Laboratory 1 : CPUSim – Fetch, Decode, Execute

The aim of this lab is to introduce the SimpleCPU computer through the construction of a simulation model of its processor and memory. A block diagram representation of this computer is shown in figure 1. Using a simulation model of this computer, assembly language programs can be executed allowing the user to see the internal steps involved in their execution i.e. the Fetch – Decode – Execute (FDE) cycle, breaking each instruction down into a series a micro-instructions (internal steps). The simulation package we will be using is CPUSim :

"CPUSim simulates computer architectures at the **register-transfer-level** (RTL). That is, the basic hardware units from which a hypothetical CPU is constructed e.g. registers, condition bits, memory (RAM) etc. The user does not need to deal with individual transistors or gates on the digital logic level of a machine. The basic units used to define machine instructions consist of microinstructions of a variety of types. The details of how the micro-instructions get executed by the hardware are not important at this level." – CPUSim User Manual

In addition to simulating the processor's hardware CPUSim also allows the user to define the processor's instruction-set i.e. mnemonics, and then evaluate each instruction's structure: the bit-fields required for each opcode and operand etc. These instruction definitions are then used to automatically create a customer assembler, allowing the hardware and software to be evaluated by single stepping through a program at a machine-instruction level, or at the micro-instruction level. At the end of this practical you will understand how to :

- Define instruction bit-fields i.e. opcode and operands.
- Implement the Fetch-decode-execute cycle on a simple processor.
- Represent a computer's architecture and instructions using RTL descriptions
- Use micro-instructions to perform the immediate addressing mode
- Write a simple assembly language program.

To start the CPUSim simulator left click on the Windows icon in the bottom left of the screen and type :

 `-> cpusim`

This will launch the simulator as shown in figure 2. To construct the simulation model the RTL modules are first defined. Describing a CPU's functionality at this level removes a lot the implementation detail e.g. multiplexers, `ALU` etc, simplifying the CPU's operations to the movement of data between registers (memory elements). The RTL block diagram for this computer is shown in figure 3.

Note, the ALU's functionality (+,−,×,÷) and switching multiplexers (`MUX`) are implicit in the description and therefore no longer shown.

The processor has three main registers:

- Program counter (**`PC`**) : contains the address in memory of the current instruction being processed, used in fetch phase.
- Instruction register (**`IR`**) : stores the current instruction being performed, processed in the decode phase.

- Accumulator (**ACC**) : a general purpose register used to store one of the instruction's operands and any result produced i.e. used in the decode and execute phases.

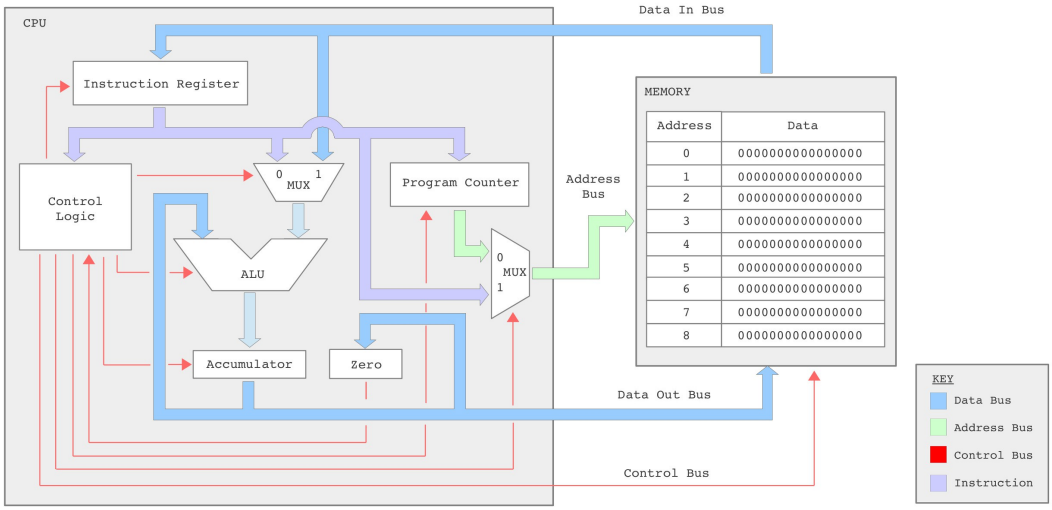


Figure 1 : SimpleCPU block diagram

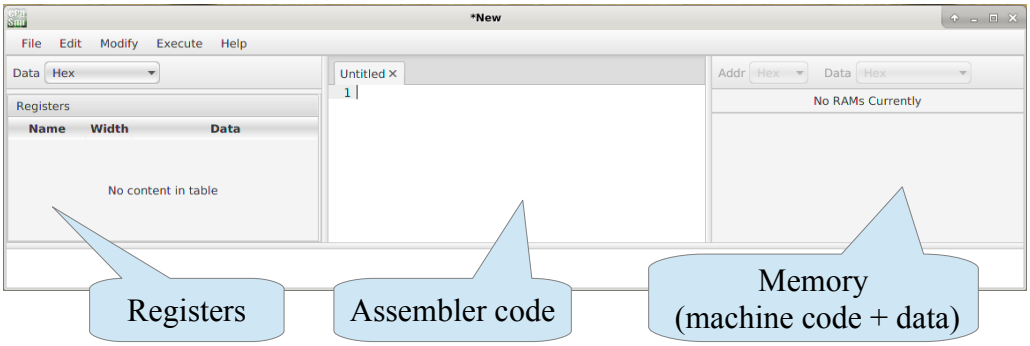


Figure 2 : CPUSim main window

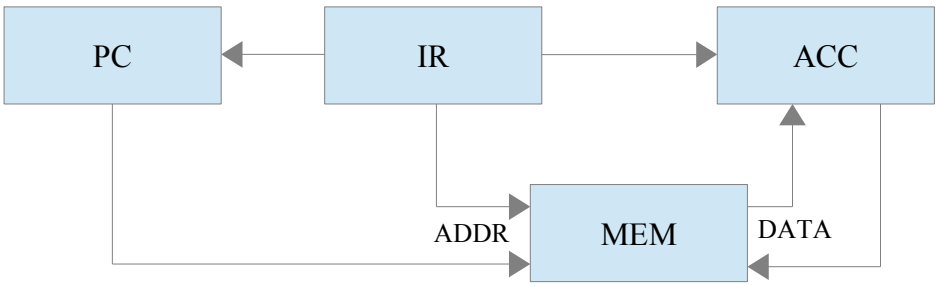


Figure 3 : Register Transfer Level (RTL) block diagram

The interconnection and the flow of data between these registers is indicated by the connecting arrows. Registers that are not connected cannot transfer data e.g. there is no path from the PC to the ACC, therefore, the PC value can not be transferred to the ACC or vice-versa. This can be confirmed if you refer back to the block diagram in figure 1.

Note, the following points below are a little bit of an aside, but I thought it would be useful to explain why this processor's architecture differs from the ones in the two recommended textbooks. These other processors have the following additional registers :

- Memory address register (**MAR**) : contains the address of the external memory locations accessed.
- Memory data register (**MDR**) : contains the data read from or written to external memory.

