 | 0 | 1 | 1 | 0 | 2 |
| lw | 100011 | - | 8 | 8 | 9 | 0 | 1 | 1 | \* | \* | 1 | 0 | 0 | 1 | 0 | 1 |
| ori | 001101 | - | 31 | 9 | 9 | 0 | 1 | 1 | \* | \* | 0 | 0 | 1 | 1 | 0 | 3 |
| sw | 101011 | - | 0 | 16 | 8 | 0 | 0 | 1 | \* | \* | 0 | 1 | - | - | 0 | 1 |
| syscall | 000000 | 001100 | 0 | 0 | 16 | 0 | 0 | - | \* | \* | 0 | 0 | - | - | 0 | - |

\*Depends on hazard detection

For each cycle, the table below shows which instruction is being processed. This may be useful in debugging your program.

† jump logic can be move to the ID stage to avoid a control stall. If it remains in the EX stage, a pipeline flush must occur.

†† replaced by Addi $v0, $zero, 10 when Beq branch is taken in the EX stage

‡fetched and then flushed when BEQ executes

**Output:**

The output of your program should be capable of matching the following table. The best way to do this is to put an output statement (i.e., write to a text file) in each of your five subroutines (Fetch, Decode, etc) showing the status of that stage in that cycle. Your output should contain the same information as in the table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Cy** | **IF** | **ED** | **EX** | **MEM** | **WB** |
| 0 | **lui $at, 0x1001**  read from 0x00400000 |  |  |  |  |
| 1 | **ori $s0, $at, 0**  read from 0x00400004 | **lui $at, 0x1001**  DataA = 0x0  DataB = 0x0  Imm = 4097 |  |  |  |
| 2 | **lw $t0, 0($s0)**  read from 0x00400008 | **ori $s0, $at, 0**  DataA = 0x0  DataB = 0x0  Imm = 0 | **lui $at, 0x1001**  ALUResult = 0x10010000  RDest=$at |  |  |
| 3 | **addi $t1, $0, 3**  read from 0x0040000c | **lw $t0, 0($s0)**  DataA = 0x0  DataB = 0x0  Imm = 0 | **ori $s0, $at, 0**  ForwardA forwarding 0x10010000 from MEM\_WB  ALUResult = 10010000  RDest=$s0 |  |  |
| 4 | **beq $t0, $0, 4**  read from 0x00400010 | **addi $t1, $0, 3**  DataA = 0x0  DataB = 0x0  Imm = 3 | **lw $t0, 0($s0)**  ForwardA forwarding 0x10010000 from MEM\_WB  ALUResult = 10010000 RDest=$t0 |  | **lui $at, 0x1001**  Result 0x10010000 written to $at |
| 5 | **add $t0, $t0, $t1**  read from 0x00400014 | **beq $t1, $0, 4**  DataA = 0x0  DataB = 0x0  Imm = 4 | **addi $t1, $0, 3**  ALUResult = 3  RDest=$t1 | **lw $t0, 0($s0)**  Read 0x14 | **ori $s0, $at, 0**  Result 0x10010000 written to $s0 |
| 6 | **addi $t1, $t1, -1**  read from 0x00400018 | **add $t0, $t0, $t1**  DataA = 0x14  DataB = 0x0  Imm = 16416 | **beq $t1, $0, 4**  ForwardA forwarding 0x3 from MEM\_WB  ALUResult = 3  Rdest=$zero |  | **lw $t0, 0($s0)**  Result 0x14 written to $t0 |
| 7 | **sw $t0, 0($s0)**  read from 0x0040001c | **addi $t1, $t1, -1**  DataA = 0x3  DataB = 0x3  Imm = -1 | **add $t0, $t0, $t1** ALUResult = 0x17  Rdest=$t0 |  | **addi $t1, $0, 3**  Result 0x3 written to $t1 |
| 8 | **j 0x0040010**  read from 0x00400020 | **sw $t0, 0($s0)**  DataA = 0x10010000  DataB = 0x14  Imm = 0 | **addi $t1, $t1, -1** ALUResult = 2  Rdest=$t1 |  |  |
| 9 | **beq $t1, $0, 4**  read from 0x00400010 | **j 0x0040010**  DataA = 0x0  DataB = 0x10010000  Imm = 4 | **sw $t0, 0($s0)**  ALUResult = 10010000  Rdest=$zero |  | **add $t0, $t0, $t1**  Result 0x17 written to $t0 |
