Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona
Master in Innovation and Research in Informatics (MIRI)
Processor Architecture (PA)

PA project report

Implementation of a pipelined processor

Vanessa Büsing

Miquel Vidal Piñol

Xavier Yepes Arbós

January 22, 2016

Contents

List of Figures

1 Pipelined processor

The development of this project has been distributed in different phases in order to ensure that, at the end of each, we have a valid product. First one, is a basic processor with a 5-stage pipeline.

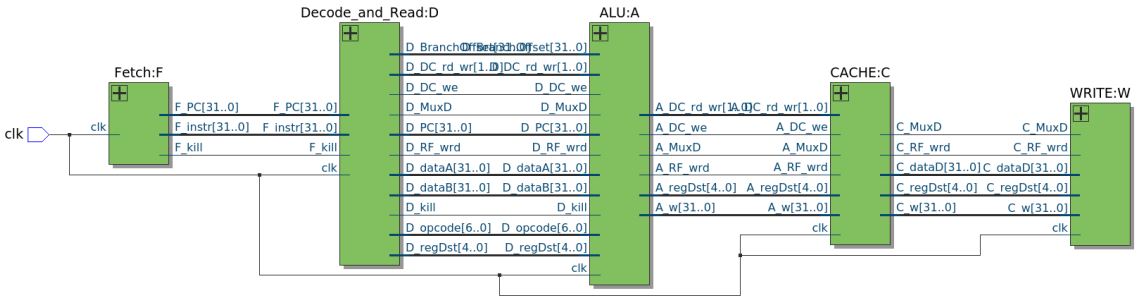

Figure 1: Pipeline stages

In the following subsection we describe the different stages of this first development phase of the basic processor.

1.1 Program Counter and Fetch

In this stage we perform the fetching of a new instruction from memory. The address of this instruction is in the Program Counter (PC). With an `Adder` module we calculate the new PC of the following instruction, this calculation is done in the `datapath`. After that, we use the signal `muxPc` to select between the three different signals that arrive to the multiplexer in order to see if we go to the `newPC`, to the `jump` or to a `branch`.

1.2 Decode and Read

In this stage the decoding of an instruction takes place. We decode the instruction in the `Control Unit`, but we also take some signals directly from the `datapath` in order to simplify the control logic. On one side, we have the wires that we extract from the `instr` bus, and we drive them to the desired port:

- `op`: Input of the ALU.
- `addrA`: Address A's input of the Register File.
- `addrD`: Address D's input of the Register File.
- `muxAddrB`: Inputs 0 and 1 of the muxAddrB.
- `muxB`: Inputs 1 and 2 of the muxB. They are the offset which are concatenated and sign extended according the type of instruction.
- `muxBr`: It is used in the sum of the address in the input 1 of the muxBr.

We also have a module called `Control Unit` (UC) who is in charge of decoding the instruction in order to generate correct control signals:

- `MuxB`: It is the selection signal for the `muxB`.
- `MuxD`: It is the selection signal for the `muxD`.
- `MuxAddrB`: It is the selection signal for the `muxAddrB`.
- `RF_wrd`: It is the signal that enables writing in the Register File.
- `DC_rd_wr`: It is a signal with 2 bits that identifies which type of memory instruction will use `dCache`. It codifies the following cases:
  - "00": Load byte.
  - "01": Load word.
  - "10": Store byte.
  - "11": Store word.
- `DC_we`: It is the signal that enables writing in the data cache.
- `IC_we`: It is the signal that enables writing in the instruction cache.
- `MuxPc`: It is the selection signal for the `muxPc`.

On the other hand, in this stage we also read the values from the registers `addrA` and `addrB`.

All the signals are connected to the register that separates `Decode` and `ALU` stages, except `addrA` and `addrB` signals.

1.3 ALU

The ALU module allows to perform arithmetic and logic operations in the Central Processing Unit (CPU).

We define our ALU module as follows:

- Inputs:
  - `op`: the operation to be performed is specified as a numerical code.
  - `x, y`: input data to be processed with the operation `op`, where `x` is the value of the source register ra and `y` can be the value of the source register rb or an immediate.
- Outputs:
  - `w`: the result of the operation `op` besides the `x` and `y` data.
  - `z`: the zero bit is used for the `BEQ` operation and indicates if `x` and `y` are equal or not.

1.4 Memory

If we are executing a Load or a Store, this stage becomes useful, because we have to access to memory. We have implemented a data cache in order to speed up the processor. In other words, reduce the period time. Currently, we have both `dCache` and `memory` modules, but we are only using the cache, due to lack of time. Our intention was to implement full memory hierarchy. We explain more in detail the memory hierarchy in section 2.

1.5 Write back

This is the last stage and it writes back to the Register File the value from the ALU or the data cache.

2 Memory hierarchy

In order to speed up the processor it is necessary use cache memories. For our project we used 2 caches, one for instructions which is used at `Fetch` stage and the other one for the data which is used at `Cache` stage.

2.1 Instruction cache

The instruction cache is directed-mapped with 4 lines of 128 bits. We decided to use a directed-mapped because is easier to implement and for out goal is enough.

Since instruction cache is not modified, we don't have to worry about any replacement policy, we just can replace a line without any problem. In case of miss, we should have to go to memory for data. This takes 5 cycles, and 5 more, for coming back. As a consequence, we have to stall the processor.

The structure of the cache is as follows:

- `Tag`: This is a container used to store tags of each line in order to compare and know if he have a hit or a miss.
- `Valid bits`: This is an array of bits indicating if a line is valid or not. If it is not valid, we can not use it.
- `Data`: This is a container where we store the data itself.

