Exercises 4
Processor Design

Computer Organization and Components / Datorteknik och komponenter (IS1500), 9 hp
Computer Hardware Engineering / Datorteknik, grundkurs (IS1200), 7.5 hp

KTH Royal Institute of Technology
December 18, 2020
Suggested Solutions

1. (a) Signals A and B are input signals to the ALU and signal F determines the operation that the ALU should perform. Note that F is a 3-bit signal, where the most significant bit F_2 is separated from the two least significant bits $F_{1:0}$. The Y signal is the output from the ALU.

   (b) By implementing several functions and reusing components for several different operations, a well-designed ALU can save hardware resources compared to if each function was implemented as a separate component.

   (c) In binary form, $F = 101_2$ which means that $F_2 = 1_2$ and $F_{1:0} = 01_2$. Hence, the *4:1* multiplexer on the right hand side selects input signal 1 that comes from the OR-gate. We can now observe that one of the inputs to the OR-gate is signal A. As a consequence, Y is always $\texttt{ff}_{16}$, regardless of the value of B.

   (d) We assume that signed numbers are represented using two's complement representation. If so, the adder component needs to have a carry-in value of 1, which means that $F_2 = 1$. Signal B needs to be inverted (recall the algorithm for negating a value in two's complement form) and then incremented with one (the carry in signal). Hence, the left-most multiplexer needs to select the inverted signal. Again, this means that $F_2 = 1$, so we can see that this holds for both places where F_2 are used. The output signal Y should be the value directly coming from the adder component. Hence, $F_{1:0} = 2$. Consequently, $F = 110_2 = 6$.

   (e) We can directly see that the right-hand side multiplexer is selecting input signal 3. Since $F_2 = 1$, we have the same functionality as in the previous exercise: subtraction, with the only difference that input signal 3 instead of 2 is selected at the right-hand side multiplexer. By using expression $[N-1]$, the highest bit from the result of $A - B$ is selected. For a two's complement number, this is the sign bit that indicates if the result is positive or negative. As a consequence, the operation that is performed is the same as the MIPS instruction "set less than", `slt`.

   (f) We can see that $F_{1:0} = 01_2$, which means that the OR-gate is used. Because $F_2 = 1$, signal B is inverted before it reaches the OR-gate. Hence, the performed operation is $A|!B$, where $|$ means OR and ! means NOT, as in the programming language C. Since $N = 4$ and $B = \texttt{C}_{16} = 1100_2$, the bitwise negated value of B is 0011_2. We can also see that $A = 10_{10} = 1010_2$. Then, after performing the OR operation, $Y = 1011_2 = 11_{10} = \texttt{B}_{16}$.

2. (a) Consider the MIPS reference sheet (can be downloaded from the course website). We can see that the `add` instruction is an R-type instruction. From the instruction encoding, we see that bits 25:21 corresponds to the `rs` field. The `rs` field is the first source operand, which in our case is the `$s0` register. From the MIPS reference sheet we can see that `$s0` has encoding number 16. Hence, the *A1* signal has value 16. The value for the *A2* signal can be found in a similar way: bits 20:16 correspond to the `rt` field, which in our case is register `$s1`. Hence, signal *A2* has value 17. Finally, for *A3*, we need to determine if *RegDst* is 0 or 1. Since we know that the `add` instruction is an R-type, the destination register is located in bits 15:11. Hence, *RegDst* must be one 1, so that the multiplexer is selecting the correct signal. Finally, the destination register is in our case `$t3`, which means that *A3* has value 11.

   (b) Each of the control signals can be derived by reasoning about the logic of the datapath. The *Jump* is used for J-type instructions. In this case, load word is an I-type instruction, that will not jump. Hence, the instruction should only increment the program counter (PC) with 4. As a consequence, both the multiplexers to the left in the figure should have their control signal values 0 and therefore both *Jump* and *Branch* should be 0 in this case.

   The load word instruction is reading from the data memory and then writing to the register file. Hence, the write enable port *WE3* of register file must be 1, resulting in that *RegWrite* must be 1.

   The *RegDst* control signal determines from which bits the destination register should be decoded from in the instruction. In this case, the load word instruction is an I-type instruction. By inspecting the MIPS reference sheet, we can see that the destination register is in the `rt` field, which is located in bits 20:16. Hence, *RegDst* must be 0 so that the multiplexer can select the correct bits from the instruction.

