Computer Architecture Project

May 17, 2019

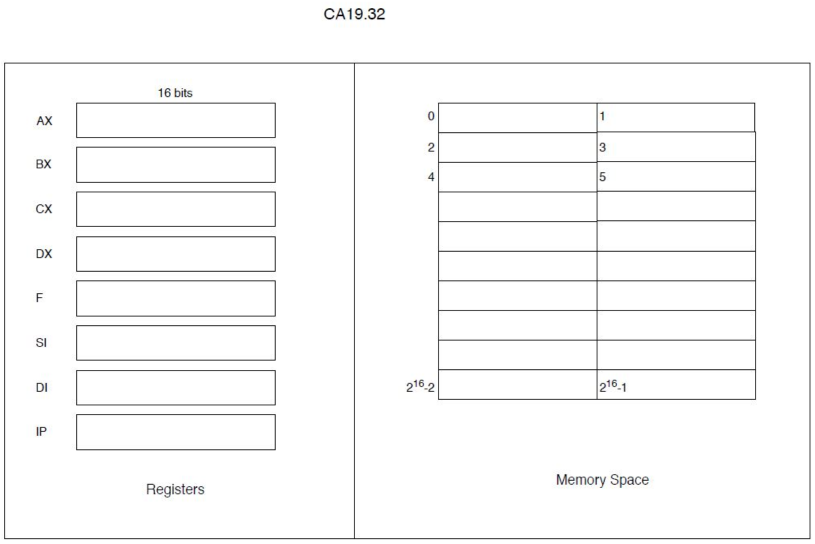
Decode Module

Contents

1 Introduction

1.1 Processor CA19.32

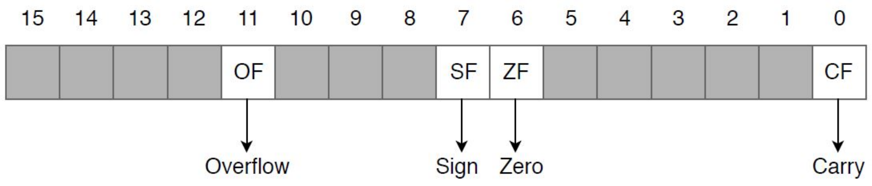
A computer based on processor CA19.32 appears as shown in the following figure that shows processor registers and the memory space:



The memory is composed of 2^{16} locations. Each location can contain one byte and can be identified by a 16-bit address. The processor accesses the memory space during the fetching and execution phase of the operating instructions.

The processor contains 8 16-bit registers: Accumulator registers: AX, BX, CX, DX Pointer registers: SI, DI, IP Flag register: F

The flag register F has only 4 significant bits:



- **CF (Carry Flag):** when it contains 1 it indicates that during the execution of the last operation a carry-over was generated or a loan was requested;

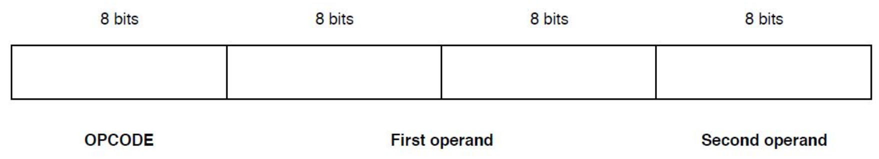
- **ZF (Zero Flag):** when it contains 1 it indicates that the last instruction executed produced a result with all bits at 0;
- **SF (Sign Flag):** when it contains 1 it indicates that the last executed instruction produced a result with the most significant bit at 1;
- **OF (Overflow Flag):** when it contains 1 it indicates that during the execution of the last instruction there was an overflow.

1.2 Instruction format

The processor uses two types of instructions:

- **operating instructions** that specify the kind of processing that the processor must perform and how it can identify the data to be processed.
- **control instructions** that specify the type of processing and the next instruction to perform.

The instruction format is the following:



The size of the instruction is 32 bits in total. The OPCODE field (8 bits) specifies the type of processing the processor will perform when executing that instruction. The first and second operands depend on the addressing mode of the opcode. Some instructions do not require the use of the 8 bits of the second operand.

The instructions are grouped in 5 formats (from F0 to F4). The first byte of the instruction encodes the format, the type of the instruction and the name of any registers referred to in the instruction itself. More in detail, the format is encoded by the three most significant bits.