In order to use these components, we have to interpret the bits of a given address. In the figure 2 we have an address and its fields. We can identify the TAG, which is used to compare valid lines, bits 4 and 5, used to set the cache line, bits 2 and 3 used to know where the instruction is placed within a line and last two, 0 and 1, that now are not useful because instructions have 4 bytes.

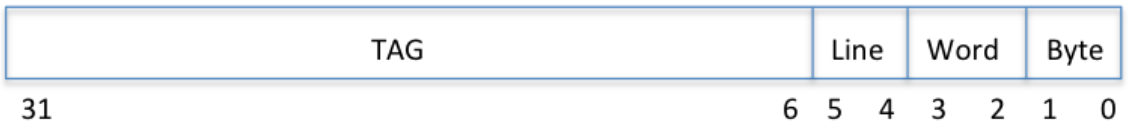

Figure 2: Address and fields

2.2 Data cache

The cache instruction is a variant from the instruction cache, a little more refined. The basic structure is the same, but we also have an array of dirty bits, because now we can have modified lines which are not updated in memory. So now, we have to use a replacement policy.

So, when we have to replace a line and has the dirty bit to 1, we have to write the line into memory. Then, if it is a store we just write into cache, but if it is a

load, we have to go to memory and come back. In the worst case, this will take 20 cycles (dirty bit to 1 + load). But as we said, we don't have this implemented, despite being our intention.

On the other hand, we also use the same fields of an address, i.e., TAG, line and word. But now, we also take into account the last field, byte (0 and 1), because we are interested in access to a byte level, since we have load byte and store byte.

2.3 Memory

At the top of memory hierarchy we have the main memory, who is shared between both instruction and data caches. Our intention was to communicate caches with memory using a bus of 128 bits, which are the bits of a cache line. Memory interface looks like as follow:

- `clk`: Input clock.
- `rd_wr`: If 0 it is a read, otherwise it is a write.
- `we`: It is the signal that enables writing.
- `addr`: It is a bus of 32 bits with the address.
- `data_wr`: It is a bus of 128 bits with the data to write into memory. It is only used by data cache because instruction cache is read-only.
- `data_rd`: It is a bus of 128 bits with the data to write into one of the caches. We need to synchronize both caches in order to use memory properly.

2.4 Controllers and arbiter

Without using any synchronization method, we would have a structural hazard, because we could have two misses at the same time, one from `Fetch` stage (instruction cache) and the other one from the `Memory` stage (data cache). So we have to ensure only one access to memory. For this, it would be necessary to implement an arbiter that gives access to only one request. Our policy would be:

- If the arbiter receives two requests simultaneously, we serve first to data cache, because we want to finish older instructions. Then, we would serve to instruction cache.
- If we are serving a cache miss (no matter if instruction or data miss) and we receive a new request, first, we finish the current request, and then, we serve the second one.

On the other hand, it would be also necessary to implement two controllers, one for each cache, in order to synchronize requests from caches to the arbiter. All these three components should be designed according to a finite state machine graph.

3 Instructions' pipeline flow

All the instructions begin their execution in the `Fetch` stage of the pipeline, where they are loaded from the Instruction Cache. In case of a cache miss, the cache line must be loaded from memory and stall the instruction flow until the cache is updated.

Once the instruction code has been loaded, it is passed in the next clock cycle to the following stage, which is the `Decode and Read` stage.

In the `Read and Decode` stage, the instruction code is decoded in the `Control Unit` module. That module, depending on the type of instruction that has been decoded, activates and controls some signals needed in the next pipeline stages.

3.1 ADD

3.1.1 Decode and Read

ADD is an arithmetic-type instruction, like SUB and MUL, and is decoded in the `Control Unit` generating the following signals:

- `DC_rd_wr` *XX*: since the dCache won't be used.
- `DC_we` *0*: we must assure that nothing is written into the dCache.
- `IC_we` *0*: we must assure that nothing is written into the iCache.
- `MuxAddrB` *0*: bits [14:10] of the instruction code encode register *b*.
- `MuxB` *00*: we want to select the register *b* that has been read from the register file to pass it to the ALU.
- `MuxD` *0*: we want to select the output of the ALU to write the result to the register file.
- `MuxPC` *00*: we want to select PC+4 as the next PC.
- `RF_wrd` *1*: we want to write into the register file.

At the same time, the registers *a* and *b* are being read from the register file.

All this signals, with the exception of `MuxAddrB`, `MuxB`, `IC_we` and `MuxPC` that are used only in this stage, will be passed through to the next pipeline stage. Also, bits [31:25] and [24:20] of the instruction code are also passed through to the next stage. These bits correspond to the opcode for the ALU and the destination register, respectively.

3.1.2 ALU

This stage receives registers *a* and *b*, signals `DC_we`, `MuxD`, `RF_wrd` and bits [31:25] (the opcode) and [24:20] (the destination register).

The module `ALU` will receive registers *a* and *b* and the opcode, which encodes an ADD operation.

The result of the ALU operation (w), as well as signals DC_we, MuxD, RF_wrd and bits [24:20] (the destination register) are passed through to the next stage.

3.1.3 Cache

In the case of an arithmetic operation, such as an ADD, in this stage there is no work to be done. The only thing that must be done is to use the signal DC_we to prevent any modification in the dCache.

The ALU result (w), signals MuxD, RF_wrd and bits [24:20] (the destination register) are passed through to the last pipeline stage.

3.1.4 Write

The final stage of the pipeline. Here, the remaining signals are being put in use.

