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 13, 2019

1 Datapath

1.1 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 1 shows the top level diagram of the register file structure.

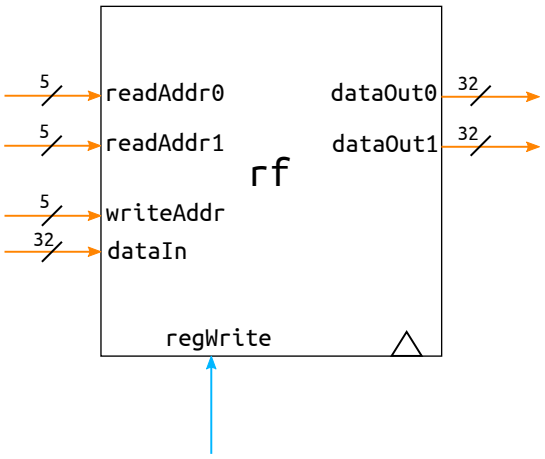

Figure 1: Register file

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 2 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 3.

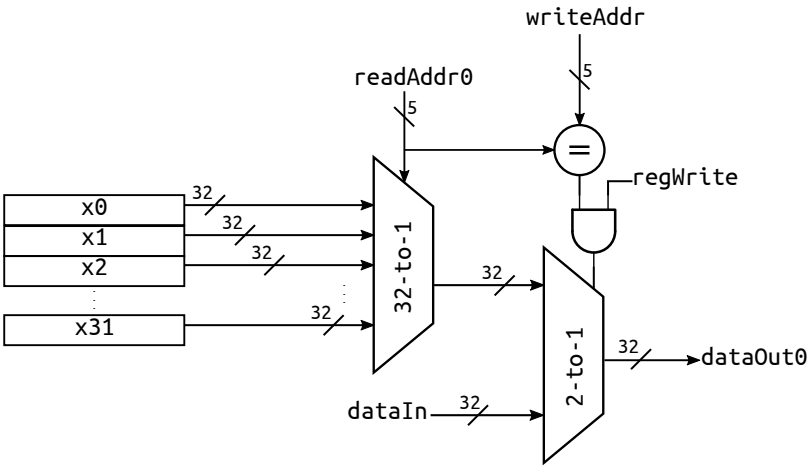

Figure 2: Read operation in the register file

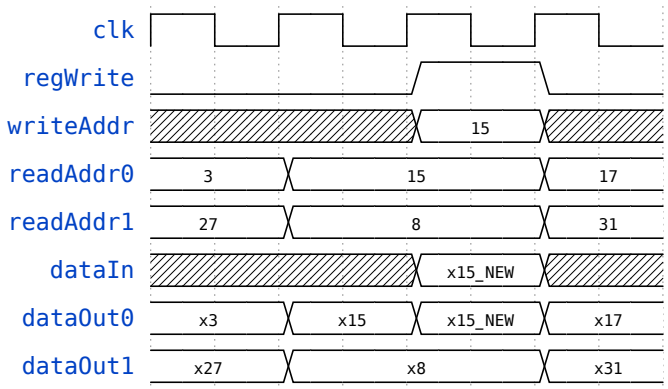

Figure 3: Register file timing

1.2 ALU

The ALU is in charge of performing all operations required by arithmetic and logic instructions, load and store, and branch comparison. Figure 4 shows its top level block diagram, which is simply composed of two inputs and one output on 32 bits, along with a 4-bit control signal to select the desired operation.

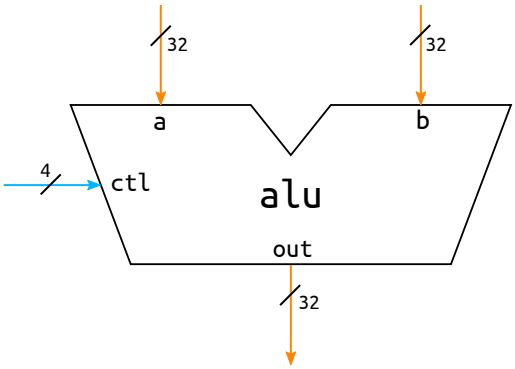

Figure 4: ALU

The complete list of operation that the ALU can perform is the following (in the order in which they are defined on the `ctl` input):

1. Add
2. Subtract
3. AND
4. OR
5. XOR
6. Left shift
7. Right shift
8. Right shift with sign extension
9. Set if equal
10. Set if not equal
11. Set if less than
12. Set if greater or equal than
13. Set if less than unsigned
14. Set if greater or equal than unsigned

Note that all operations were described behaviorally as per specifications, in order to be as implementation independent as possible and open to every optimization that a synthesis tool can perform.

