Integrated System Architecture
Lab session 3 report - RISC-V special project

Marco Andorno (247222)
Michele Caon (253027)
Alessio Colucci (xxxxxx)
Matteo Perotti (251453)
Giuseppe Sarda (255648)

March 11, 2019

1 Datapath

1.1 Memory

Both instruction and data memory are implemented using a simple behavioral memory model, with both synchronous read and write operations. According to the specifications of the RV32I, instructions are 32-bit long and the biggest addressable unit is also a 32-bit word, so this data with applies in both cases.

Figure 1 shows the interface of this block, where the `address` is left parametric, as it can differ between instructions and data memory. Read and write operations are handled by the couple of control signals `memRead` and `memWrite`, of which only one should be asserted at each clock cycle to perform the desired action. Both signals active represent an forbidden condition and should be avoided by the whatever is in charge of controlling the memory.

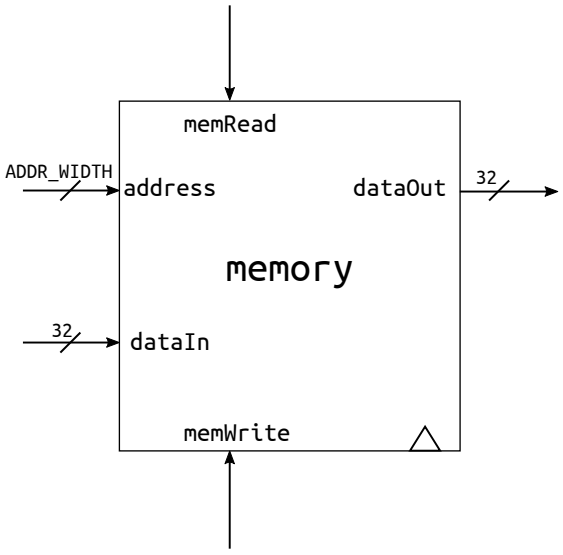

Figure 1: Memory

Figure 2 shows the usual timing diagram of this fully synchronous memory, according to which both reads and writes take place at the next clock cycle after the proper control signals are asserted.

1.2 Register file

The RISC-V register file is composed of 32 registers, each 32-bit wide (for RV32I), called `x0` to `x31`, where `x0` is a special register hardwired to the value 0, which can turn useful for some instructions. Figure 3 shows the top level diagram of the register file structure.

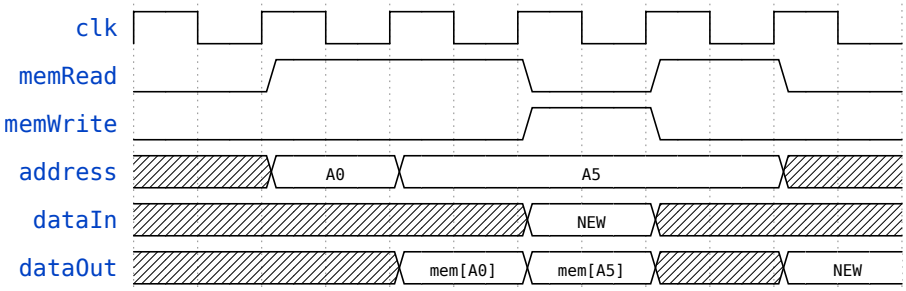

Figure 2: Memory timing diagram

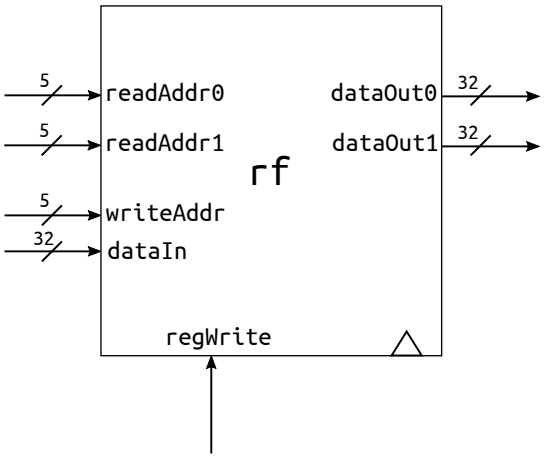

Figure 3: Register file

Writes to the register file are, of course, synchronous and happen on the positive edge of the clock. For a correct write operation, the destination register must be selected using the `writeAddr` port, the input data must be placed on the `dataIn` port and the signal `regWrite` must be asserted. Internally, the register file will enable only the selected register using a decoder.