Format 0 does not contain operands. Instructions belonging to this format are:

- HLT
- NOP

Format 1 has a single immediate operand that represents an address. Instructions belonging to this format are:

- JE immediate
- JNE immediate
- JA immediate
- JAE immediate
- JB immediate
- JBE immediate
- JG immediate
- JGE immediate
- JL immediate
- JLE immediate
- JZ immediate
- JNZ immediate
- JC immediate
- JNC immediate
- JO immediate
- JNO immediate
- JS immediate
- JNS immediate
- JMP immediate

Format 2 has a single operand that represents the encoding of a register. The register can contain:

- a memory address in the case of jump instructions
- the value to be changed for the other instructions

Instructions belonging to this format are:

- JE register
- JNE register
- JA register
- JAE register
- JB register
- JBE register
- JG register
- JGE register
- JL register
- JLE register
- JZ register
- JNZ register
- JC register
- JNC register
- JO register
- JNO register
- JS register
- JNS register
- INC register
- DEC register
- NEG register
- NOT register
- JMP register

Format 3 has two operands: the first represents an immediate and the second the encoding of the destination register.

Instructions belonging to this format are:

- MOV immediate, register
- ADD immediate, register
- SUB immediate, register
- CMP immediate, register
- MUL immediate, register
- IMUL immediate, register
- DIV immediate, register
- IDIV immediate, register
- AND immediate, register
- OR immediate, register
- SHL immediate, register
- SAL immediate, register
- SHR immediate, register
- SAR immediate, register
- LOAD immediate, register
- STORE immediate, register

Format 4 has two operands: the first represents the encoding of the source register and the second the encoding of the destination register.

Instructions belonging to this format are:

- MOV register, register
- ADD register, register
- SUB register, register
- CMP register, register
- MUL register, register
- IMUL register, register

- DIV register, register
- IDIV register, register
- AND register, register
- OR register, register
- SHL register, register
- SAL register, register
- SHR register, register
- SAR register, register
- LOAD register, register
- STORE register, register
- XCHG register, register

1.3 Instruction cycle

The instruction cycle can generally be divided into three distinct phases:

1. Fetch
2. Decode
3. Execute

1.3.1 Fetch Phase

This is the first phase of the instruction cycle. The program counter's value is used for addressing (accessing) a specific cell from the program memory. This memory cell contains the instruction that have to be executed by the CPU at the end of the instruction cycle.

1.3.2 Decode Phase

In this phase the control unit of the CPU performs few actions based on the value of the instruction register (the fetched value):

- **Decode the instruction**: the opcode of the instruction defines what type of instruction it is (e.g data processing, memory access etc.).
- **Fetch the operands**: based on the opcode and additional information contained in the instruction format, the control unit determines what type of operands are needed and where they should be fetched from.

1.3.3 Execute Phase

The execution is the last phase of the instruction cycle. Here the actual operation specified by the fetched instruction is performed. If it is a data processing instruction then the operands are passed through the ALU and/or other data manipulation units if available for processing. At the end of the execute phase, the result of the executed instruction is stored in either the data memory or a CPU register.

2 Overall description

The whole system is run by an Orchestrator that is a system driven by events. Each of those events is created by module, like ours: Decode and Execute Module.

First of all, the object "system", which is the instance of the class that represents the orchestrator itself, is created. Then all the modules' instances are added to the orchestrator using the **AddModule function.**

After this step the **Run function** is invoked.

As we said, the orchestrator works only with events, so it's necessary that when the modules are added on the orchestrator at least one of those must create an event, otherwise no events could run.

The **Run method** is the core of the orchestrator: it checks if there are any events in each module and it adds each one of them into a event's queue sorted in chronological order.

Now there is a loop on the event's queue that ends whenever this one is empty. For each event in the queue, the **Tick function** is invoked.

The **Tick function** updates the current time of the orchestrator with the time of the event that is actually running.

The **notifyAll function** is invoked, this one is needed to send to every module the event. Now the module should recognize whether itself is the recipient of the event and then, if necessary, the module starts to work.