To keep the SimpleCPU simple (reduce hardware) these are not included in this design. This raises the question why are they included in the other architectures. The main reasons are mainly linked to :

- Functionality : driving the system buses from the MAR and MDR registers de-couples the internal bus from the external bus e.g. driving the address bus from the MAR rather than the PC. At the start of an instruction fetch these registers will contain the same value, but, if we now use a MAR the PC is free and can therefore be incremented at the same time as the fetch. **Note**, looking for operations (micro-instructions) that can be overlapped is a key way of increasing processing performance. The MAR and MDR register can also act as temporary buffers during more complex addressing modes e.g. registser and memory defered which we will be looking at later.
- Bus drivers : from a silicon implementation point of view it takes more power to drive signals off-chip than on-chip e.g. driving the address and data system buses across a mother-board to memory IC's. Having higher powered registered drivers (MAR and MDR) on the actual IO pins helps ensure signal integrity (voltage levels) and also minimises the critical path delay (routing delays), as the signal source is on the edge of silicon, rather than from the middle of the IC. Therefore, maximising external bus clock speeds.

If you have not had the chance to have a flick through the recommended text books I would strongly recommend you do, its always good to get someone else's point of view when looking at any problem, so that you can come to your own.

Task 1

The first step in creating the simulator is to define the processor's registers and memory. To add the registers shown in figure 3 to the simulation model left click on the pull down menu:

```
Modify -> Hardware modules ...
```

This will open the 'Edit modules' window, left click on the pull down menu:

```
Type of Module -> Register
```

Left click the 'New' button at the bottom of this window. This will add a new row to the register table using the default name '?'. Double left click in these text boxes and enter each register's name and data width as shown in figure 4.

Note, the `addr`, `data` and `status` registers were added to overcome limitations in the simulator, they do not exist in the actual hardware. These will be discussed in more detail later in the practical. When entering these registers ensure the different widths are correct i.e. 2, 8 and 16 bits.

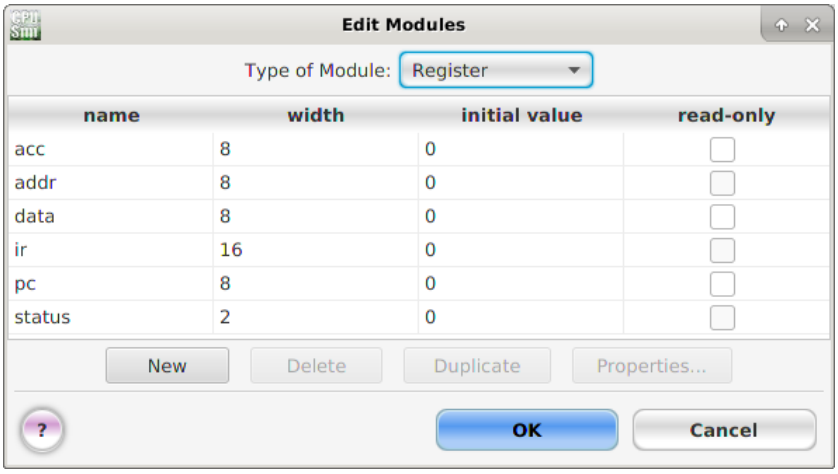


Figure 4 : processor registers

To add condition bits (flags) single left click on the pull down menu:

```
Type of Module -> ConditionBit
```

Left click the '`New`' button and enter the condition bit names and bit positions as shown in figure 5. A condition bit must be assigned to a register, therefore, clicking within the register text box will display a pull down menu of the previously defined registers, select '`status`'. In addition to the normal ALU flags a '`halt`' flag has also been included. When set, this flag signals to the simulator that the execution of the current assembly language program should be stopped.

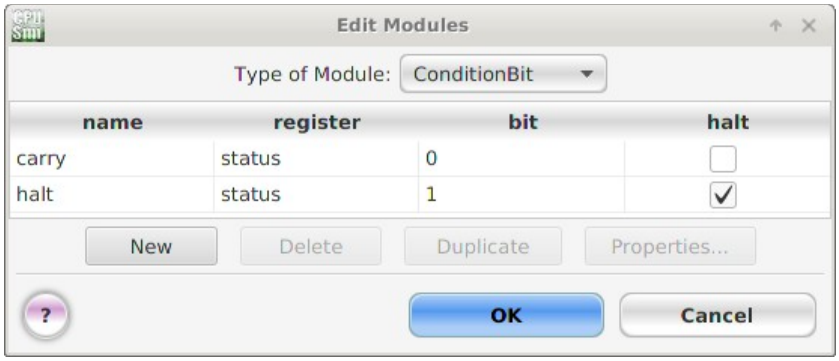


Figure 5 : CPU condition bits

The final hardware module to be defined is the computer's memory, single left click on the pull down menu:

```
Type of Module -> RAM
```

Left click the '`New`' button and enter the memory name and size as shown in figure 6. Left click on the OK button to finish.

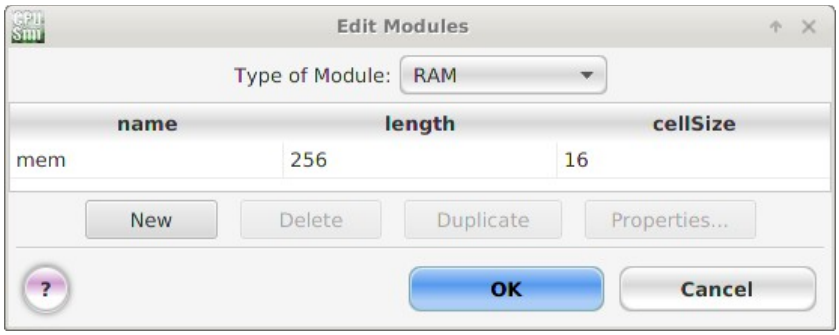


Figure 6 : computer's memory

Task 2

This processor uses a fixed length 16 bit instruction format. Each instruction within the CPU's instruction-set is implemented by a set of micro-instructions i.e. the set of RTL operations that must be performed to implement the desired functionality. Micro-instructions also define the step-by-step sequence of operations needed to perform the fetch-decode-execute cycle.

Note, micro-instructions are internal steps, completely separate from the processor's instruction-set i.e. machine-level instructions. Micro-instructions are not directly accessible to a programmer.

To introduce these ideas consider the simple program shown in figure 7 i.e. the two machine-level instructions `MOVE` and `ADD`. This processor has an accumulator based architecture which uses a 1-operand instruction format, therefore instructions do **not** need to define the second operand or where the result will be stored, as it is always the accumulator (`ACC`), the single general purpose data register.

```
start:
     move 0x01       ;move the value 1 into ACC
     add  0x02       ;add the value 2 to the ACC
```

Figure 7 : Test program 1

Note, computer architectures are sometimes defined by the number of operands used:

- `0-operand` : stack machines, common in early computers to simplify limit memory resources. Location of operands (data) is implicit i.e. always in the same registers (stack), or hard-coded within the processor. An example of a 0-operand instruction for the SimpleCPU would be: `CLR` i.e. clear the `ACC`. This instruction does not need an operand bit-field as there is only one `ACC` and the value is implicit, known i.e. zero.
- `1-operand` : accumulator based architecture i.e. one general purpose register, such as the SimpleCPU. Example instruction formats shown in figure 7 e.g. the `ADD` instruction only specifies one operand, the value 2. The second operand i.e. the current `ACC` value, and destination of the result i.e. the `ACC`, are implicit.
- `2 or 3 operands` : reduced instruction set computers (`RISC`), or complex instruction set computers (`CISC`). These machines have multiple general purpose registers, allowing more flexibility in where operands are stored and

results saved. Later versions of the SimpleCPU processor have four general purpose registers: RA, RB, RC and RD. Examples of a 2-operand and 3-operand instructions would be :

- ADD RA, RB - this instruction adds the data stored in RA and RB, saving the result in RA.
- ADD RA, RB, RC - this instruction adds the data stored in RB and RC, saving the result in RA.