ALU result w is written in the register indicated by bits [24:20] using the signal RF_wrd, that enables the register file write. It is necessary to use signal MuxD in order to write into the register file the result w.

3.2 LD

3.2.1 Decode and Read

The LD instruction is decoded in the `Control Unit` generating the following signals:

- `DC_rd_wr` bits[26:25] of the instruction code. If the instruction is a LDB it will be *00* and *01* in case of a LDW.
- `DC_we` *0*: we must assure nothing is written into the dCache.
- `IC_we` *0*: we must assure nothing is written into the iCache.
- `MuxAddrB` *0*: however this signal does not really matter, as the register *b* will not be used.
- `MuxB` *01*: we want to select the immediate encoded in the bits [14:0] of the instruction code to pass it to the ALU.
- `MuxD` *1*: we want to select the data coming from the dCache to write it to the register file.
- `MuxPC` *00*: we want to select PC+4 as the next PC.
- `RF_wrd` *1*: we want to write into the register file.

Signals `DC_rd_wr`, `DC_we`, `MuxD`, `RF_wrd`, register *a*, bits [14:0] (encoding an immediate with sign extended), bits [24:20] (the destination register) and bits[31:25] (the opcode) are passed through to the next stage.

3.2.2 ALU

The *ALU* will compute the addition of the value in register *a* and the immediate encoded in bits [14:0].

The result of the `ALU` operation (`w`), as well as signals `DC_rd_wr`, `DC_we`, `MuxD`, `RF_wrd` and bits [24:20] (the destination register) are passed through to the next stage.

3.2.3 Cache

In this stage, the signal `DC_we` is used to prevent any writing in the dCache and the signal `w` (the ALU output) is used as the data address to be read from the dCache. Signal `DC_rd_wr` is used to choose what kind of operation we must do in the dCache, in this case a LDB or a LDW.

With the signal `MuxD` we select the data that has just been read from the dCache to pass it through to the next stage. Signal `RF_wrd`, as well as bits [24:20] (destination register), is also passed through to the next and final stage.

3.2.4 Write

The data read from the dCache is written in the register indicated by bits [24:20] using the signal `RF_wrd`, that enables the register file write. It is necessary to use signal `MuxD` in order to write into the register file the data loaded from the dCache.

3.3 ST

3.3.1 Decode and Read

The ST instruction is decoded in the `Control Unit` generating the following signals:

- `DC_rd_wr` bits[26:25] of the instruction code. If the instruction is a STB it will be *10* and *11* in case of a STW.
- `DC_we` *1*: we want to write into the dCache.
- `IC_we` *0*: we must assure nothing is written into the iCache.
- `MuxAddrB` *1*: we want to read from the register *b* because this is the data that we will store into the dCache.
- `MuxB` *01*: we want to select the immediate encoded in the bits [14:0] of the instruction code to pass it to the ALU.
- `MuxD` *X*: we will not write anything into the register file.
- `MuxPC` *00*: we want to select PC+4 as the next PC.
- `RF_wrd` *0*: we must assure nothing is written into the register file.

Signals `DC_we`, `RF_wrd`, `DC_rd_wr`, register *a*, register *b*, bits [14:0] (encoding an immediate with sign extended), and bits[31:25] (the opcode) are passed through to the next stage.

3.3.2 ALU

The *ALU* will compute the addition of the value in register *a* and the immediate encoded in bits [14:0].

The result of the `ALU` operation (`w`), as well as signals `DC_we`, `DC_rd_wr`, `RF_wrd` and register *b* are passed through to the next stage.

3.3.3 Cache

The signal `w` (the ALU output) is used as the dCache data address where data of register *b* is going to be written.

Signal `RF_wrd` is also passed through to the next and final stage.

3.3.4 Write

A store in this stage does not do anything, but we have to ensure that the register file is not modified using the signal `RF_wrd`.

3.4 BRANCH

3.4.1 Decode and Read

The branch instructions, BEQ and JUMP, are decoded in `Control Unit` generating the following signals:

- `DC_rd_wr` *XX*: we don't need this signal because it is only used by memory instructions.
- `DC_we`: we must assure nothing is written into the dCache.
- `IC_we`: we must assure nothing is written into the iCache.
- `MuxAddrB` *0*: bits [14:10] of the instruction code encode register *b*.
- `MuxB` *[00-10]*: if we have a BNQ, it will be 00, as we want to read from the register file. Otherwise, if we have a JUMP, it will be 10, as we want the offset Hi, Medium and Low of the instruction.
- `MuxD` *X*: we will not write anything into the register file.
- `MuxPC` *[00-01-10]*: we can have three possibilities.
  - *00*: if we have a BNQ but we don't have to jump.
  - *01*: if we have a BNQ and we have to jump.
  - *10*: if we have a JUMP.
- `RF_wrd` *0*: we must assure nothing is written into the register file.

3.4.2 ALU

In this stage we have to distinguish to cases:

- BNQ: if we have a BNQ, we will use the ALU for comparing the two read registers and set z. If they are equal, z will be 1, otherwise it will be 0. The value of z is sent to the `Decoder` in order to set properly `muxPc`. Furthermore, we have to add the PC of BNQ (`DA_Pc`) with the offset (`DA_BranchOffset`).
- JUMP: if we have a JUMP, we will use the ALU for adding the value of register `a` with the offset, which is coded with bits [24:20] and [14:0]. We also pass the `opcode` to the Decoder in order to warn it that it has to set properly `muxPc`.