Reads are instead combinational and can occur on two different registers at a time, thanks to two different read ports. To select the correct output value, a 32-to-1 multiplexer is used on each read port. However, in order to avoid data hazards during the write back stage, the register file also implements bypassing of input data directly to the output if the same register is read and written during the same clock cycle. Figure 4 shows this read selection process (no multiplexer is used to select 0 in case the register being read is `x0` as we can suppose it is hardwired directly at its output).

To better illustrate the behavior of the register file operations, their timing diagram is shown in 5.

1.3 ALU

1.4 Branch and Jump management

A general view of the unit is given in Figure 6.

Types of instructions There are two classes of instructions which can lead to a modification of the sequential flow of the program. In the RV32I ISA they are:

1. Branches
2. Jumps

The former is a conditional change of the usual choice of the next address to be put in the PC, whereas the latter is unconditional. The condition, whenever present, is always based on the result of an ALU comparison.

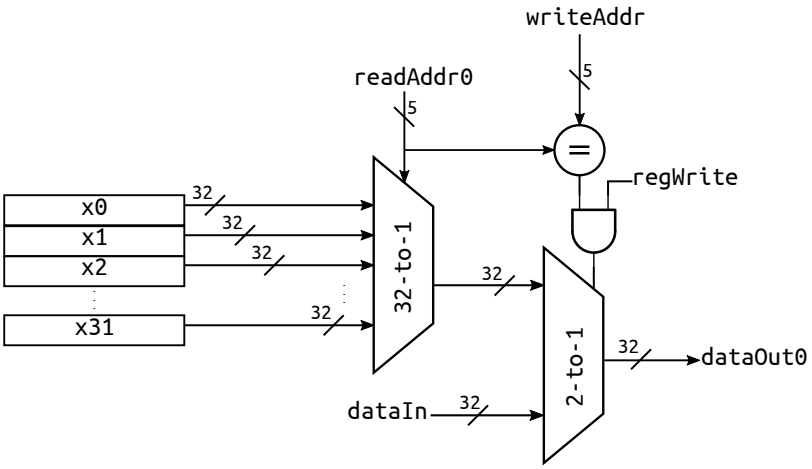

Figure 4: Read operation in the register file

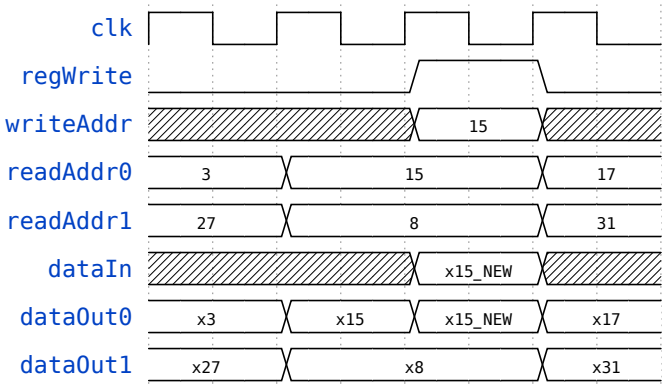

Figure 5: Register file timing

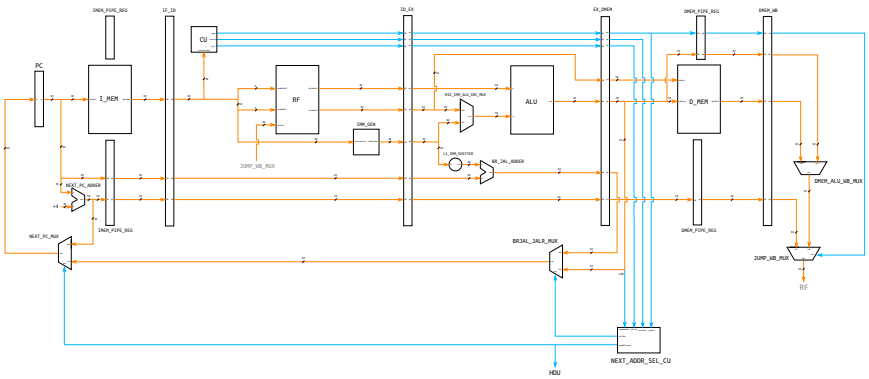

Figure 6: Branch and Jump management HW

The **Branch** instruction exists in different flavours, depending upon which comparison has to be performed between the content of two registers. It follows a list of all of them:

