Introduction

In this document, we will go through some simulation to see the functionality of the subblocks for better understanding their functions. Then we will be introduced about 2 full programs to see how the whole system works.

Run simulation with command:

```
run.csh <vector>
```

The output of simulation:

```
compile.log      # The log file for the last compilation
sim_<vector>.log # The simulation log file for the vector
<vector>.fsdb    # The waveform of the simulation
```

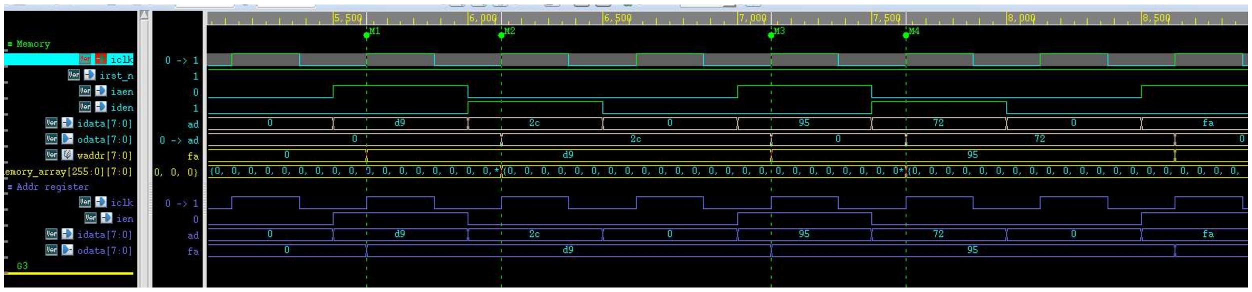
When verifying the submodules, the testbench will try to "force" the inputs so that it can monitor the output without the need of activating all other blocks.

Memory

In this test, we try to verify the memory by writing and reading data. Through this test, we can also verify the functionality of the `common register` module.

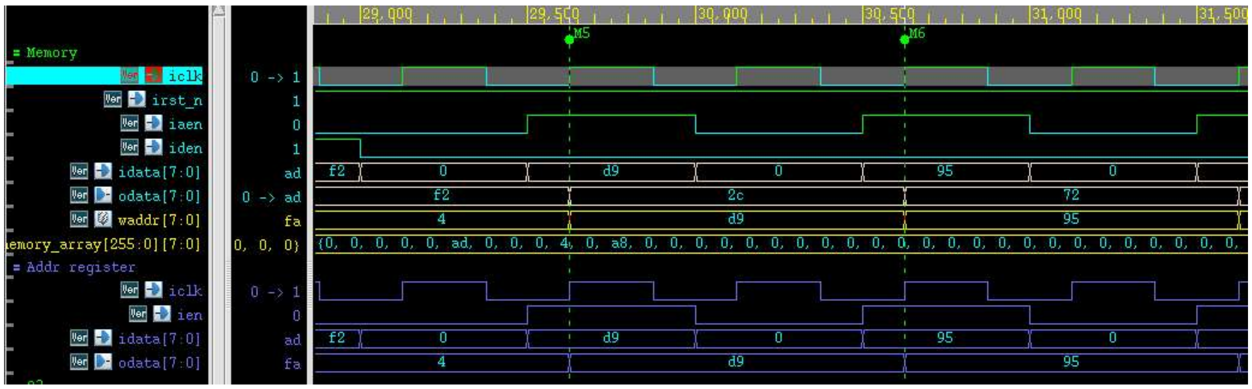
- Randomize data & address 16 times, then write to the memory
    - In each step, enable address register to capture the address. (We assure that there is no duplicate address in this step)
    - Then enable data to capture the data.

- Read all 16 addresses in previous step and compare with the written data to see if it is stored correctly or not.
    - In each step, enable address register to capture the address.
    - Compare the data on the memory's output with the written data.



In the figure above, M1 & M3 are where we store the address. The signal `waddr` (yellow color) is the output of the address register. Whenever the signal `iaen` asserts, the signal changes to the value of `idata`. The blue signals are the ones from Adress register (which is common module).

The markers M2 & M4 are the time we store data into the memory array (yellow color). The array updates value whenever `iden` signal asserts.