We have also pass to the Decoder the signal `DAA_kill` which indicates if the branch has been killed previously by another branch. This is the last instruction for the branch instruction, but we ensure that will not modify anything. So, we will pass to the next stage signals `DAA_DC_we` and `DAA_RF_wrd`.

3.4.3 Cache

Here branch instructions do nothing, but we use signal `DAA_DC_we` for ensuring that we don't write into dCache. We pass signal `DAA_RF_wrd` to the last stage.

3.4.4 Write

In the last stage, branch instructions also don't do anything, but we use signal `DAA_RF_wrd` for ensuring that we don't write into the register file.

4 Tests and chronograms

In this section we will show some waveforms of the processor. We would like to put waveforms of our pipelined processor, but it doesn't work properly yet. So we decided to put some waveforms of our basic processor (mono-cycle) which works fine.

In figure 3 there's a waveform of an ADD instruction.

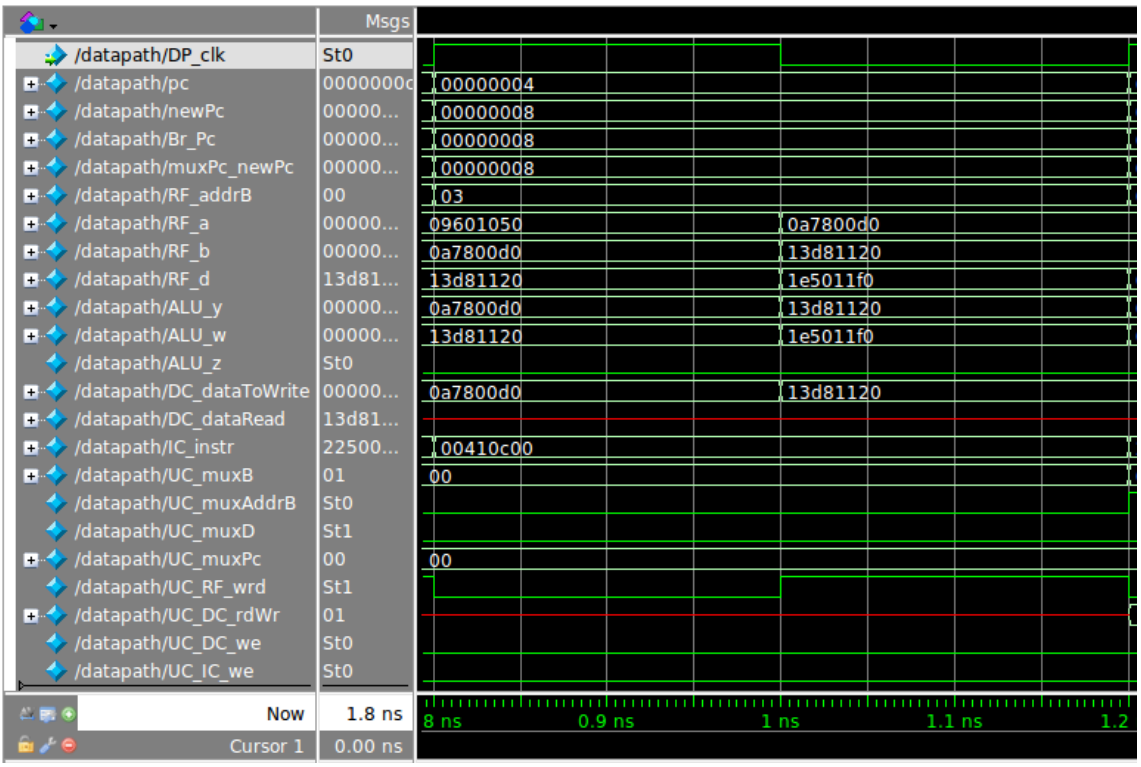

Figure 3: Add

In figure 4 there's a waveform of a LDB instruction.