All 'set if *' operations set the output to the value 1 (`0x00000001`) if the condition is true, or 0 otherwise. This approach was chosen instead of using flags (such as Carry, Overflow, Negative and so on) to compute conditions as it was deemed simpler to implement and thorough enough, given that there would not have been other uses for the flags.

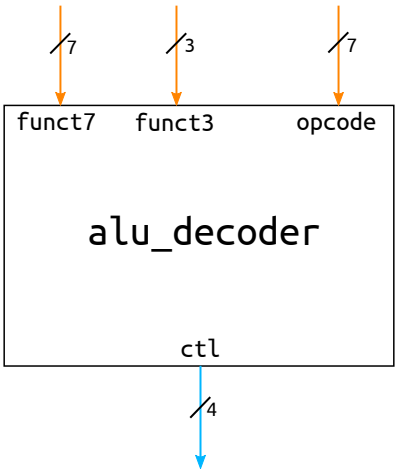

Figure 5: ALU decoder

1.2.1 ALU decoder

The control input of the ALU is generated by a special decoder starting from the opcode, funct3 and funct7 fields of each instruction which requires an ALU operation, as described in figure 5. This block consists only of a series of conditional statements (that can easily be mapped to multiplexers) which select the correct control signal.

Another approach would have been to split the control of the ALU into two decoding steps[1], but a preliminary analysis concluded that no practical advantage would be obtained this way. Moreover, only few bits of the input fields are required to make a definite decision on the control output, but in order to keep modularity and continuity in the design, the whole fields are given in the interface of the block.

1.3 Branch and Jump management

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

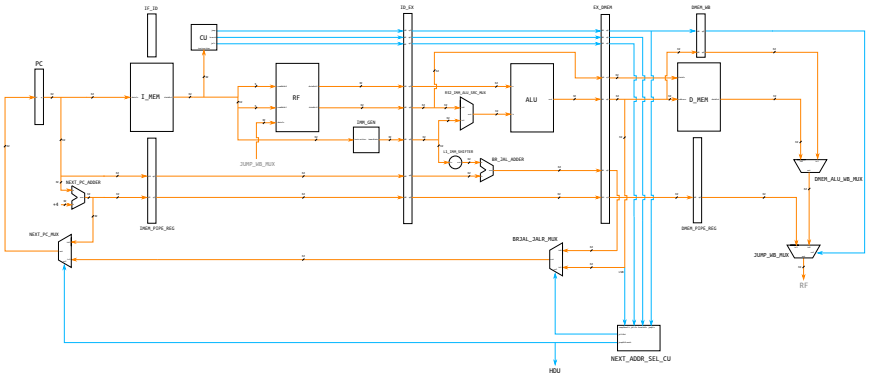

Figure 6: Branch and Jump management HW

[1] As suggested in Chapter 4 of D. Patterson, J. Hennessy, *Computer Organization and Design RISC-V Edition*, Morgan Kaufmann, 2017

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.

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 output 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.

2 Control

2.1 Forwarding Unit (FWU)

Forwarding allows to avoid stalling the pipeline when a data hazard occurs between two subsequent instructions, if the needed data is already available in a following pipe stage. In this five stage pipeline results are written to the destination register during the write back stage, three clock cycles after the operand read in the decode stage. This means that, if an instruction modifies a certain register, only instructions starting from the third after the original would read the correct new data, or equivalently that up to two bubble should be inserted in case of a data hazard.

Forwarding simply bypasses data to the beginning of the execution stage (i.e. at the ALU inputs) if the required results are already present at the ALU output or in the following memory access stage.

To do so, the logic performs some checks:

- Check that the earlier in the pipe actually modifies some register (`regWrite` is asserted) and that the address does not point to register `x0`.

- Compare the destination register in the EX/MEM stage with both source register in the ID/EX stage and, if there is a match, drives the selection signal of the corresponding ALU input multiplexer to select the previous ALU output (`fwdA/fwdB` = 10).
- Otherwise, compare the destination register in the MEM/WB stage with both source register in the ID/EX stage and, if there is a match, drives the selection signal of the corresponding ALU input multiplexer to select the result currently in the memory access stage (`fwdA/fwdB` = 01).