In above figure, the output of the memory `odata` updates whenever the address register gets new value in M5 & M6. And the output's value is what we written previously on M2 & M4

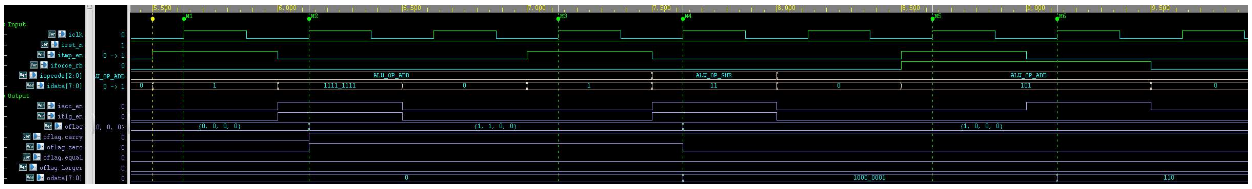
ALU block

In this test, randomize will be hard to show the functionality of the block. Thus, the testbench use direct test method:

- ADD 255 & 1. Expect to see result 0 with flags carry 1, zero 1
- Right shift 3. Expect to see result b1000_0001 with flag carry 1. The previous step creates carry flag to be 1, thus it affects the result of this step.
- ADD 5 with force RB to be 1. We expect the carry in previous step doesn't affect this step because it should be AIR calculation; the output increase 1 no matter what value RB is.

The steps to run this testbench follows CPU's order:

- Enable `itmp_en` to latch RB data (data is on `idata`)
- Change `idata` to RA value; apply `iopcode`, enable `iflg_en` & `iacc_en` to capture the output & flags.

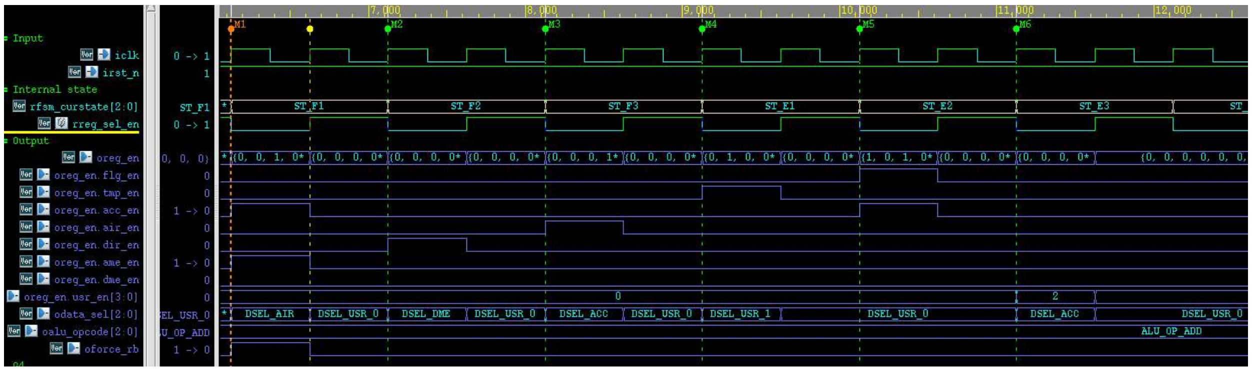


In the figure above, we don't drive `iflg_en` in M6 because this is special ALU operation (calculate AIR), we don't update flag on this mode.

CPU

ALU command

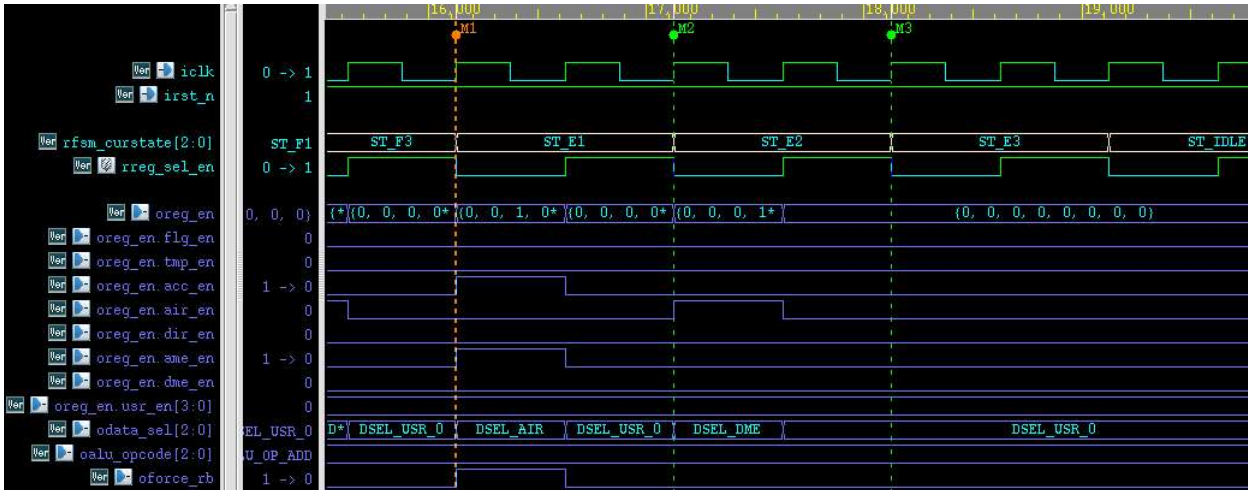
In this test, we assign ALU ADD instruction into the CPU and monitor the operation on the outputs.