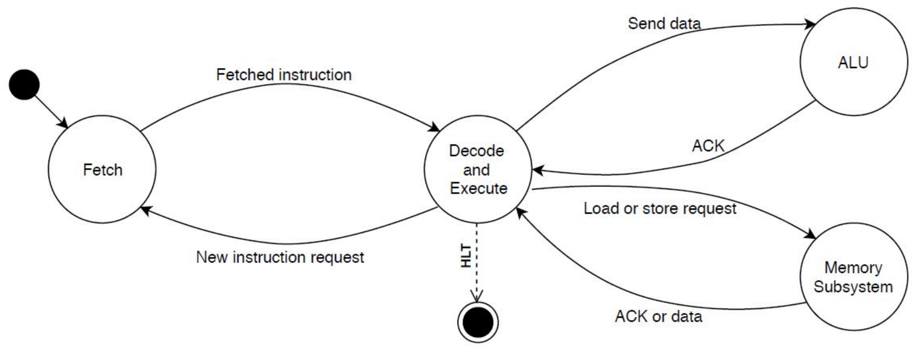
Now that a module is running, is possible that eventually it will perform a **sendWithDelay** function. This method is needed to create a new event with a certain message for another module. All these events that a module creates are added to an events' queue defined in the module itself.

When the module finishes the assigned work, its events are added to the events' queue of the orchestrator and then first ones are cleared.

Now we are back in the **Tick method**, so it deletes the last event, because it's been already handled at this point, and it terminates.

The orchestrator now it will check if there are any events and it will invoke the **Tick method** until no events are in the events' queue.

3 Interaction with other components



3.1 Fetch Module

Shared structures between Fetch and Decode modules are contained in the file: fetch_registers.h.

The Fetch module, in order to communicate the end of its operations, sends an empty message: (void* magic_struct = NULL) and leaves the data in the following structure:

```
struct fetch_registers{
    uint16_t ip;
    uint32_t mdr;
    uint16_t mar;
    uint8_t opcode;
    uint16_t source;
    uint8_t dest;
};
```

The Decode module uses only the fields highlighted in bold. In particular:

- **ip** field is changed when a jump instruction is executed
- **opcode**, **source** and **dest** are used during the decoding and executing of each instruction.

3.2 ALU Module

Shared structures between ALU and Decode modules are contained in the file: **decode_registers.h**.

To simulate the communication between ALU and Decode two data structures have been implemented, whose fields can be read and modified by the ALU, for this reason the ALU always responds with empty messages.

The first structure is called decode_registers and contains the following fields:

```
struct decode_registers{
    uint8_t opcode;     //opcode
    uint16_t operand1;  //it can be only read and
        contains the first operand
    uint16_t operand2;  //it can be only read and
        contains the second operand
    uint8_t  destination_reg;   //it can be only read
        and contains the first operand the encoding of
        the destination register
};
```

The second structure is called global_registers and contains:

```
struct global_registers{
    uint16_t flag;  //flag register
    uint16_t general_regs[6];   //registers defined in
         the Instruction Set:(AX, BX, CX, DX, SI, DI)

};
```

This second structure is necessary because only the ALU and Decode/Execute modules can access and modify these registers.

The flag register has been implemented with a single variable of type uint16_t that reflects the physical structure of the register (as shown in figure ...) instead of using a separate variable for each flag to optimize the space used.

The following mask has been created to access the single bits of the flag register (the mask has been made available also to ALU module.).

```
bool extract_flag(uint8_t index){
    return (global_regs.flag >> index) &1;
}
```

where the field index represents the index of the flag to be extracted.

3.3 Memory Subsystem Module

Shared structures between Memory Subsystem and Decode modules are contained in the file: **memory_message.h** .

Interaction takes place only during LOAD and STORE operations.

3.3.1 Communication from decode to memory subsystem

In this case the following structure is used:

```
struct memory_message {
    bool type;          // 0 = read, 1 = write
    uint16_t address;
    uint16_t data;
};
```

3.3.2 Communication from memory subsystem to decode

Varies according to the type of instruction:

- **Write:** An empty message (void* magic_struct = NULL) is sent as an ACK to communicate the end of the writing.
- **Reading:** the data is inserted in the date field of the previous structure

4 Decode and Execute Module Features

Every time the Decode/Execute module receives a message from the orchestrator, the following steps are executed:

1. Check that the destination of the message is the Decode module, otherwise the message is discarded.
2. Check the sender of the message and call the correct function to manage the received message. If the sender is:
   (a) the **fetch**: it is necessary to obtain the format of the instructions (from 0 to 4) and, according to the format, call the function that deals with the management of instructions belonging to that format. If the operation requires it, the fields of the shared structures must be filled and a message must be sent to the module involved in the operation.