1. BEQ
2. BNE
3. BLT
4. BGE
5. BLTU
6. BGEU

All these instructions belong to the B type one. They have an immediate field split along the word, which indicates an effective immediate divided by two. Indeed, with a RISC-V standard architecture it is possible to address this way an half of a word at most, but never the single byte. The effective immediate is calculated with a bit-reorganization, a sign extension and a left shift of one position, to reach the final 32-bits width. The instruction contains also the addresses of two registers, whose content will be compared by the ALU to decide whether to take the branch or not. To distinguish which type of comparison is needed, it is necessary to know the instruction field func3.

The **Jump** instruction can be of two types, each bringing to a different hardware path for the data:

1. JAL
2. JALR

JAL is a J instruction, whereas **JALR** is an I one. This difference is reflected on the HW implementation: more on this later. A jump instruction is unconditional, but still need for an address computation. This operation is different for the two instructions: for JAL it is sufficient to use the branch-addr-computation hardware, whereas JALR requires the non shifted immediate to be added to the content of a register (the next address is not derived by the current one).

Instruction execution Since so far no **BPU** (**B**ranch **P**rediction **U**nit) is present in the design, a "branch not taken" assumption is always done when the content of the PC is updated and the decoded instruction is a **BR**anch. The simplest way to manage a branch is to delay the decision until the execution stage, waiting for the ALU to do the comparison. The effective decision is then taken in the MEM stage, not to exacerbate a path which can be critical by itself. Also the calculation of the next address, which involves the immediate and the program counter, is performed in the execution stage. A possible improvement could be to anticipate the comparison and the next address calculation in the decoding stage, but to keep the design simple the first solution was chosen. This is compliant with the calculation of the address for a **JAL** execution. It is worth to mention that the absence of a condition to be verified is enaugh to simplify the anticipation of the address calculation and bring it in the decode stage. However this solution would increase the number of resources if the other branch/jump instructions are still executed in an another stage. A **JALR** instruction beheaves in a slightly different way: the address calculation is performed by the ALU, because the immediate is added to the content of a register.

A branch instruction has no side effects once it has been executed. On the contrary, a jump instruction leaves in the pipeline the next instruction address to be saved in a destination register. This is not a issue though, because it is possible to see that even without forwarding units no data hazards could arise. If the pipline was longer, maybe the forwarding unit would be the only thing to have the day saved (the design has it, though).

Effective calculation The address calculation in case of branches/jumps is performed in the execution stage and it depends on the type of instruction:

Branch/JAL It is based on the "current" PC value (current for that precise instruction!). The immediate is sign extended, one position left shifted and added to the PC value (percolated through the pipeline until there) by means of another adder. In the meantime, if the instruction is a JAL, the address of the next instruction goes on through the stages.

JALR It involves a sum between an immediate and the content of a register. The ALU performs this operation without shifting the immediate. When the result has to be used, the LSB is substituted with a zero. Even in this case, the address of the next instruction follows its path towards the write-back stage.

Next address selection CU To control the multiplexers for the next address selection, there's the need for knowing:

1. Whether the instruction in the MEM stage is a branch or a jump.
2. Which is between the two.
3. The result of the comparison.
4. If the instruction is a JALR.

The main CU generates two signals **branch** and **jump** which percolate along the pipeline, to allow the "Next address selection CU" to solve the first two points. The result of a comparison is simply the LSB of the ALU result. The main CU has to generate another signal "jalr" to indicate a JALR instruction.

The "jump" control signal is used also in the writeback stage, to select the right input for the register file. If a jump is performed, the data to be written in the destination register is the "next" address wrt the jump instruction.

In any case, the IMEM pipe register, together with IF/ID, ID/EX and EX/DMEM ones, have to be flushed. This brings to a performance loss of 4 instructions for each taken branch or executed jump.

Next address generation The next address is chosen by means of two multiplexers. The first **BRJAL_JALR_MUX** takes in input the result of the ALU with the LSB masked, and the output of the additional adder of the execution stage. These two input come from the EX/DMEM pipe register. The ouput of **BRJAL_JALR_MUX** is one of the input of the other multiplexer **NEXT_PC_MUX**; the other input is the content of the PC + 4.