- 3+operands : mostly CISC, could also consider this to include very long instruction word (VLIW) computers. These advanced computer architectures will be considered in later modules, but in general allow improved memory efficiency i.e. smaller code size, less instructions need to be fetched, through the use of increasingly complex instructions, processing multiple operands.

An instruction's operands define where data is stored and results saved. Therefore, the size of an instruction i.e. the number of bits needed to represent it, is dependent on the number of operands processed, the number of instructions and the addressing modes supported by a processor.

Note, a key point to appreciate is that the size of an instruction is largely independent of the types of data it processes. This can cause problems in how instructions and data are stored in memory e.g. the SimpleCPU uses 8 bit data and 16 bit instruction widths, how this difference is handled is discussed later in this practical.

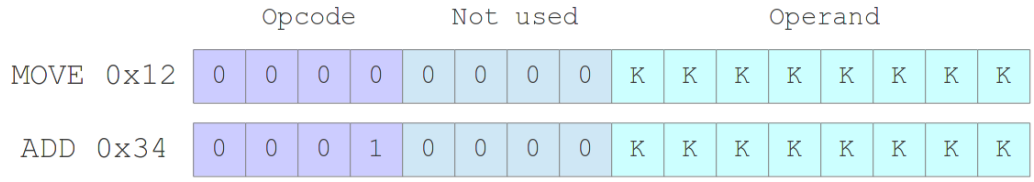


Figure 8 : MOVE and ADD instruction formats

The bit-fields used to define the MOVE and ADD instructions are shown in figure 8. These instructions use the immediate addressing mode i.e. the operand they process is an 8 bit constant (K bits) that is "immediately" available form the instruction register (IR) after the instruction fetch.

To add these new machine-instructions to the simulation model single left click on the pull down menu:

```
Modify -> Machine Instructions
```

This will open the 'Edit machine instructions' window, as shown in figure 9. Before we can create a new instruction we must define the bit-fields used e.g. opcode, not-used and operand, as highlighted in figure 8. To define these bit-fields left click on the 'Edit fields' button.

Next, left click the 'New' button at the bottom of this window. This will add a new row to the fields table using the default name '?'. Double left click in these text boxes and enter each field names, type and widths as shown in figure 10.

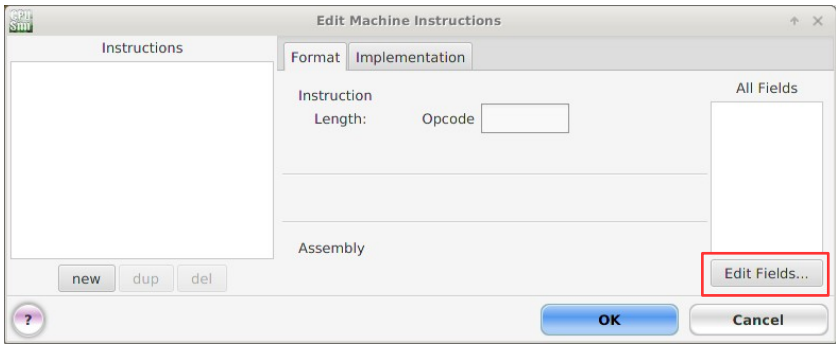


Figure 9 : Edit machine-instructions window

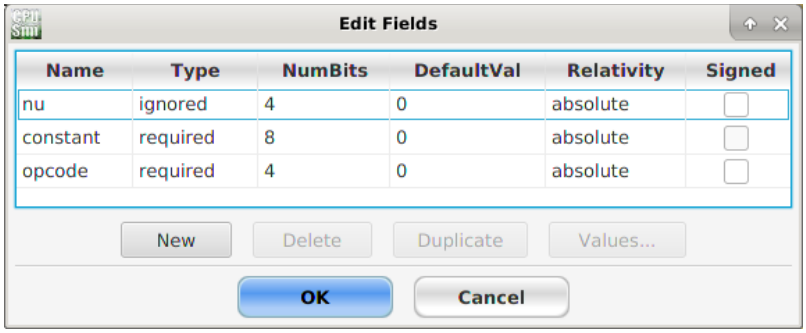


Figure 10 : Edit bit-fields window

When finished click Ok. Within the '`Edit machine instructions`' window left click on the '`New`' button at the bottom left of this window. This will add a new row to the instructions list using the default name '`?`'. Double left click in these text boxes and enter the name `move`, next, left click hold and drag the required bit-fields from the '`All fields`' list into the middle instruction panel, as shown in figure 11. The Assembly panel (bottom) will update automatically.

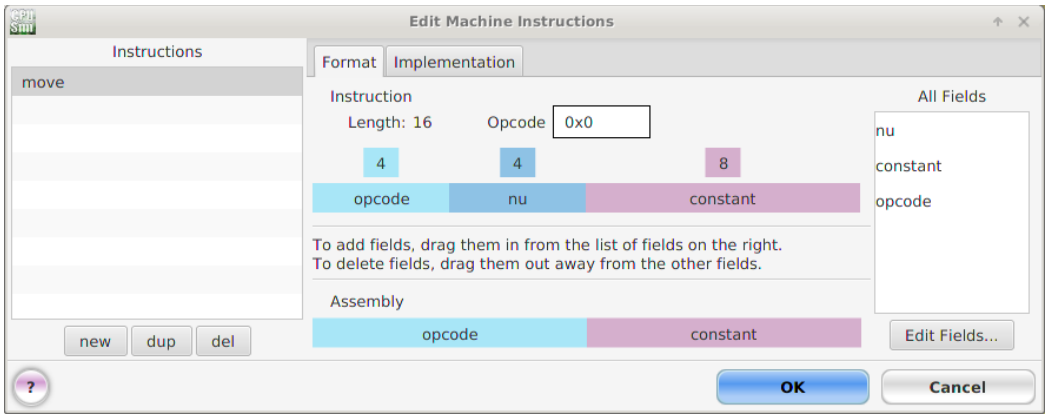


Figure 11 : `MOVE` instruction

Task : repeat this for the `ADD` instruction, its bit-fields and opcode value (0x01) are shown in figure 8. When finish click `Ok` to return to the main window. If you would like to check your solution the `ADD` instruction is shown in Appendix A.

To define what these instruction should do within the simulator we need to define the

step-by-step sequence of operations they perform i.e. their micro-instructions.
To add a new micro-instruction to the simulation model left click on the pull down menu:

```
Modify -> Microinstructions ...
```

This will open the 'Edit Microinstructions' window, single left click on the pull down menu:

```
Type of Microinstruction -> TransferRtoR
```

This first group of micro-instructions are those involved in transferring data between registers. To implement the `MOVE` instruction we need to move the constant `K` from the instruction register (`IR`) to the accumulator (`ACC`).