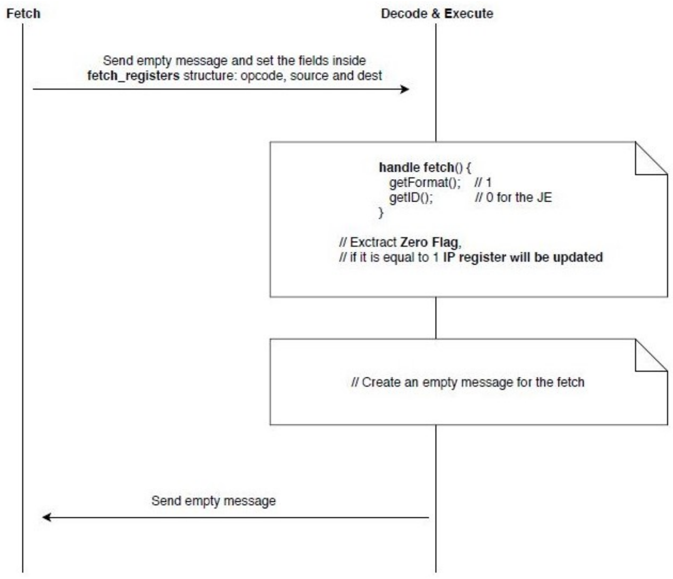
(b) the **memory subsystem**: in this case the message can contain either the ACK for a write or the data required for reading (this corresponds to the second phase of execution of the instruction).

(c) the **ALU**: in this case the message contains an ACK representing the conclusion of the required operation.

3. Send a message to the fetch to request a new instruction.

5 Use cases

5.1 Format 1 example: JE immediate address

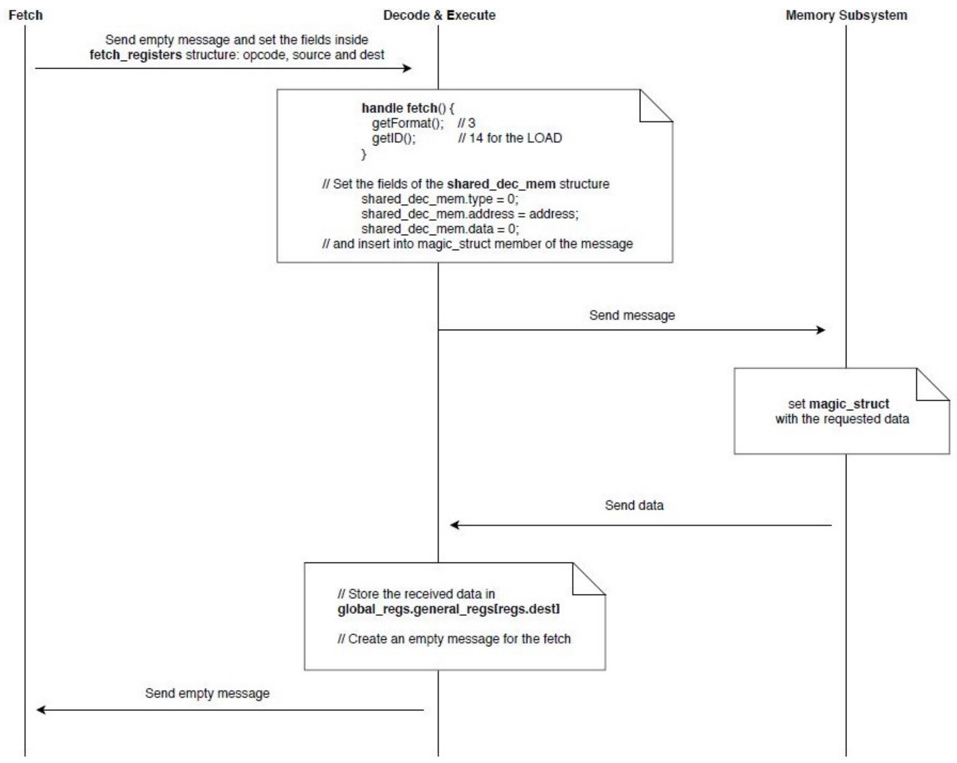

Fetch
Decode & Execute
Send empty message and set the fields inside
fetch_registers structure: opcode, source and dest
handle fetch() {
getFormat(); // 1
getID(); // 0 for the JE
}
// Exctract Zero Flag,
// if it is equal to 1 IP register will be updated
// Create an empty message for the fetch
Send empty message