- M1-M6 mark the states F1-E3 as designed. Each state requires 2 clock cycle to complete, and we only care about the first one, because the second one is used for registers to capture the value.
- Fetch states (M1-M3):
    - ALU is in ADD operation with forcing add 1. The data select is AIR by the signal `odata_sel`.
    - Then DIR & AIR are enabled to capture data from memory and accumulator (watch the value of `odata_sel`).
- Execute states (M4-M6):
    - First RB is selected (in this test, it is R1) and the ALU's temporary register is enabled.
    - Then RA is selected (R0), ALU is in ADD operation, accumulator & flag are enabled.
    - The result in accumulator is selected by `odata_sel`, then RB is enabled (see value of `usr_en`)

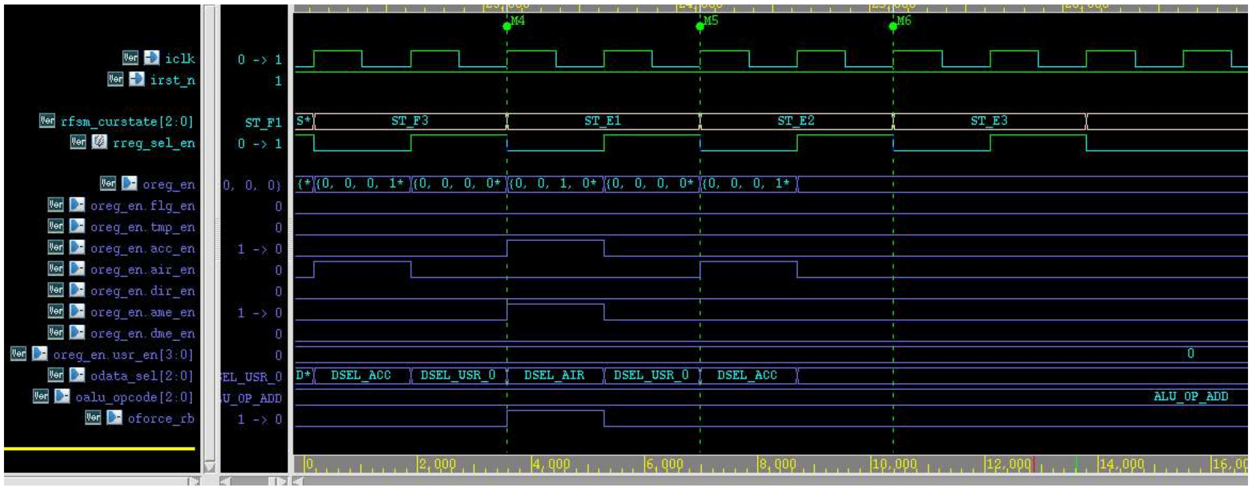
JUMP command

The JUMP commands require 2 memory addresses instead of 1, and the final value of AIR depends on the flags' comparison.



The Fetch states stay the same across instruction, below are the Execute states when flag matching is good:

- AME is enabled at `ST_E1` (see `ame_en`), the data is from AIR that is calculated in the previous state. In this state, the signal `acc_en` doesn't affect if the flag is matched.
- AIR captures the data from memory in `ST_E2`.
- CPU doens't do anything in `ST_E3` for this instruction.



In the case the flags are not matched, CPU change the source of data in `ST_E2`. The port `odata_sel` now select accumulator block instead of memory. The accumulator gets the new value in `ST_E1`, where the ALU is forced ADD 1.

Full program

In this section, we will go through some program to see how the whole system works. Simulation will need to use *force* ... *release* syntax to write the program into the memory before enabling the CPU.

Multiplication

Multiplication program

Multiplication in binary is just like in decimal, example:

```
  00001011
* 00000101
__________
  00001011
00001011
__________
  00110111
```

To make it to be a program, first we put the 2 numbers into 2 variable V1, V2, and the result is V3. The step will be:

- Step 1: Check LSB of V2, if it is 1, then V3 = V1 + V3
- Step 2: Leftshift V1, Rightshift V2
- Step 3: Repeat from Step 1 until we shift all bits.