Left click the '`New`' button. This will add a new row to the micro-instruction table using the default name '`?`'. Double left click in this text box and enter the micro-instruction names and parameters as shown in figure 12.

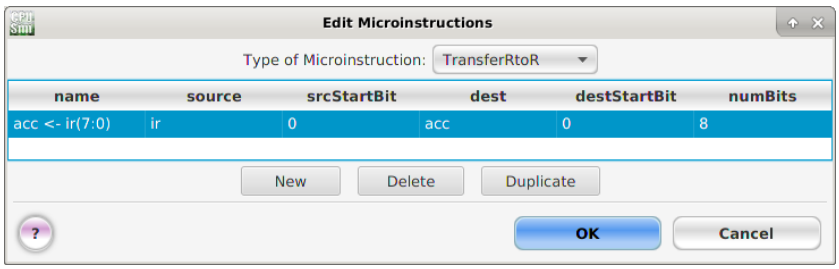


Figure 12 : TransferRtoR micro-instructions

TransferRtoR micro-instruction fields:

- `Name` : purely descriptive text, description of the register transfer level functionality to be performed.
- `Source` : pull down menu allowing user to select one of the user defined registers as the data source.
- `SrcStartBit` : specify starting bit position of the data to be transferred within the source register. Allows micro-instructions to transfer data fields from within larger data words.
- `Dest` : pull down menu allowing user to select one of the user defined registers as the data destination.
- `DestStartBit` : specify starting bit position of the data to be transferred within the destination register. Allows micro-instructions to transfer data fields from within larger data words.
- `NumBits` : specify number of bits to be transferred.

To implement the `ADD` instruction we need to add the constant `K` from the instruction register (`IR`) to the accumulator (`ACC`). Left click on the pull down menu:

```
Type of Microinstruction -> Arithmetic
```

Next, enter the micro-instruction names and parameters as shown in figure 13.

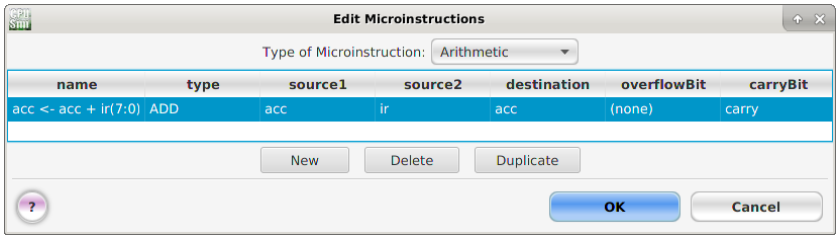


Figure 13 : TransferRtoR micro-instructions

Arithmetic micro-instruction fields:

- `Name` : purely descriptive text, description of the register transfer level functionality to be performed.
- `Type` : pull down menu allowing user to select one of the predefined functions: `ADD`, `SUBTRACT`, `MULTIPLY` or `DIVIDE`.
- `Source1` : pull down menu allowing user to select one of the user defined registers as the first data source.
- `Source2` : pull down menu allowing user to select one of the user defined registers as the second data source.
- `Destination` : pull down menu allowing user to select one of the user defined registers as the result destination.
- `OverflowBit` : used in signed arithmetic to set a status bit (flag) in the event that the result requires more than the 8 bits available in the ACC.
- `CarryBit` : used in unsigned arithmetic to set a status bit (flag) in the event that the result requires more than the 8 bits available in the ACC.

The `MOVE` and `ADD` machine-instruction are very simple and can be executed using the two micro-instructions defined. To enable the processor to load these instruction from memory and process them we need to also define the micro-instructions needed in the Fetch and Decode phases.

To read data/instructions memory left click on the pull down menu:

```
Type of Microinstruction -> MemoryAccess
```

Next, enter the micro-instruction names and parameters as shown in figure 14.

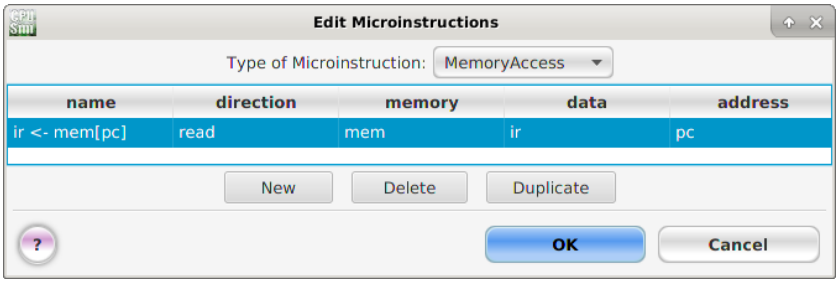


Figure 14 : MemoryAccess micro-instructions

MemoryAccess micro-instruction fields:

- `Name` : purely descriptive text, description of the register transfer level functionality to be performed.

- `Direction` : pull down menu allowing user to select one of the predefined functions: `READ` or `WRITE`.
- `Memory` : pull down menu allowing user to select one of the user defined memory devices.
- `Data` : pull down menu allowing user to select one of the user defined registers as the data source/destination.
- `Address` : pull down menu allowing user to select one of the user defined registers as the address source.

After the completion of the instruction fetch the program counter (`PC`) is incremented to the next instruction address. In the event the program counter overflows i.e. changes from 0xFF to 0x00 (max address is 255), the halt flag is set, signalling to the simulator that an error has occurred i.e. the program has run out of memory. To define the increment micro-instruction, left click on the pull down menu:

```
Type of Microinstruction -> Increment
```

Next, enter the micro-instruction name and parameters as shown in figure 15.

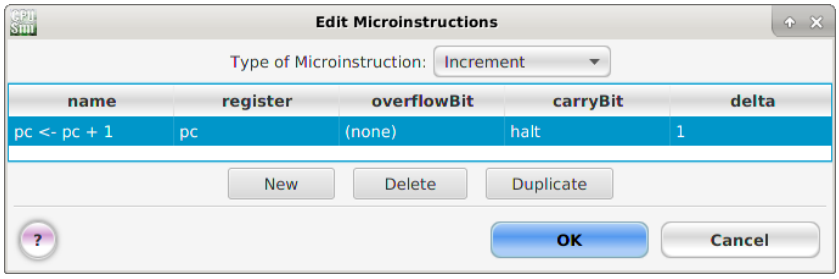


Figure 15 : Increment micro-instruction

The final micro-instruction simply defines what internal register holds the current instruction to be executed i.e. the instruction register (`IR`), left click on the pull down menu:

```
Type of Microinstruction -> Decode
```

Next, enter the micro-instruction name and parameters as shown in figure 16.

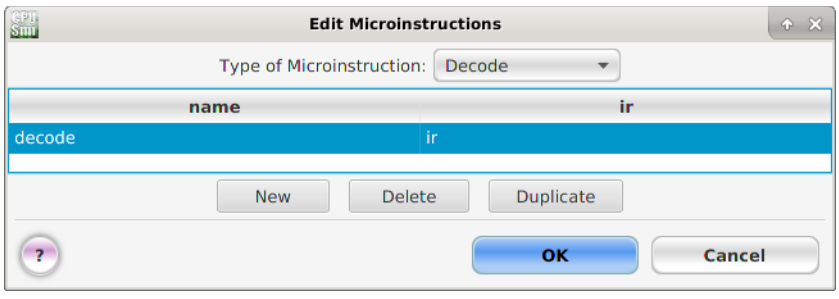


Figure 16 : Decode micro-instruction

Using these micro-instructions we can now define the processor's Fetch – Decode – Execute (FDE) cycle at the register transfer level. To add these operations to the simulation model left click on the pull down menu:

`Modify -> Fetch sequence ...`

This will open the '`Edit Fetch Sequence`' window, expand the micro-instruction category folders by clicking on the ▶ icon. Then left click, hold and drag the required micro-instruction into the '`Fetch Sequence Implementation`' panel, as shown in figure 17.