The Decode/Execute receives an empty message from Fetch indicating the presence of updated data: opcode, source and dest in the shared registers of the fetch_registers structure.

The format is extracted from the opcode field (in this case is the Format 1) and the same is done for the instruction ID (in this case is 0). The operation corresponds to a JE immediate address, hence the ZF flag will be extracted through the mask extract_flag() and its value will be checked. If it is equal to 1, the IP register will be modified by entering the address on which the program should jump.

At this point, the Decode creates a new empty message and sends it to the Fetch, to request a new instruction.

5.2 Format 3 example: LOAD immediate, register


Fetch
Decode & Execute
Memory Subsystem
Send empty message and set the fields inside **fetch_registers** structure: opcode, source and dest
Send message
set **magic_struct** with the requested data
Send data
Send empty message

```
handle fetch() {
  getFormat();   // 3
  getID();       // 14 for the LOAD
}

// Set the fields of the shared_dec_mem structure
    shared_dec_mem.type = 0;
    shared_dec_mem.address = address;
    shared_dec_mem.data = 0;
// and insert into magic_struct member of the message
```

```
// Store the received data in
global_regs.general_regs[regs.dest]

// Create an empty message for the fetch
```

The Decode/Execute receives an empty message from Fetch indicating the presence of updated data: opcode, source and dest in the shared registers of the fetch_registers structure. The format is extracted from the opcode field (in this case is the Format 3) and the same is done for the instruction ID (in this case is 14). The operation corresponds to a LOAD immediate, register.

The Decode will set the fields of the shared_dec_mem structure of type memory_message:

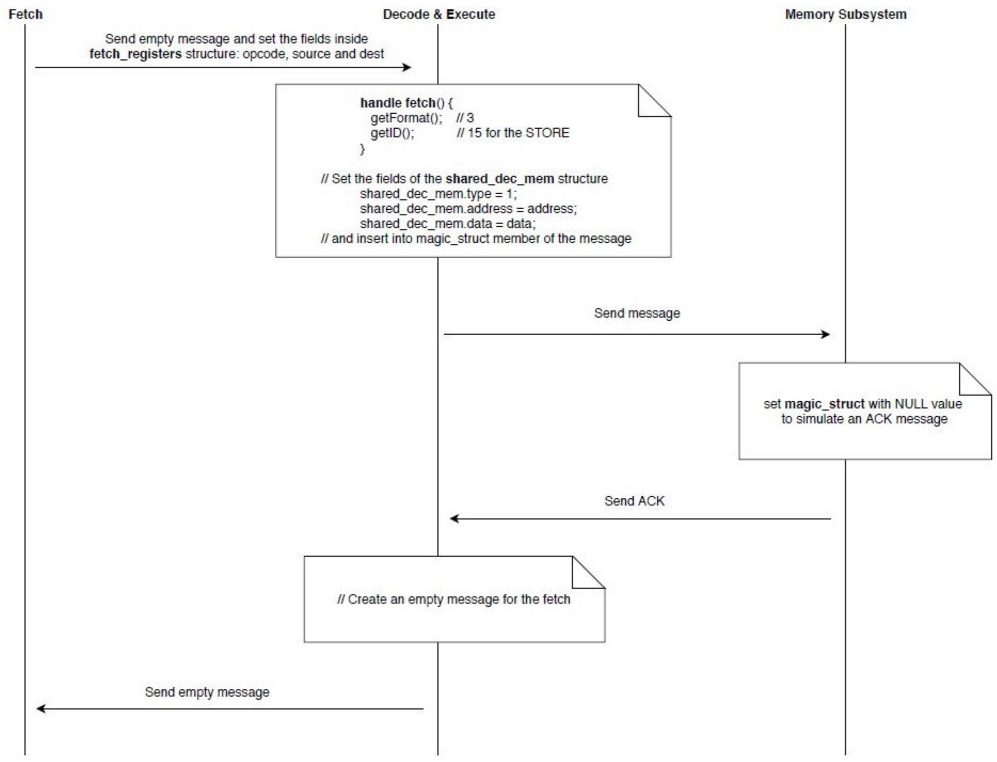
- 0 for the type of operation requested to the memory subsystem (0 for reading, 1 for writing)
- the address of the memory location from which the value must be read
- 0 for the date field, because in this case is not used (it will be managed later)