| 10 | **add $t0, $t0, $t1**  read from 0x00400014 | **beq $t1, $0, 4**  DataA = 0x2  DataB = 0x0  Imm = 4 | **j 0x0040010**  ALUResult = 0  Rdest=$zero | **sw $t0, 0($s0)**  Write 0x17 | **addi $t1, $t1, -1**  Result 0x2 written to $t1 |
| 11 | **addi $t1, $t1, -1**  read from 0x00400018 | **add $t0, $t0, $t1**  DataA = 0x14  DataB = 0x2  Imm = 16416 | **beq $t1, $0, 4**  ALUResult = 2  Rdest=$zero |  |  |
| 12 | **sw $t0, 0($s0)**  read from 0x0040001c | **addi $t1, $t1, -1**  DataA = 0x2  DataB = 0x2  Imm = -1 | **add $t0, $t0, $t1**  ALUResult = 0x19  Rdest=$t0 |  |  |
| 13 | **j 0x0040010**  read from 0x00400020 | **sw $t0, 0($s0)**  DataA = 0x10010000  DataB = 0x14  Imm = 0 | **addi $t1, $t1, -1**  ALUResult = 1  Rdest=$t1 |  |  |
| 14 | **beq $t1, $0, 4**  read from 0x00400010 | **j 0x0040010**  DataA = 0x0  DataB = 0x10010000  Imm = 4 | **sw $t0, 0($s0)**  ALUResult = 10010000  Rdest=$zero |  | **add $t0, $t0, $t1**  Result 0x19 written to $t0 |
| 15 | **add $t0, $t0, $t1**  read from 0x00400014 | **beq $t1, $0, 4**  DataA = 0x1  DataB = 0x0  Imm = 4 | **j 0x0040010**  ALUResult = 0  Rdest=$zero | **sw $t0, 0($s0)**  Write 0x19 | **addi $t1, $t1, -1**  Result 0x1 written to $t1 |
| 16 | **addi $t1, $t1, -1**  read from 0x00400018 | **add $t0, $t0, $t1**  DataA = 0x16  DataB = 0x1  Imm = 16416 | **beq $t1, $0, 4**  ALUResult = 1  Rdest=$zero |  |  |
| 17 | **sw $t0, 0($s0)**  read from 0x0040001c | **addi $t1, $t1, -1**  DataA = 0x1  DataB = 0x1  Imm = -1 | **add $t0, $t0, $t1**  ALUResult = 0x1A  Rdest=$t0 |  |  |
| 18 | **j 0x0040010**  read from 0x00400020 | **sw $t0, 0($s0)**  DataA = 0x10010000  DataB = 0x16  Imm = 0 | **addi $t1, $t1, -1**  ALUResult = 0  Rdest=$t1 |  |  |
| 19 | **beq $t1, $0, 4**  read from 0x00400010 | **j 0x0040010**  DataA = 0x0  DataB = 0x10010000  Imm = 4 | **sw $t0, 0($s0)**  ALUResult = 10010000  Rdest=$zero |  | **add $t0, $t0, $t1**  Result 0x1A written to $t0 |
| 20 | **add $t0, $t0, $t1**  read from 0x00400014 | **beq $t1, $0, 4**  DataA = 0x0  DataB = 0x0  Imm = 4 | **j 0x0040010**  ALUResult = 0  Rdest=$zero | **sw $t0, 0($s0)**  Write 0x1A | **addi $t1, $t1, -1**  Result 0x0 written to $t1 |
| 21 | **addi $v0, $0, 10**  read from 0x00400024 |  |  |  |  |
| 22 | **syscall**  read from 0x00400028 | **addi $v0, $0, 10**  DataA = 0x0  DataB = 0x0  Imm = 10 | **beq $t1, $0, 4**  ALUResult = 0  Rdest=$zero |  |  |
| 23 | **00000000**  read from 0x0040002c | **syscall**  DataA = 0x0  DataB = 0x0  Imm = 12 | **addi $v0, $0, 10**  ALUResult = A  Rdest=$v0 |  |  |
| 24 | **00000000**  read from 0x00400030 | **00000000**  DataA = 0x0  DataB = 0x0  Imm = 0 | **syscall**  ALUResult = 0  Rdest=$zero |  |  |
| 25 | **00000000**  read from 0x00400034 | **00000000**  DataA = 0x0  DataB = 0x0  Imm = 0 | **00000000**  ALUResult = 0  Rdest=$zero |  | **addi $v0, $0, 10**  Result 0xa written to $v0 |

Note that in cycle 20, the add instruction is flushed, the PC is changed and the branch target instruction is fetched. The “lost” instruction (the add) is what the book refers to as the “branch delay slot” instruction.