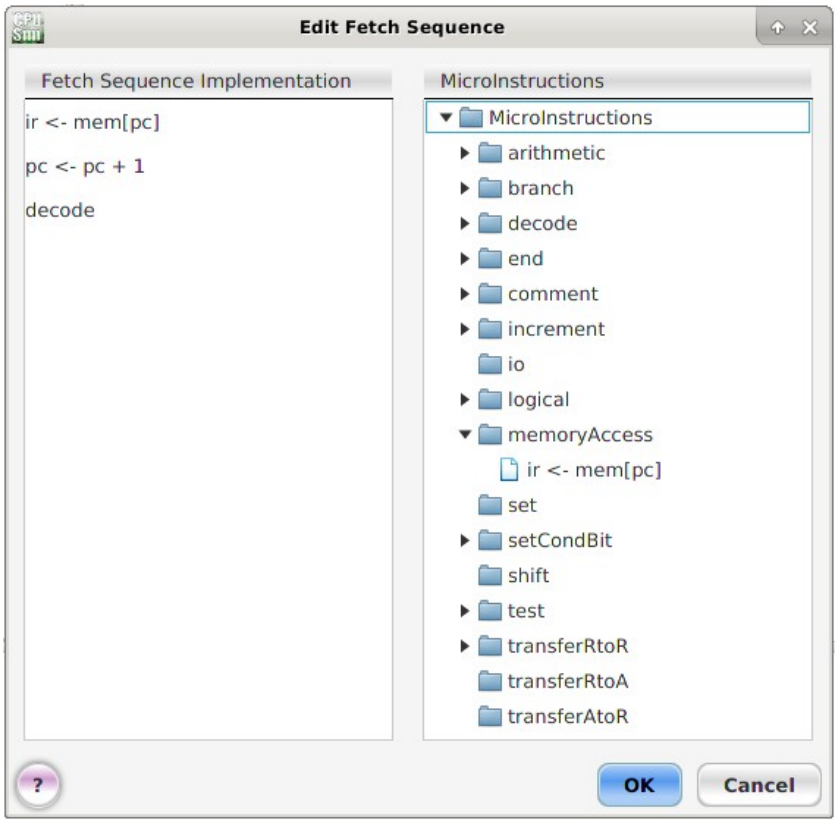


Figure 17 : Fetch-Decode sequence

Note, if you insert the wrong micro-instruction, left click, hold and drag that micro-instruction out of the sequence panel back into the micro-instruction panel. If needed you can also changing the order that the fetch sequence is performed by again left clicking, hold and dragging the micro-instruction up or down the sequence list.

To define the micro-instructions used by each machine-instruction left click on the pull down menu:

`Modify -> Machine Instructions`

This will open the '`Edit machine instructions`' window, left click on the '`Implementation`' tab. Next, expand the micro-instruction category folders by clicking on the ▶ icon. Then left click, hold and drag the required micro-instruction into the '`Execute Sequence`' panel, as shown in figure 18.

Note, micro-instruction can be moved or removed as previously described. All machine-instruction sequences must finish with the predefined `end` micro-instruction to terminate the FDE cycle.

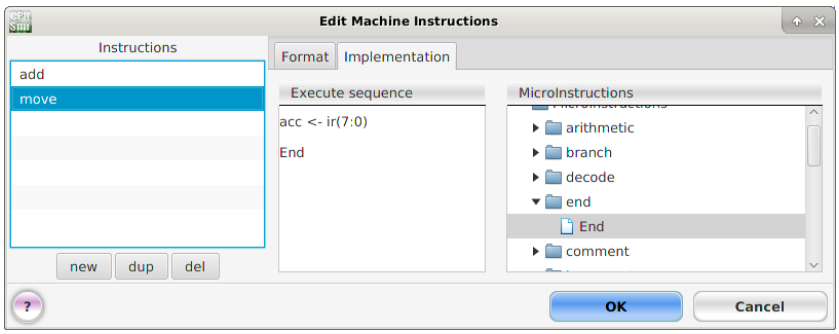
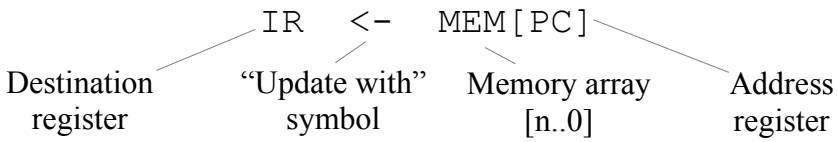


Figure 18 : Adding `MOVE` micro-instructions

Task : repeat this for the `ADD` instruction, when finish click `Ok` to return to the main window. If you would like to check your solution the `ADD` instruction is shown in Appendix A.

Note, each micro-instruction name describes the function to be performed using the standard register transfer syntax :



A high level program is broken down into a series of machine-instructions, these in turn are performed using the Fetch-Decode-Execute cycle, which in turn are implemented inside the computer as a series of micro-instructions.

Figure 19 : test program

Task 3

Now that the test program's instructions have been defined you can write, assemble and execute the program shown in figure 7. To enter a new program left click on the pull down menu:

```
File -> New text
```

This will open a new edit window as shown in figure 19. Enter this program, then left click on the pull down menu:

```
File -> Save text
```

Save this program to your working directory using the file name `test.a`. Instruction names are highlighted in **GREEN** indicating that a matching machine code instruction has been found by the assembler, otherwise there is a syntax error.

During a simulation you can set breakpoints i.e. highlight instruction that will cause the simulation to pause. This requires the simulator to know which register is used as the program counter, this is done by left clicking :

`Execute -> Options ...`

Then click on the breakpoints tab, selecting Breakpoints and update the program counter pull-down menu, as shown in figure 20.

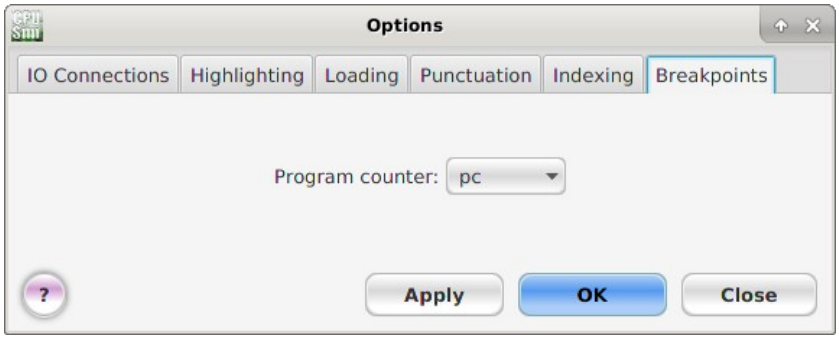


Figure 20 : Breakpoint setup

To allow you to single step through the program at the micro-instruction level, left click on :

`Execute -> Debug Mode`

To assemble and load this program left click on the pull down menu:

`Execute -> Assemble & load`

If there are no syntax errors, this will update the memory panel with the machine code values of the instructions. Change the number base of each window to hexadecimal, using the Address and Data pull down menus, as shown in figure 21.