Note that, according to the list above, forwarding gives precedence to data present in the EX/MEM stage over the MEM/WB stage if the same register is present in both, as the former contains the latest result.

2.1.1 Load/store forwarding

The designed forwarding unit handles also the another special case of data hazard that occurs when a load is followed immediately by a store to the same memory location, such as in memory to memory copies.

In this case the FWU checks that the two involved instructions are actually a load (`memToReg` asserted in the MEM/WB stage) and a store (`memWrite` asserted in the EX/MEM stage) and that the destination and source registers are the same, and if that is the case drives the control of the multiplexer selecting the memory data input to choose the memory output.

Figure 7 shows the complete FWU with all inputs on which decisions are taken and the three multiplexer control outputs.

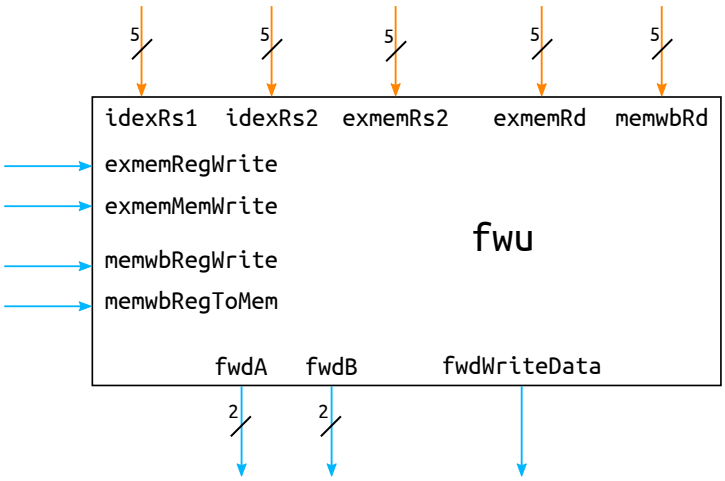

Figure 7: Forwarding Unit

2.2 Hazard Detection Unit (HDU)

When data cannot be forwarded, then the pipeline must be inevitably stalled by preventing the fetch of a new instruction and inserting a bubble. Specifically, this happens when a data hazard occurs between a load (`memRead` asserted in the decode stage) and another using instruction (unless it is a store, for which forwarding accounts).

Moreover, when a branch is taken or an unconditional jump occurs, a similar action of flushing the entire pipeline to get rid of invalid instructions already in execution must be taken.

Both this occurrences are handled by the Hazard Detection Unit, that according to the aforementioned checks, outputs three signals:

- `stall_n`: active low, is connected to the enable of the Program Counter and the IF/ID pipe register to prevent them from changing in the event of a stall.
- `flushIdEx`: to drive the multiplexer inserting the NOP in the ID/EX register, that will propagate to the rest of the pipeline, in case of a stall or a jump.

- `flushIfIdExMem`: to drive the multiplexer inserting the NOP in the IF/ID and EX/MEM registers in case of jump.

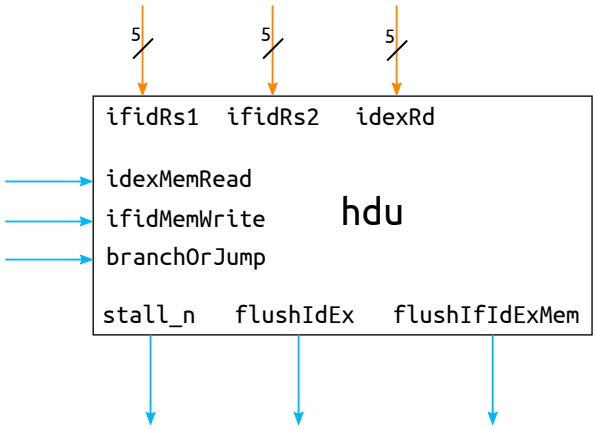

Figure 8: Hazard Detection Unit

A resume is reported in Table 1:

	stall_N	***flush_IdEx***	***flush_IfId_ExDmem***
No Hazard	0	0	0
Control Hazard	0	1	1
Data Hazard	1	1	0

Table 1: HDU output

Inserting a NOP The NOP instruction is not present in the RISC-V ISA. It is possible to emulate it though, using an **ADDI x0 x0 0**. This instruction does nothing, because the register x0 is hardwired to reference. The instruction NOP belongs to the set of pseudo-assembly instructions: they are translated in RISC-V language on the fly by the assembler, and they exist for programmers ease only. For an effective NOP insertion in the IF/ID stage there's the need for a sequential control of the multiplexer which drive the source of the RF. There is no way of doing it before the pipe register, unless a NOP instruction is already present somewhere in memory.