   For the load word instruction, the immediate value -4 should be added to the address that is read out from the source register (in this case `$s4`). Output signal *RD1* will contain the address in `$s4`, that is, value `0x0000ff08`. Hence, if we add value -4 we get address `0x0000ff04`, which is input address signal A of the data memory. The *ALUSrc* must have value 1 so that immediate value of the instruction is read (bits 15:0 of an I-type instruction) and then sign extended. The ALU should perform an addition. Consider again the ALU in exercise 1. Signal F_2 needs to be 0 so that no subtraction is performed and $F_{1:0}$ needs to have value 2. Hence, the *ALUControl* signal should be $\texttt{010}_2$.

   The load word instruction is writing to the register file, but not to the data memory. Control signal *MemWrite* should therefore be 0. However, the load word signal is reading from the data memory. Consequently, control signal *MemToReg* should be 1. Finally, since the instruction is loading from the data memory, but not writing to it, it does not matter what the value of the write data signal *WD* is.

3. First, we need to determine which instruction that is executed. The most significant byte is `0x21` $=$ $\texttt{00100001}_2$. The six most significant bits are $\texttt{001000}_2 = 8$. If we look it up in the MIPS reference sheet, we see that it must be instruction add immediate, `addi`. We can also observe that this is an I-type instruction.

   Our next step is to decode the instruction, to see what the first source register `rs` is. This register is located at bits 25:21 of the instruction. We have already converted the most significant byte of the instruction to bits, and proceeds therefore with the next byte `0x4b` $=$ $\texttt{01001011}_2$. Hence, the five bit `rs`-field has value $\texttt{01010}_2 = 10_{10}$, corresponding to register `$t2`. From the register values in the exercise, we see that `$t2` $=$ `0xfffffff5`. This is a negative number represented using two's complement. Hence, we can get the corresponding positive value by negating the bits and adding the value with one. It is enough to just look at the least significant byte because the rest of the bits are all set to 1. That this, `0xf5` $=$ `11110101`, which after flipping the bits and adding with one is `00001011` $= 11_{10}$. Hence, register `$t2` contains value -11_{10}.

   Next, we need to compute the immediate value. For the I-type, the immediate value is located in the least significant 16 bits. The actual value that is added is the sign extended version of this value. For this instruction, the 16-bit immediate value is `0xfffd`. This is a negative value, with the signed bit set to 1. Hence, the sign-extended 32-bit value is `0xfffffffd`. By using the procedure above, we can see that this value correspond to the decimal value -3.

   When the instruction is executed, the resulting value that should be stored back in the register file is $-11 + (-3) = -14_{10}$. Converting this value to its two's complement representation is done as usual by taking the positive number, flipping all bits and add 1. We can do this first by using an 8-bit number, and then sign extended it to 32 bits (since we know that -14 fits in a byte). First, 14_{10} $=$ `00001110`. After bit-flipping we have `11110001`, and after adding by one, we get `11110010` $=$ `0xf2`. Hence, the resulting 32-bit value is `0xfffffff2`, which is the write data input *WD3* that was asked for in the exercise.

4. (a) False. $RegWrite$ does not have to be zero if $MemToReg$ is zero. For instance, for an `add` instruction, $MemToReg$ is zero because the value computed in the ALU bypasses the data memory. In this case, $RegWrite$ needs to be 1, so that the result can be written back to the register file.

   (b) True. When a conditional branch instruction, such as `beq`, is executed, the Zero bit of the ALU indicates if the result of the ALU computation is zero or not. If the $Branch$ control signal and the Zero bit are both 1, then the multiplexer on the left hand side will chose the branch target address that is computed from the immediate value and the program counter. See computation of Branch Target Address (BTA) in the MIPS reference sheet.

   (c) True. If the 6 most significant bits are 0, this means that the instruction is an R-type instruction. In such a case, the destination register is located in bits 15:11, which means that $RegDst$ must be 1. If the 6 most significant bits indicate that it is an I-type instruction, $RegDst$ must be 0. If it is an J-type instruction, the value of $RegDst$ is not relevant.