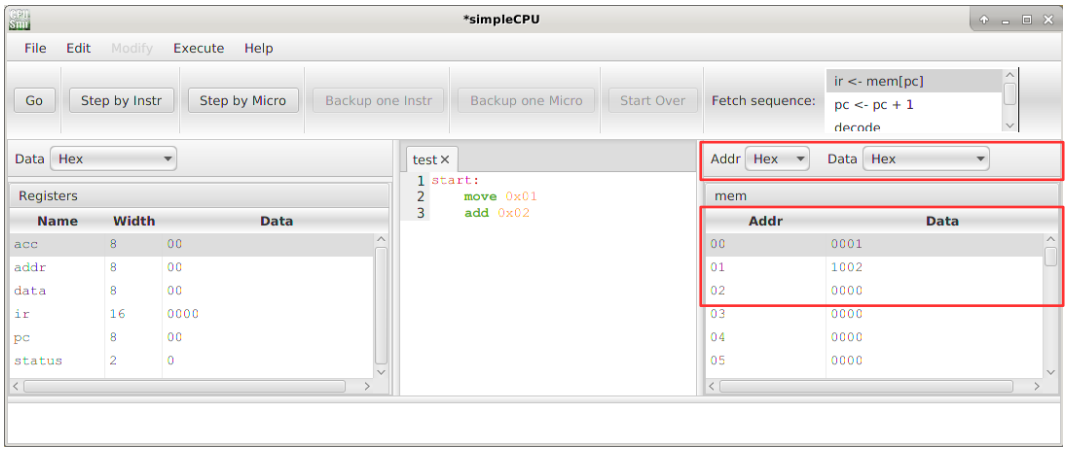


Figure 21 : RAM and register display windows

Note, the first instruction fetched by the CPU is always from memory location 0 i.e. when the simulation is reset the `PC` is reset to 0.

Single left click on the '`Step by Micro`' button to view the different phases of the

Fetch – Decode –Execute cycle and the micro-instruction used to implement them. This will highlight the micro-instruction currently being performed in the top right panel and highlight registers that are updated in the left register panel.

Task : repeatedly left click on the '`Step by Micro`' button to execute the `MOVE` and `ADD` instructions. If the program has been implemented correctly the final result stored in the `ACC` should be 3 and the `PC` register will be updated to 2. What will the computer do if you try execute the "instruction" at address 2 i.e. an "empty" memory location? Left click on the '`Step by Micro`' button to confirm your answer.

Note, to re-enable the greyed out `Modify` task bar pull-down menu unselect the '`Debug Mode`' option.

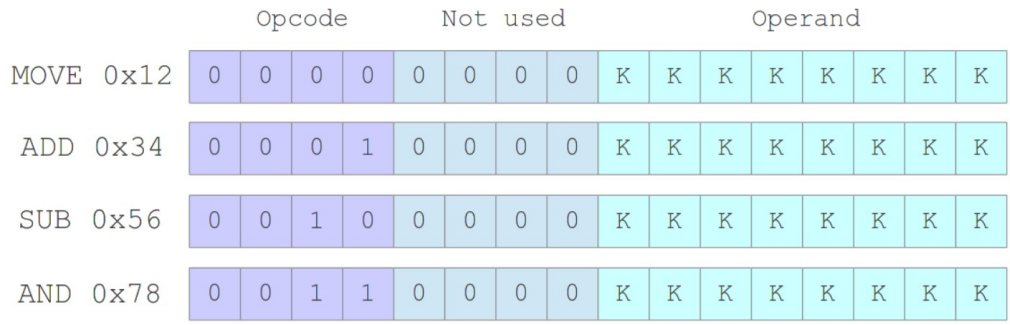


Figure 22 : Immediate addressing mode instructions

```
start:
     move 0xFF      ;move the value 255 into ACC
     sub  0xAA      ;subtract the value 170 from ACC
     and  0xBE      ;bitwise AND ACC with the value 190
```

Figure 23 : Test program 2

Task 4

The immediate addressing mode instructions supported by the SimpleCPU processor are shown in figure 22. **Note**, to help machine-code "readability" the top two bits of the opcode field for an instruction using this address mode is always "00".

Task : implement the `SUB` and `AND` instructions. If you would like to check your solutions the `SUB` and `AND` instruction are shown in Appendix B and C.

Note, you can only perform logical micro-instructions upon registers of the same size. Therefore, to implement the `AND` instruction you will need to implement another register-to-register transfer instruction to load the lower 8 bits of the instruction register into the `data` "register", this can then be used as an operand source for the bitwise AND micro-instruction. In the actual hardware this "register" is simply implemented using wires, selecting bits `IR(7:0)`, as no memory functionality is required i.e. its a simple bit-slice operation.

Task : create a new program to implement the test code shown in figure 23. What logic function does subtracting a value from 255 implement e.g. 255-170? What is the final result? Run this program through the simulator to confirm your answer.

Hint, perform the subtraction 255-170 in binary using pen and paper, compare the result to the binary value for 170, how do they differ?

Task 5

As shown in task 3 if we do not tell the processor to stop, it will continue to fetch instructions even if the program has finished e.g. fetching an "instruction" from an empty memory location will cause the processor to execute the instruction `MOVE 0` as the data stored has the same bit pattern 0x0000 as this instruction, as shown in figure 24.

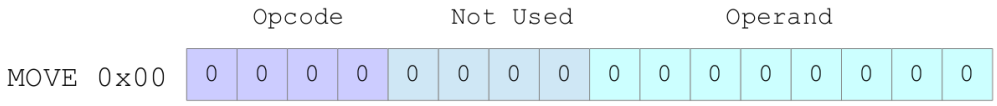


Figure 24 : MOVE 0 instruction

If a processor was to incorrectly execute data as if it were instructions this could cause the processor to crash, therefore, we need a way to halt the fetch-decode-execute cycle. To stop the processor we can define a new 0-operand instruction: STOP, as shown in figure 25. This instruction does not require an operand bit-field as its implicit what this instruction needs to do i.e. set the halt bit in the simulator.

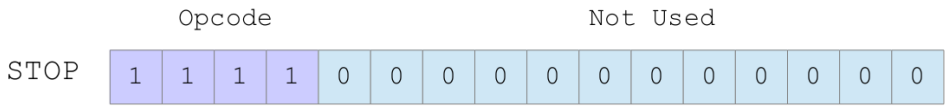


Figure 25 : STOP instruction

Task : implement the `STOP` instruction. If you would like to check your solution screenshots of the `STOP` instruction are shown in Appendix D.

Hint, this instruction uses a `Set` micro-instruction to set the `halt` status bit in the the status register. When this flag is set to `TRUE` i.e. a logic '`1`', the simulator will stop. There are two flags in the status register, make sure you set the correct bit. A new instruction bit field must be defined as the `STOP` machine-level instruction is not passed an operand, as shown in figure 26. The 1-operand instructions are passed the values 0x01 and 0x02 (orange text). 0-operand instruction are not.