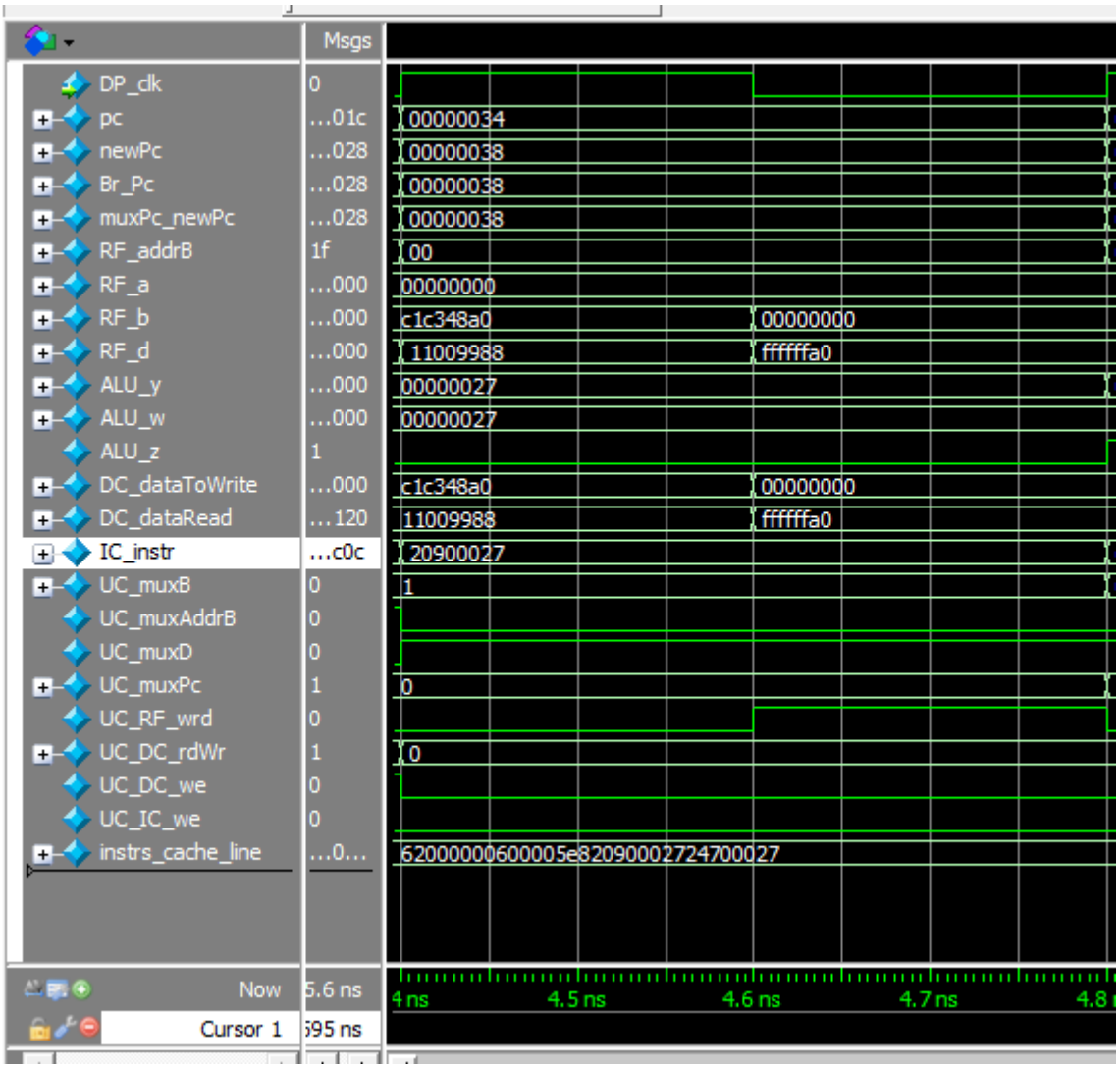

Figure 4: Load byte

In figure 5 there's a waveform of a LDW instruction.

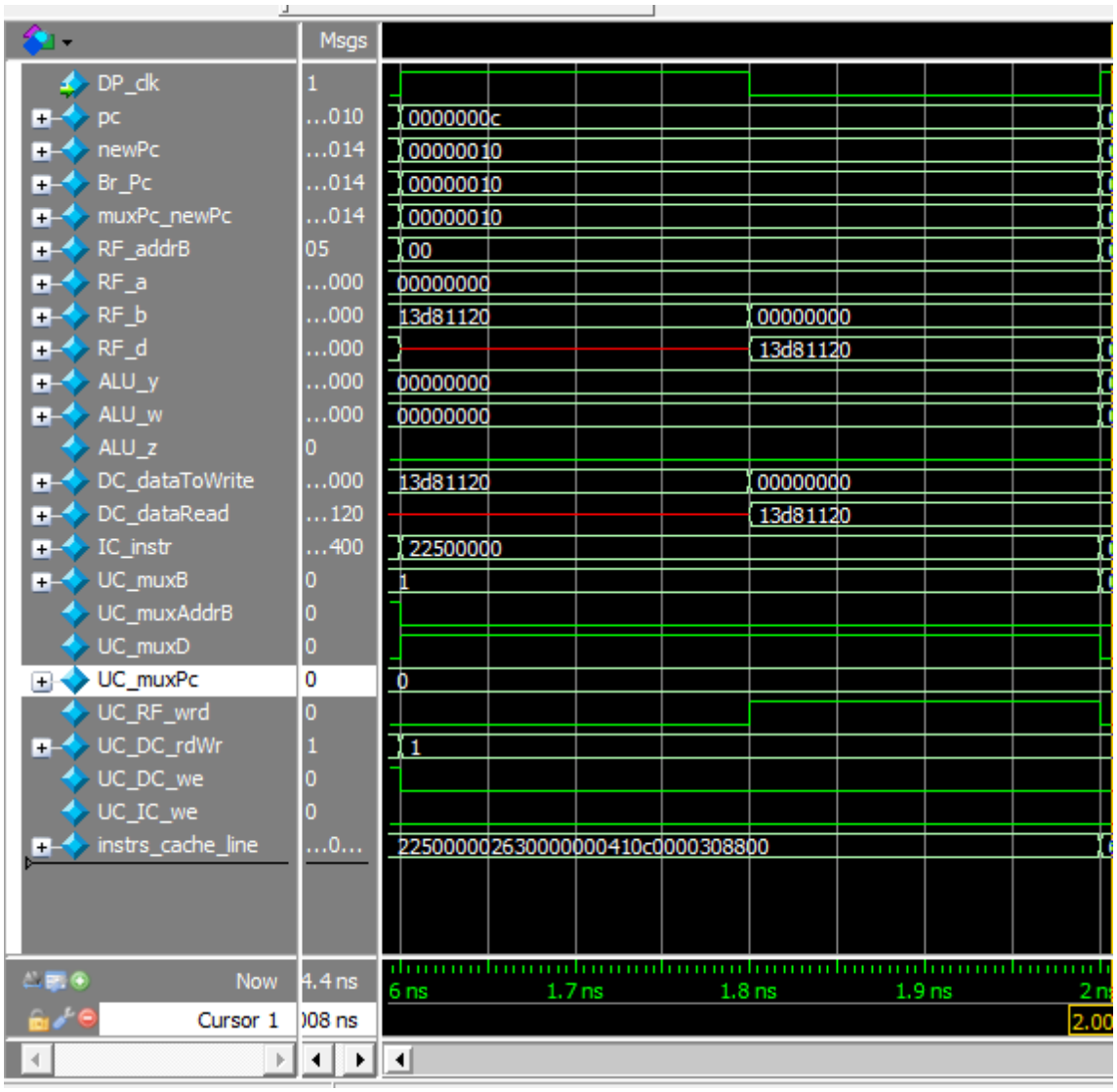

Figure 5: Load word

In figure 6 there's a waveform of a STB instruction.