After setting the fields, this structure is inserted into the magic_struct field of the message that will be sent to the memory subsystem.

The memory subsystem will read the requested data from the memory, will insert it in the magic_struct message and will send it to the Decode/Execute module.

As soon as the Decode/Execute receives the message, it will fill the register indicated in the LOAD instruction with the data received from the memory subsystem. Next, it will create an empty message for the Fetch to request a new instruction.

5.3 Format 3 example: STORE immediate, register



Fetch
Decode & Execute
Memory Subsystem
Send empty message and set the fields inside fetch_registers structure: opcode, source and dest
handle fetch() { getFormat(); // 3 getID(); // 15 for the STORE }
// Set the fields of the shared_dec_mem structure shared_dec_mem.type = 1; shared_dec_mem.address = address; shared_dec_mem.data = data; // and insert into magic_struct member of the message
Send message
set magic_struct with NULL value to simulate an ACK message
Send ACK
// Create an empty message for the fetch
Send empty message

The Decode/Execute receives an empty message from Fetch indicating the presence of updated data: opcode, source and dest in the shared registers of the fetch_registers structure. The format is extracted from the opcode field (in this case is the Format 3) and the same is done for the instruction ID (in this case is 15). The operation corresponds to a STORE immediate, register.

The Decode will set the fields of the shared_dec_mem structure of type memory_message:

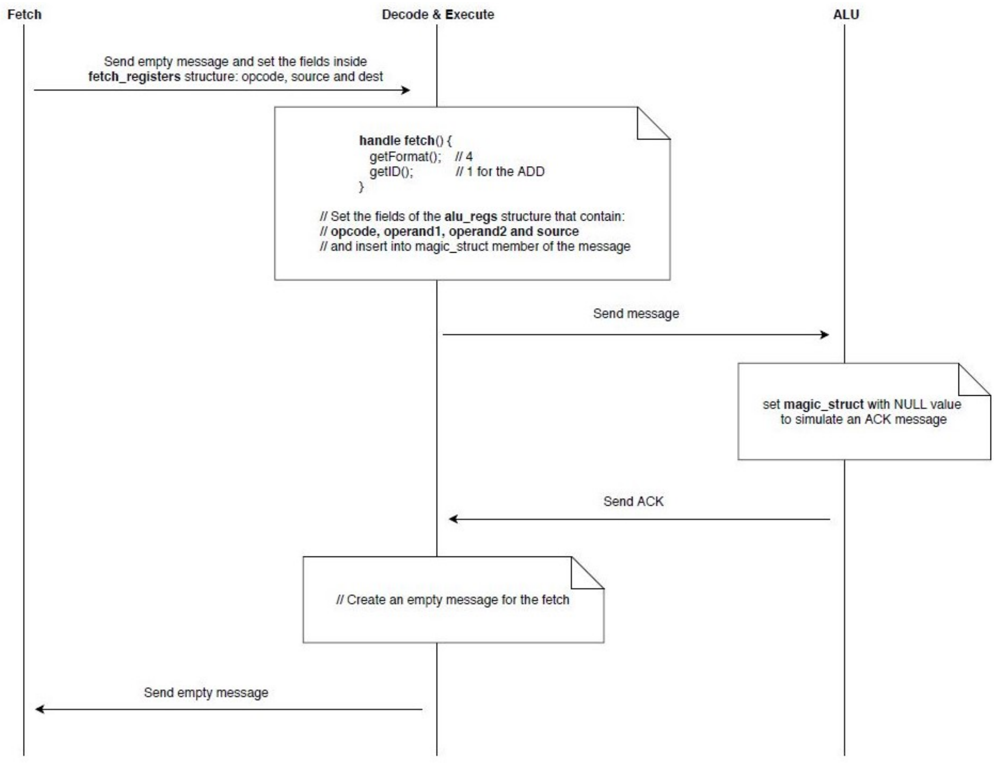
- 1 for the type of operation requested to the memory subsystem (0 for reading, 1 for writing)
- the address of the memory location in which
- the data will be written the data that will be written

After setting the fields, this structure is inserted into the magic_struct field of the message that will be sent to the memory subsystem.