Figure 26 : STOP instruction

Task : update either test program to include the `STOP` instruction, as shown in figure 26. Confirm that the simulator is stopped, a HALT message will be displayed in the bottom panel, as shown in figure 27.

IMPORTANT, when you have completed this task don't forget to save this processor model as we will be continuing its development in the next laboratory. To save this

machine left click on :

`File -> Save Machine`

Then enter the file name : `simpleCPU.cpu`. The assembly language programs used can also be save by selecting the `Save Text` option from the `File` menu.

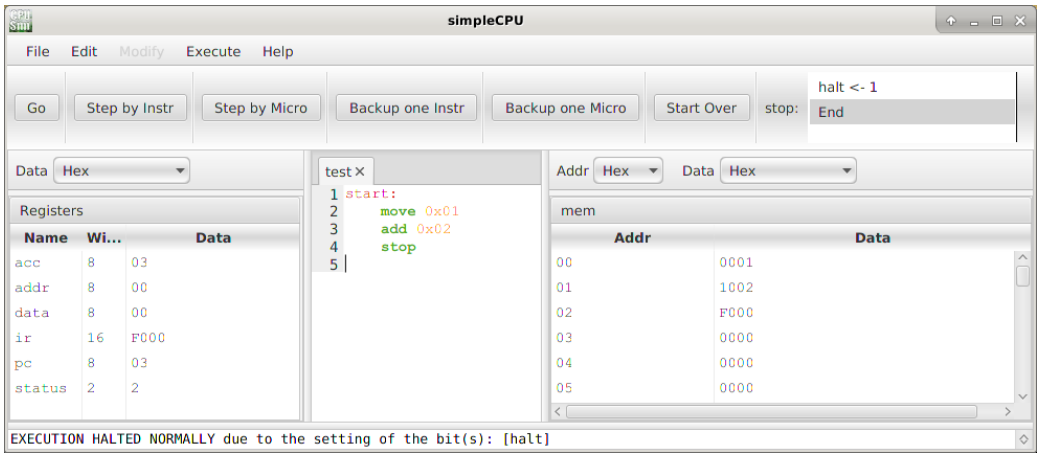


Figure 27 : halting the simulator

Summary

The SimpleCPU computer is based on a Von Neumann architecture i.e. a stored program computer, using one memory that contains both instructions and data, as shown in figure 28.

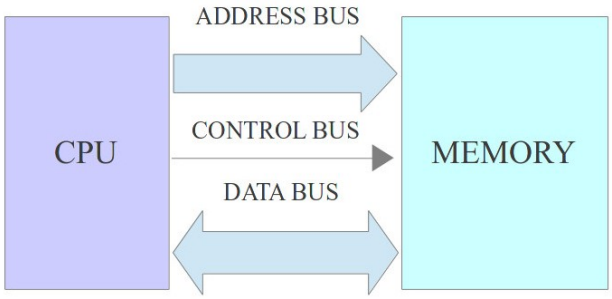


Figure 28 : Von Neumann architecture

This causes a conflict regarding a memory location's width i.e. how many bits are stored in each location, as instructions are 16bits and data is 8bit. The number of addressable memory locations is fixed by the address bus, in this case 8bits (256 location). This gives the designer a few choices:

- RAM 256 x 8bit : each memory location stores 8bits, therefore, an instruction has to be split over two memory locations. As a result the processor's fetch phase will now have to read two memory locations to retrieve the high and low byte of the instruction. The advantage of this approach is that the processor can store data to any memory address.
- RAM 256 x 16bit : each memory location stores 16bits, therefore, an instruction can be fetched in a single memory transaction. However, as the ALU and ACC are 8bits there is now a problem of how an 8bit value is stored

in a 16bit memory location i.e. how can you update one byte without affecting the other. There are ways to achieve this, but, they would require specialised instructions and additional hardware. The simple solution taken in this processor is to store data in the lower byte and pad the higher byte with 0x00. This means that memory locations used to store data will now waste 8bits, as the CPU can only read data from and data write to the low byte.

- RAM 128 x 16bit : memory is now byte addressable, each memory location stores a 16bit values, however, the processor can also read/write to either the high or low byte. Therefore, the effective address bus is reduced to 7bits (128 x 16bit locations) as making the memory byte addressable means the processor still has to specify 256 x 8bit memory locations. Implementing this in hardware is a little tricky. The simplest solution is to have two 128 x 8bit memories connected in parallel, one storing the high bye, the other the low byte, as shown in figure 29. Address bus lines A7 - A1 are used as the "address bus", address line A0 is used to select which memory should be accessed when performing data byte read/writes e.g. A0=0: low byte, A0=1: high byte. With a little bit of glue logic this also simplifies byte writes as the unused memory device can simply be disabled i.e. when writing to the low byte the high byte must not be altered. Instruction fetches can be performed in a single transaction, but instructions are now aligned to even byte addresses e.g. at addresses 0,2,4,6,8 ...

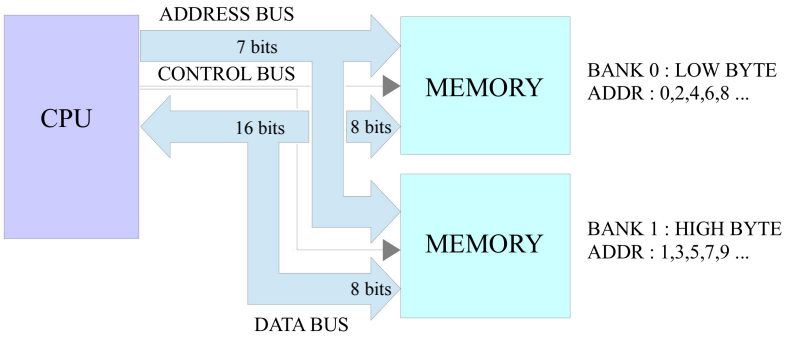


Figure 29 : 128 x 16bit memory architecture

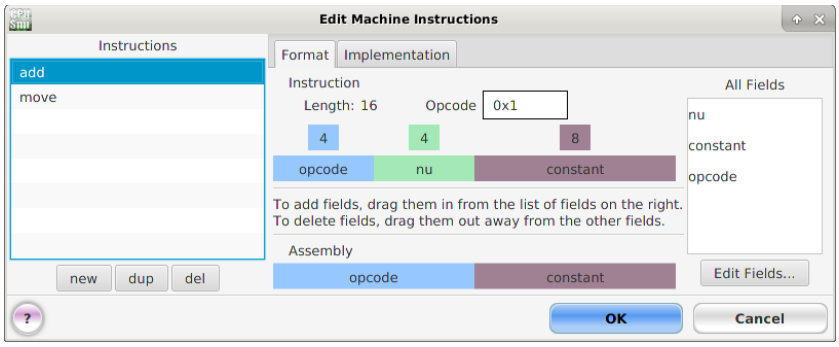
Memory architectures have a significant impact on a computers processing performance e.g. how many memory transactions are required to fetch an instruction. However, when designing a system you also need to consider memory efficiency and hardware costs, so selection is typically a compromise between these factors.