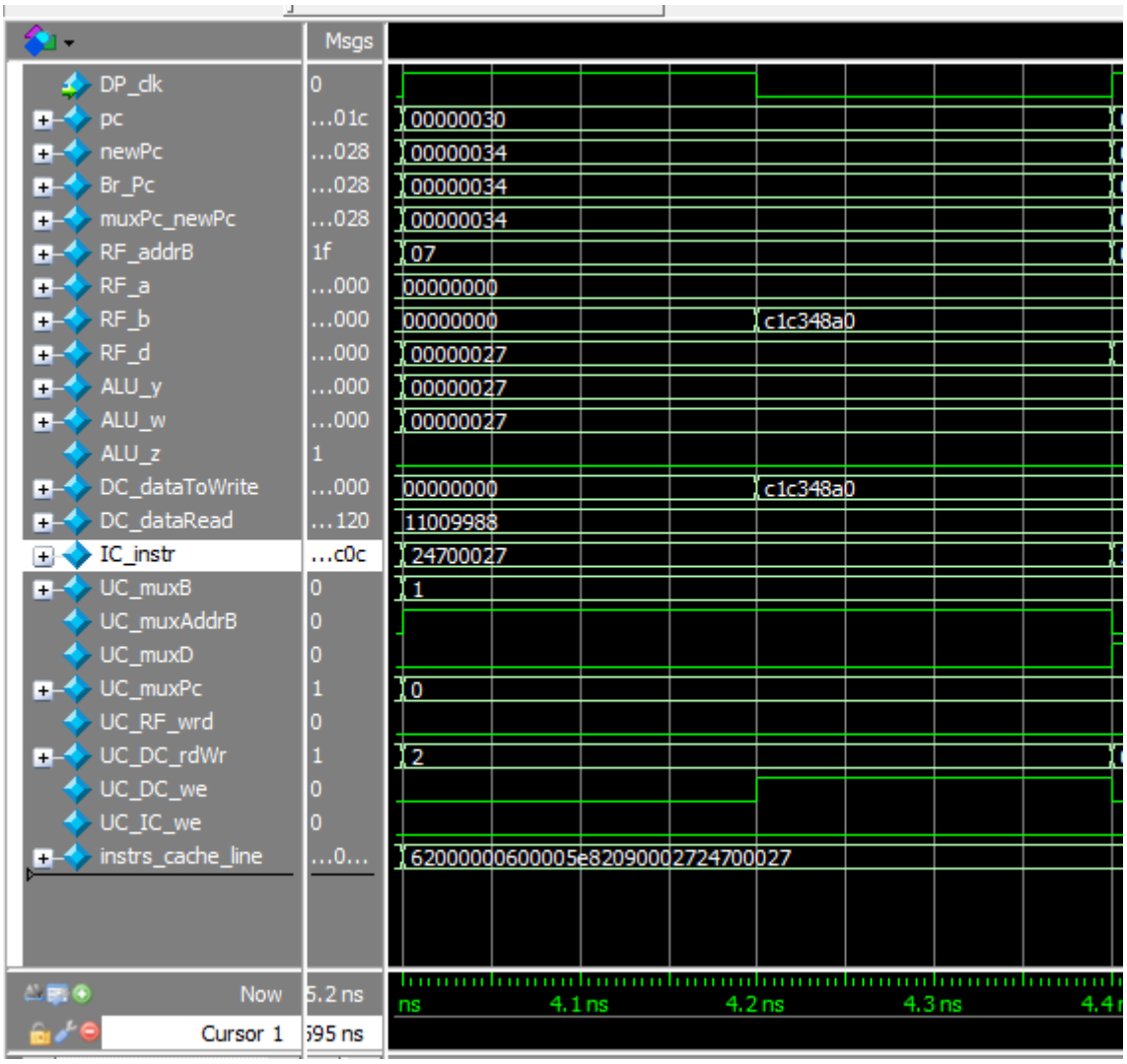

Figure 6: Store byte

In figure 7 there's a waveform of a STW instruction.