(d) False. If the instruction is an R-type instruction, the 6 least significant bits is the `funct` field, which affects the *ALUControl* signal. However, these are not the only bits that affects the *ALUControl* signal. For instance, consider the two I-type instructions `ori` and `addi`. They have both different *ALUControl* signal values (for OR and ADD operations, respectively). The 6 least significant bits for these two instructions are instead part of the immediate value.

5. (a) False. A pipelined processor can reduce timing delays in the circuit that may make it possible to clock the processor at a higher clock frequency. However, the pipeline may introduce pipeline hazards and the processor may need to introduce pipeline stalls. Thus, each instruction can take more than one cycle, thus giving a higher CPI compared to a single-cycle processor (not lower).

   (b) True. If the critical path (the longest delay in the circuit) can be shorter, the processor can be clocked at a higher clock frequency. Higher clock frequency means that more clock cycles are executed each second. Note, however, that the statement says "can get better performance". If the number of clock cycles per instruction increases as well, it does not necessarily mean that we improve the performance.

   (c) False. Many modern processors have pipelines with more than 10 stages. More pipeline stages can enable higher clock frequencies. However, higher clock frequencies may affect the energy (the processor may not be possible to run on low voltage). Designing an optimal design is extremely hard, if even possible.

   (d) True. This is usually one of the main things that happens in the decode stage.

6. (a) By assuming that the comparison is done by the ALU, the comparison is in fact done in the execute (E) stage. However, if the branch is taken, the program counter (PC) is updated in the memory (M) stage. An alternative would be to update the PC also in the execute stage, but this would most likely significantly increase the critical path of the processor. The `beq` instruction is using the value in register `$s0`, which is written by the `add` instruction. The resulting value for the `add` instruction is written to the register file in the writeback (W) stage, but is needed by `beq` instruction in the decode (D) stage (needs to read the value from the register file). This is an example of a data hazard and also an example of a read-after-write (RAW) hazard. This specific hazard can be solved by forwarding. After the execute (E) state, the `add` instruction has the result for `$s0` ready. The value can then be directly forwarded, so that it becomes available in the execute (E) stage of the `beq` instruction.

   (b) Yes, the load word `lw` instruction, followed by the `xor` instruction, results in a data hazard because `lw` writes the result in `$s0` and the `xor` instruction uses `$s0` as one of its operands. However, in this case, it is not possible to solve the hazard using forwarding. The result of the `lw` instruction is not ready until at the end of the memory (M) stage, but the result is needed in the beginning of the execute (E) stage for the `xor` instruction. Hence, the hazard must be solved by using stalling, that is, by inserting "bubbles" in the pipeline and delay the computation one clock cycle. By delaying the pipeline one clock cycle, we can now use forwarding so that the `xor` instruction can get the result of `$s0` after that it is read from memory. Note that this example assumes that the memory can be read in one clock cycle, which is not true for computers with larger memories. In such a case, the memory read operation can take multiple cycles.

(c) Since the processor assumes branch-not-taken, there will be no penalty (no bubbles) if the branch is not taken, i.e., in this case if `$s0` and `$s1` are not equal. In such a case, the next instruction is fetched in the normal way. However, if the branch is taken, the pipeline must be flushed, and there will be branch misprediction penalty of 3 cycles. The processor can know that the branch should be taken after the execute stage, that is, it updates the PC in the memory (M) stage, which means that 3 instructions have to be flushed. This kind of hazard is called a control hazard, since it is a result of changing the control of the program (updating the program counter (PC)). The hazard is solved by stalling.

(d) By moving the comparison to the decode stage, the branch misprediction penalty for taking the branch is reduced to one cycle. Since the PC can be updated at the end of the decode stage, only the next coming instruction (`lw` in this case) needs to be flushed. This control hazard is still solved using stalling, but fewer cycles needs to be stalled. Note, however, that by moving the comparison to the decode stage, we need to have the information that we are comparing one cycle earlier. In this case, when we have a data hazard from the `add` instruction, it means that the `beq` instruction needs to be stalled one cycle, so that the result is ready in the decode stage (by forwarding) from the execute stage from the `add` instruction. Hence, by moving the comparison to the decode stage, we introduce (in this case) a stall when we execute the `beq` instruction. However, we reduce the branch misprediction penalty for branch taken from 3 to 1 cycle.