The memory subsystem will write the provided data into the memory at the specified address and will send an empty message (an ACK) to the Decode/Execute to indicate that the operation has been completed.

When the Decode/Execute receives the ACK, it will create an empty message for the Fetch to request a new instruction.

5.4 Format 4 example: ADD register, register


Fetch
Decode & Execute
ALU
Send empty message and set the fields inside **fetch_registers** structure: opcode, source and dest
handle fetch() {
getFormat(); // 4
getID(); // 1 for the ADD
}
// Set the fields of the **alu_regs** structure that contain:
// **opcode, operand1, operand2 and source**
// and insert into magic_struct member of the message
Send message
set **magic_struct** with NULL value to simulate an ACK message
Send ACK
// Create an empty message for the fetch
Send empty message

The Decode/Execute receives an empty message from Fetch indicating the presence of updated data: opcode, source and dest in the shared registers of the fetch_registers structure. The format is extracted from the opcode field (in this case is the Format 4) and the same is done for the instruction ID (in this case is 1). The operation corresponds to a ADD register, register.

The Decode will set the fields of the structure alu_regs type decode_registers:

- **opcode:** it will contain the opcode of the instruction, in this case is 10000001
- **operand1:** it will contain the first operand of the operation, i.e. the content of the source register indicated by the instruction
- **operand2:** it will contain the second operand of the operation, i.e. the content of the register indicated by the instruction.
- **destination_reg:** it will contain the encoding of the destination register, in which the result of the requested operation will be stored.

After setting the fields, this structure is inserted in the magic_struct field of the message that will be sent to the ALU.

The ALU will perform the requested operation and will send an empty message (an ACK) to the Decode/Execute to indicate that the operation has been completed.

When the Decode/Execute receives the ACK, it will create an empty message for the Fetch to request a new instruction.

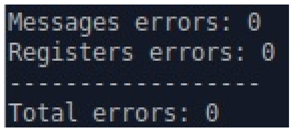
6 Tests

To carry out the test phase, the behaviour of Fetch, ALU and Memory Subsystem modules were simulated, i.e. all those modules that interact with Decode/Execute module.

For the creation of messages and the notification of events to Decode/Execute module, the function getEventList has been created and it returns the event which contains the response message.

```
vector<event*> getEventList(Decode* dec, string&
    source, string& dest, void* magic) {
     message* mex = new message;
     strcpy(mex->source,(const char*)source.c_str());
     strcpy(mex->dest,(const char*)dest.c_str());
     mex->magic_struct = magic;
     mex->next = NULL;
     event ev;
     ev.time = 0;
     ev.m = mex;
    return dec->notify(&ev);
}
```

An example of final output is:



```
Messages errors: 0
Registers errors: 0
------------------
Total errors: 0
```

6.1 Correctness of generated messages

To verify the correctness of messages generated by the Decode/Execute module, some "ad hoc" messages are created using the getEventList function to simulate the behavior of the Fetch module in the initial phase and the same is done for ALU and Memory Subsystem modules, in case of operations that require the intervention of the latter.

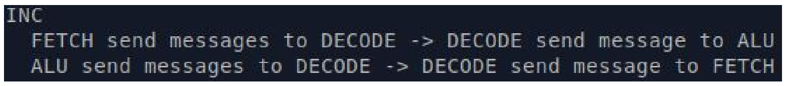
For emulating Fetch behavior, the opcode register is set and a message is created, indicating Fetch as the source and Decode as the destination.

The event that contains the message created as specified above is inserted in the event queue. The getEventList function returns an event containing a message whose destination is checked and must be equal to:

1. Fetch, in case of jump instructions (conditioned or not). <screenshot test>
2. ALU, in case of instructions that require arithmetic operations. <screenshot test>
3. Memory Subsystem, in case of LOAD and STORE instructions. <screenshot test>

In cases 2 and 3 it is also necessary to check that, once the message is received from the module involved in the communication, a new message is created by Decode/Execute module with Fetch as destination to signal the completion of the instruction.