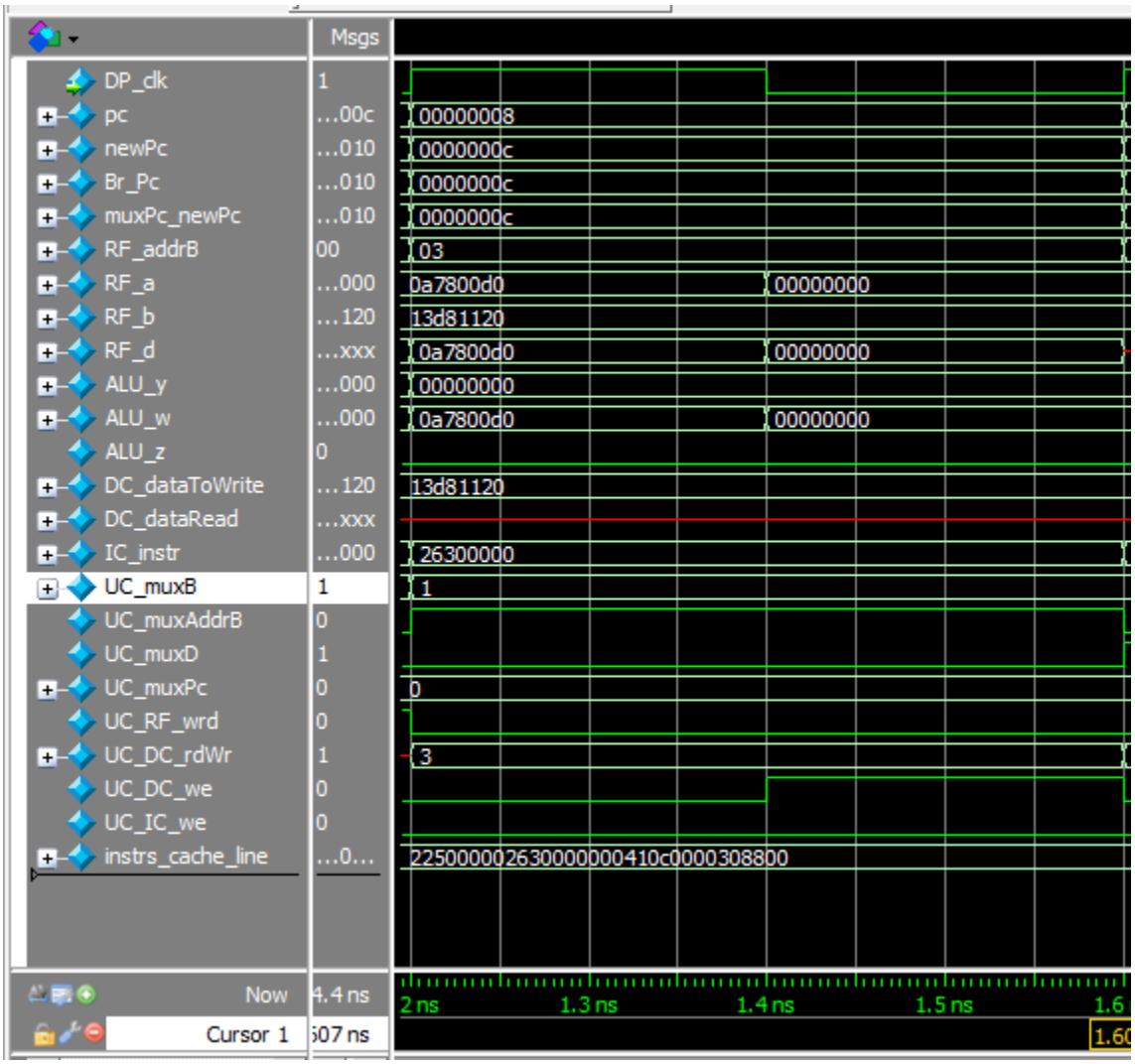

Figure 7: Store word

In figure 8 there's a waveform of a BEQ instruction.