In figure 9 a high level DP for hazard management is depicted. Moreover, a timing diagram to show how instructions are flushed is depicted in figure 10. The three instructions which follow a jump or a taken branch are discarded.

2.3 Control Unit (CU)

3 Testbench

3.1 Memory

Some kind of memory model is needed in order to perform a full simulation of the processor core. Given that the design of a full fledged memory subsystem is beyond the scope of this experience, we resorted to a simple behavioral model of a synchronous memory.

This model is only intended for simulation purposes and does not map a real memory chip on its own, but emulates the function of a more complex memory controller able to select individual bytes among a 32-bit word both in read and in write operations.

Figure 11 shows the interface of this block, where the `address` is left parametric, as it can differ between instructions and data memory. Note that compliant to the RISC-V byte addressing specification, each address represents a single byte, even if the data width is always 32 bits, which is the width of the data bus of the architecture. The data width for load and stores is selected by the `addrUnit` signal, according to the following encoding:

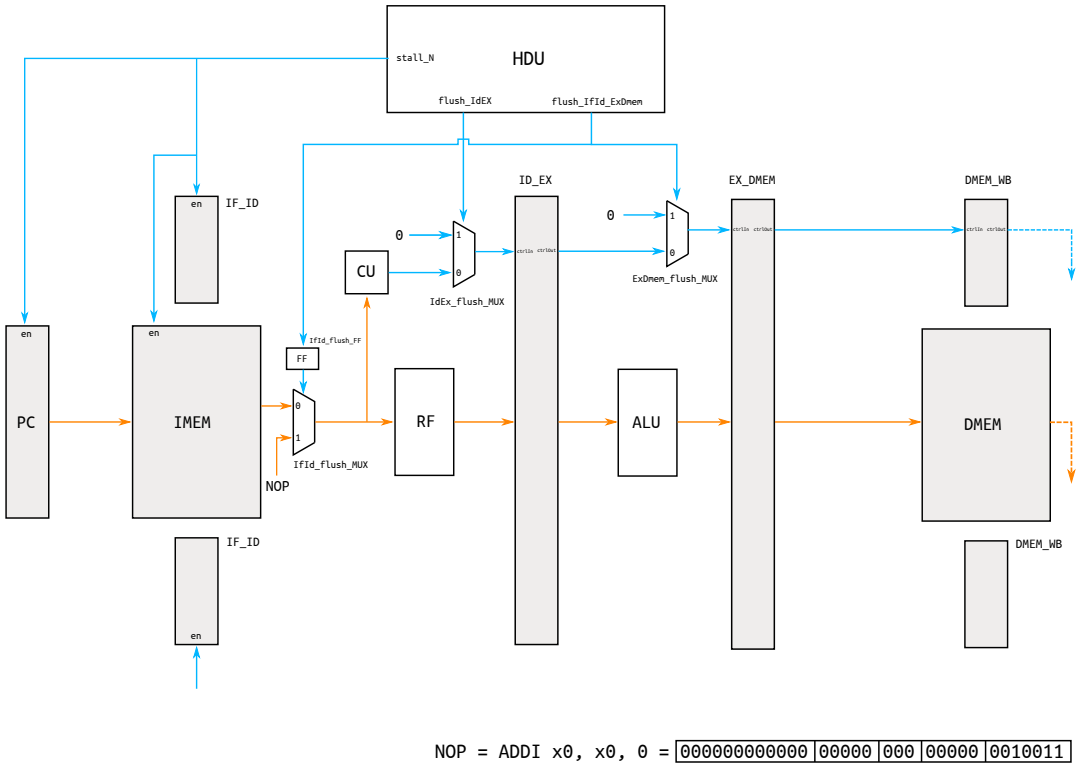

Figure 9: Hazard management HW

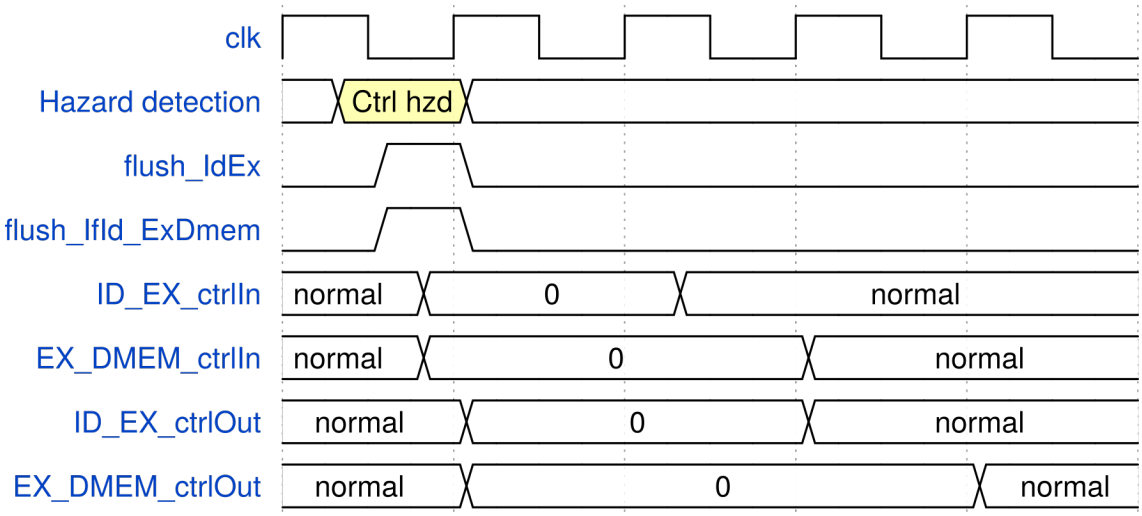

Figure 10: Hazard management timing

- 00: byte
- 01: halfword (16 bits)
- 10: word (32 bits)

Independently of the data width chosen, the correct output is always provided within a single clock cycle. It this behavior was to be replicated on a real byte-addressed memory chip, it would take (at most) four read operations and a clock four times faster.

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.

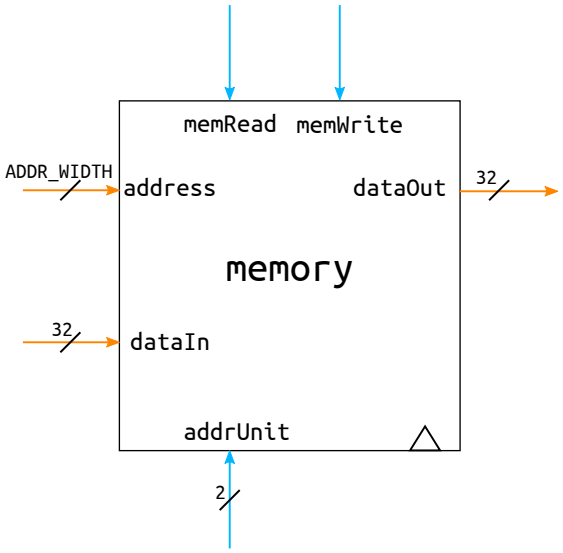

Figure 11: Memory

Figure 12 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.

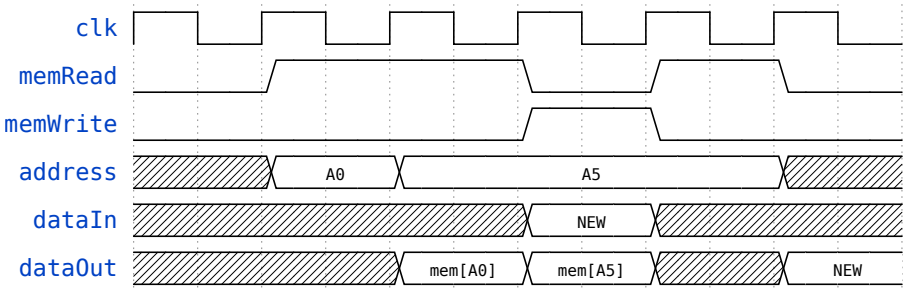

Figure 12: Memory timing diagram