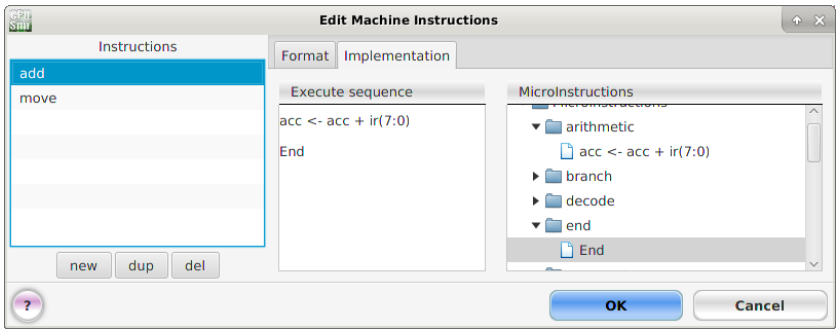
The operations performed by a processor's instruction-set are commonly defined using the RTL syntax. These define the internal micro-instructions used to implement the machine-level instructions used by the program e.g. the immediate addressing mode. These micro-instructions are also used to implement the fetch-decode-execute cycle that realises the stored program processing model.

You can download CPUSim software for home use and additional information on this simulator from:

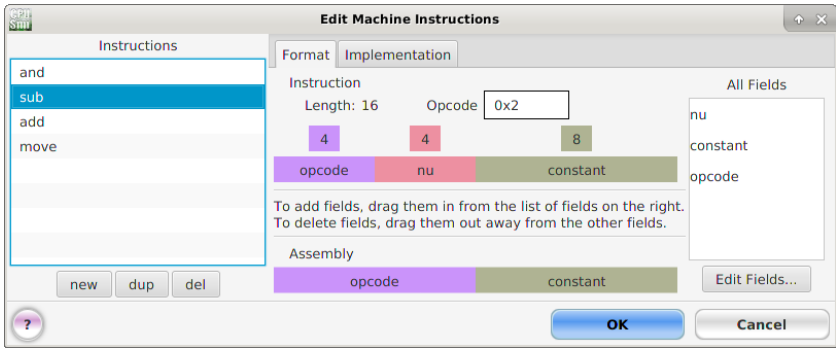
http://www.cs.colby.edu/djskrien/CPUSim/

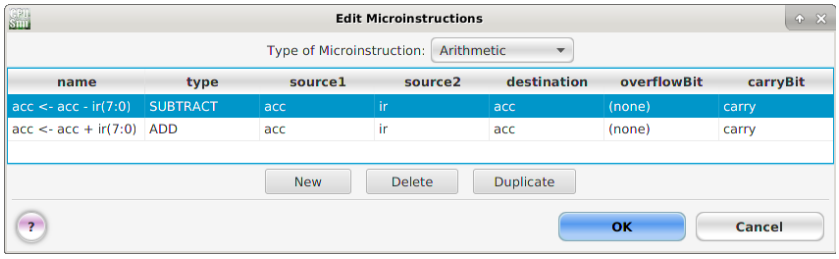
Appendix A : ADD instruction

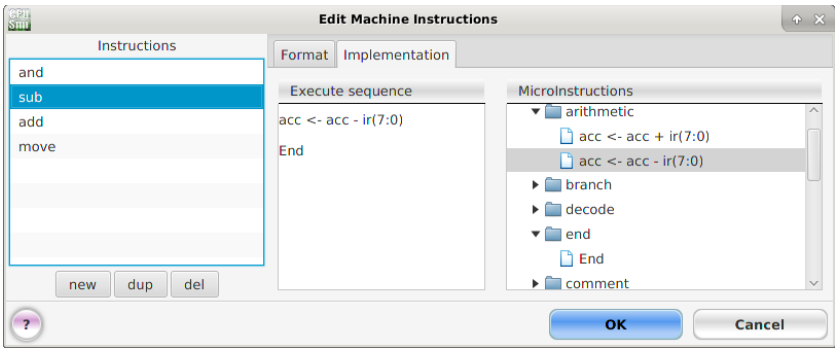




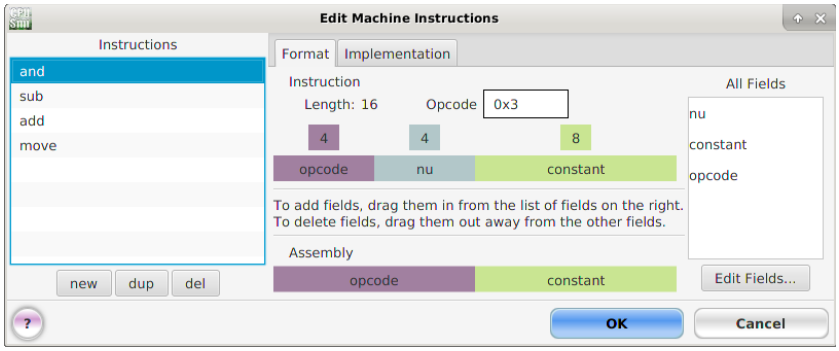
Appendix B : SUB instruction

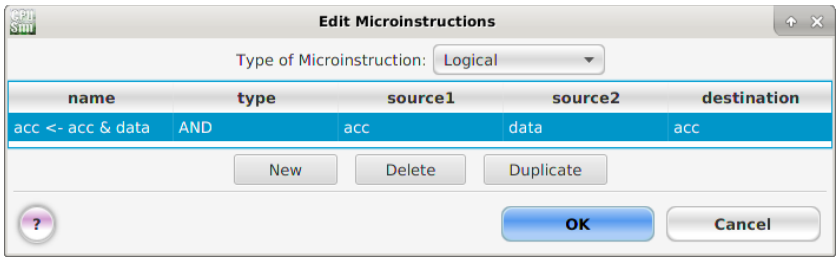


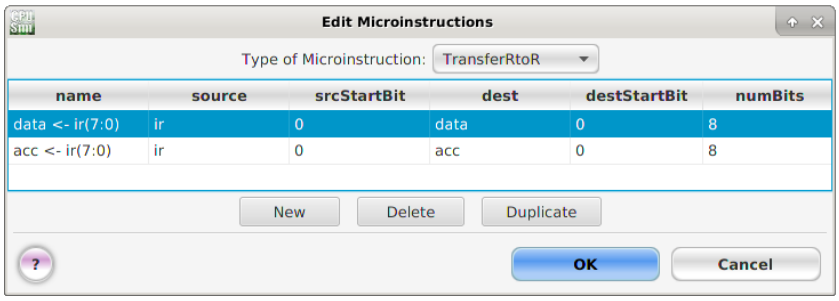


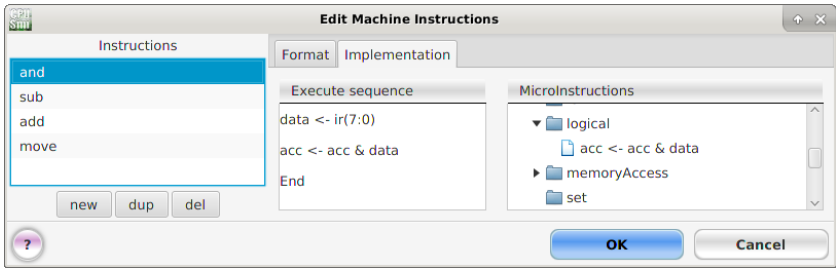


Appendix C : AND instruction









Appendix D : STOP instruction