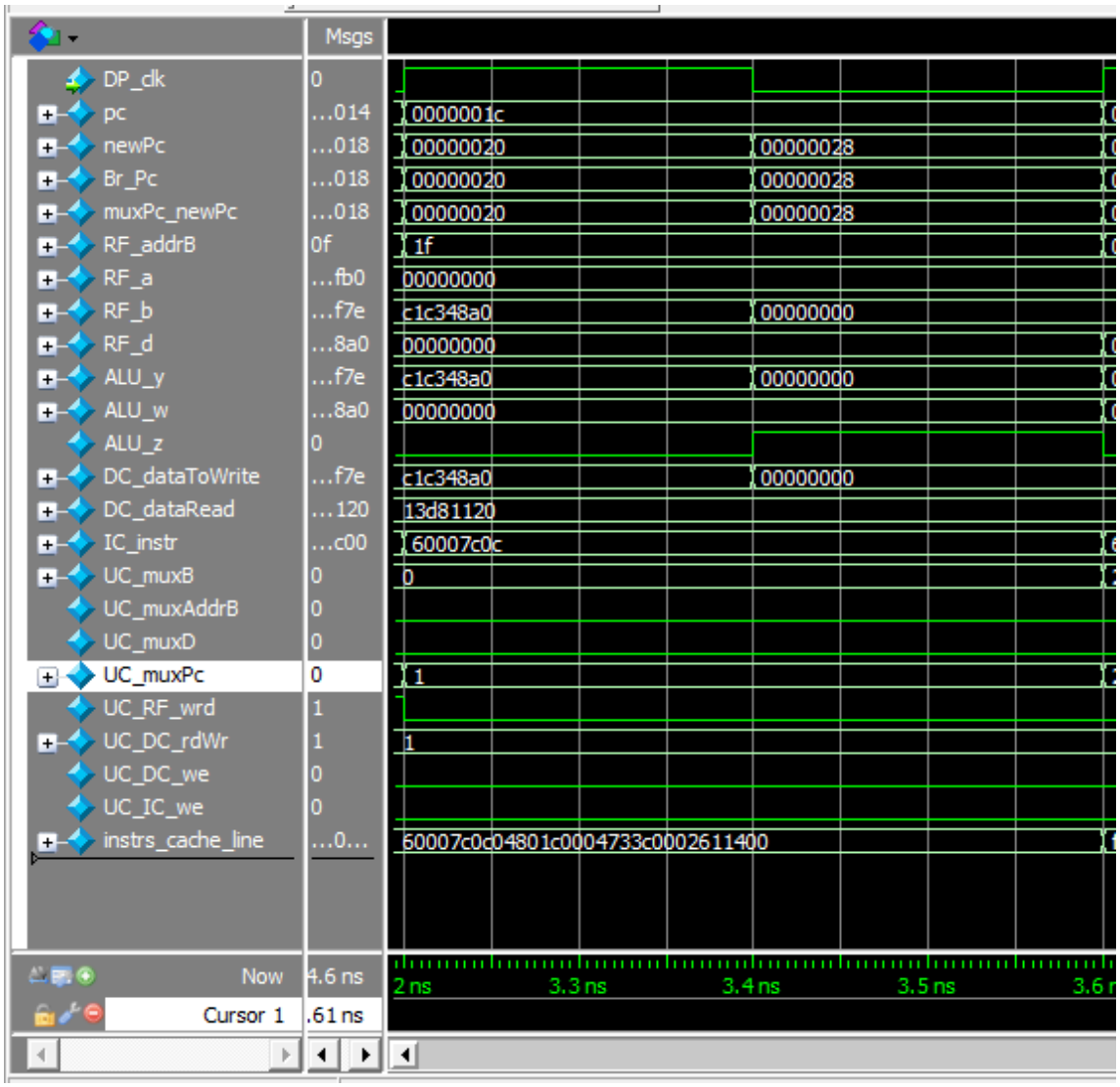

Figure 8: Branch equal (BEQ)

In figure 9 there's a waveform of a JMP instruction.

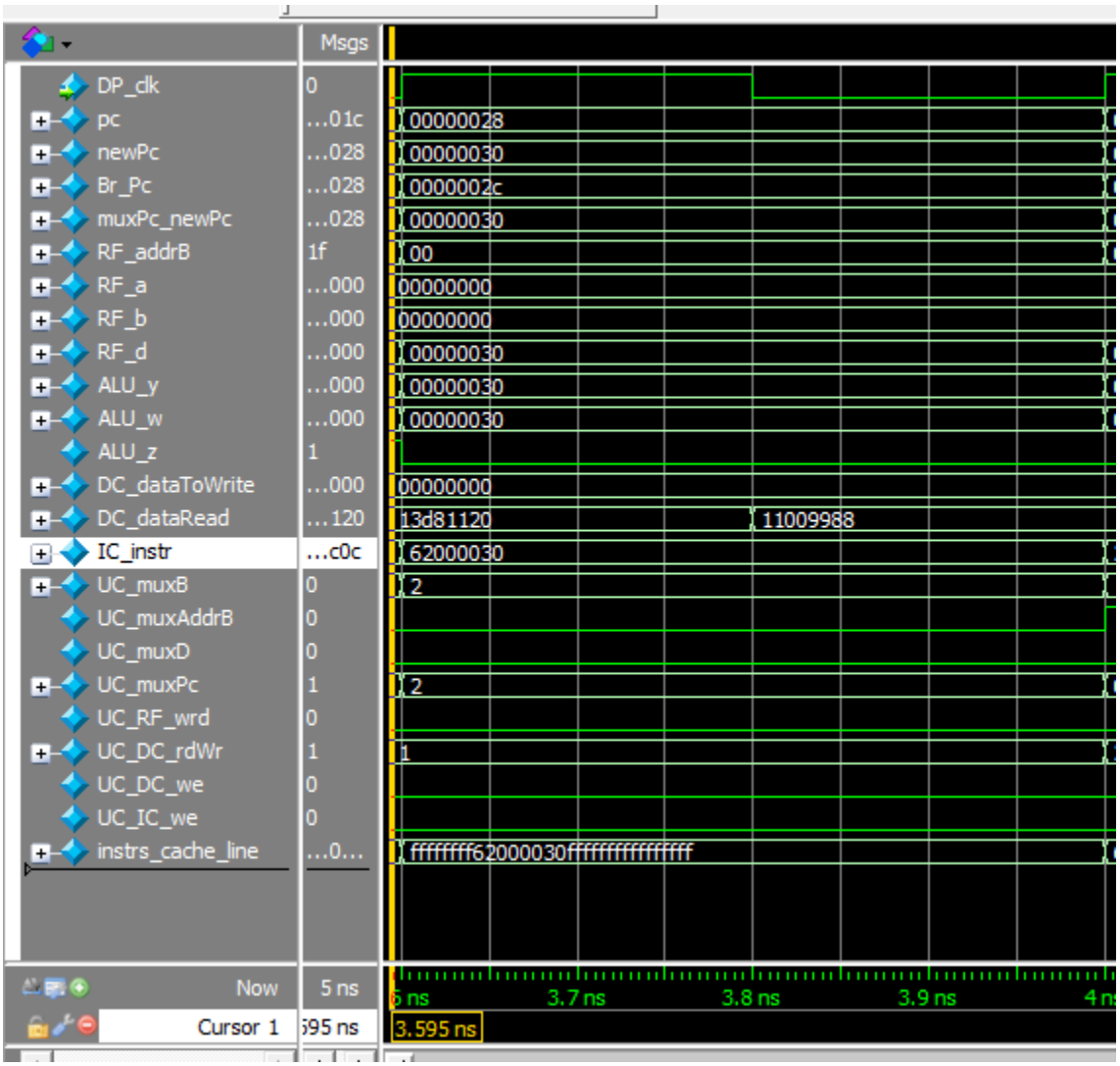

Figure 9: Jump (JMP)