Example of test output:



```
INC
  FETCH send messages to DECODE -> DECODE send message to ALU
  ALU send messages to DECODE -> DECODE send message to FETCH
```

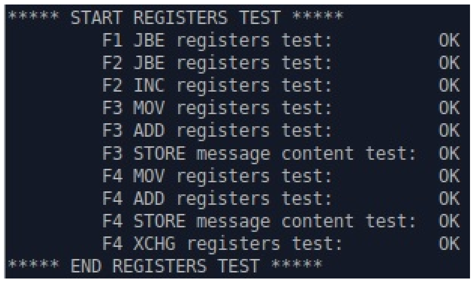
6.2 Correctness of contents of the registers and magic_struct inside message

To test the correctness of the registers it is necessary to simulate the outputs of the modules that interact with Decode/Execute and to evaluate the state of all the registers at the end of the execution.

The test was performed on some representative instructions for each format, such as:

- F1
  - JBE immediate
- F2
  - JBE register
  - INC register
- F3
  - MOV immediate, register
  - ADD immediate, register
  - STORE immediate, register
- F4
  - MOV register, register
  - ADD register, register
  - STORE register, register
  - XCHG register, register

Example of test output:



```
***** START REGISTERS TEST *****
        F1 JBE registers test:         OK
        F2 JBE registers test:         OK
        F2 INC registers test:         OK
        F3 MOV registers test:         OK
        F3 ADD registers test:         OK
        F3 STORE message content test: OK
        F4 MOV registers test:         OK
        F4 ADD registers test:         OK
        F4 STORE message content test: OK
        F4 XCHG registers test:        OK
***** END REGISTERS TEST *****
```

6.2.1 Check registers correctness: ADD F3 example

Before invoking the Decode/Execute module, the following registers have been set to simulate the Fetch module:

- **OPCODE** with ADD of third format
- **Source Register** with an immediate
- **Destination Register** with encoding of BX register (for example)
- **BX Register** with a value

at this point the module uses the function getEventList to perform the operations that consist in setting ALU registers and notifying to the ALU module an event containing an empty message.

In order to ensure the correctness of the operations the following things have to be verified:

- **operand1 of ALU's** registers must be equal to content of the **Source Register**
- **operand2 of ALU's** registers must be equal to content of the **BX Register**
- **destination of ALU's** registers must be equal to the **Destination Register**
- **OPCODE of ALU's** registers must be equal to the **OPCODE**

if at least one of these is incorrect, an error is generated.

6.2.2 Check magic_struct correctness: STORE F4 example

Before invoking the Decode/Execute module, the following registers have been set to simulate Fetch module:

- **OPCODE** with STORE of fourth format
- **Source Register** with encoding of BX register (for example)
- **Destination Register** with encoding of CX register (for example)
- **BX Register** with a value
- **CX Register** with a value

at this point using the function **getEventList** the module performs the same operations done in the previous case.

Decode/Execute and MemorySubsystem modules communicate through a MemoryMessage structure, that is contained in the magic_struct field of a message.

When Decode/Execute receives a message with OPCODE corresponding to the STORE F4, it creates a new message, inserts the information of above registers inside MemoryMessage and sends it to the MemorySubsystem.

In order to ensure the correctness of the created message, the following things have to be verified:

- **MemoryMessage type** must be a write operation
- **MemoryMessage address** must be equal to **BX Register**
- **MemoryMessage data** must be equal to **CX Register**

if at least one of these is incorrect, an error is generated.

7 Results

As shown by the tests carried out, the Decode/Execute module behaves as expected and can be inserted into the simulation of the CA19.32 computer described in this document.

One possible optimization of the project is to use the pipeline principle to perform multiple operations in parallel in order to increase performance.

To implement this solution, the Decode module must be clearly separated from the Execute module. In this way, the Decode can receive a new instruction provided by the Fetch, while the Execute is waiting for the conclusion of the previous operation taken over by the ALU.

8 Bibliography and references

P. Corsini. "Dalle porte AND OR NOT al Sistema Calcolatore, un viaggio nel mondo delle reti logiche in compagnia del linguaggio Verilog". 2015.