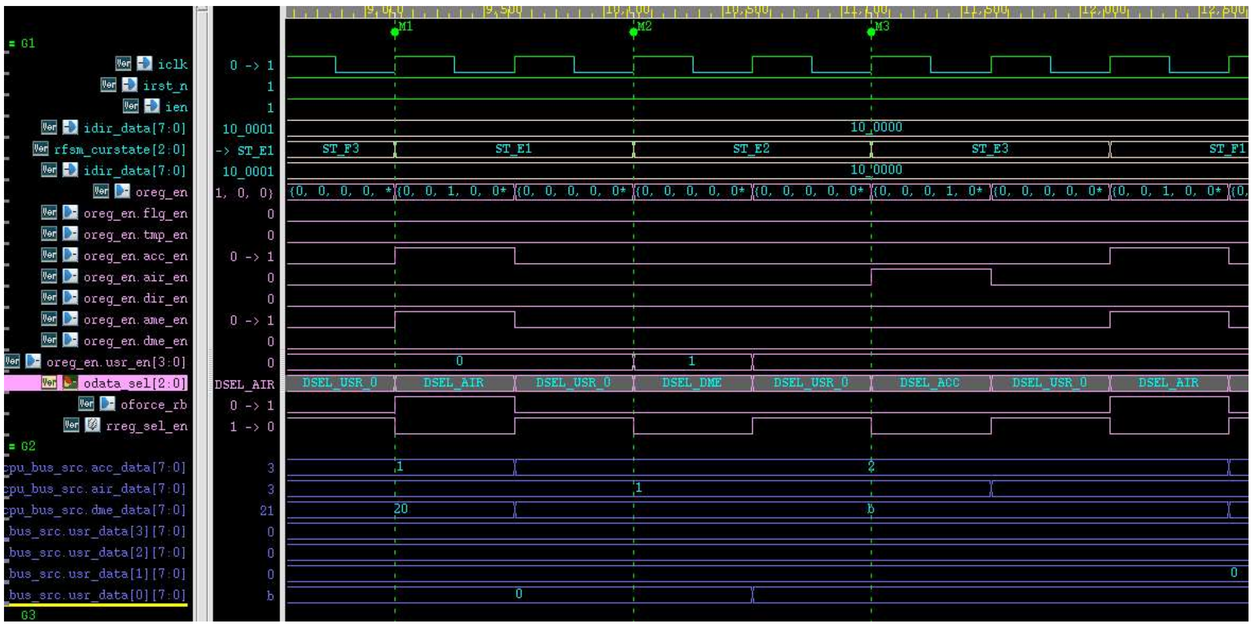
From above algorithm, we use R0, R1 and R2 to be the 3 variables. We also need R3 to be the bit counter, that make sure we only shift 8 bits. In this program, we do a little different by doing Rightshift V2 in step 1, so that we can check the value by using carry flag in the ALU.

Program: Multiply

Multiplication simulation

The first command of the program:

```
DATA R0 11  # First number
```



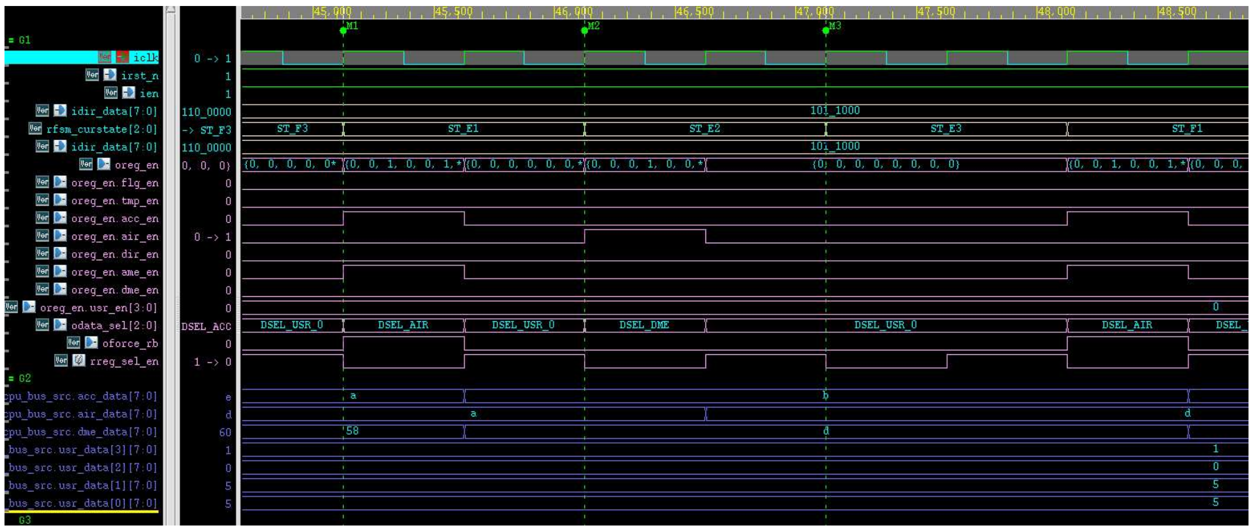
The DATA command executes sequence:

- ST_E1:
  - AIR value is 1 from ST_F3, AME is enabled to get the data on the next address (odata_sel=DSEL_AIR)
  - ACC is forced ADD to prepare new AIR value
- ST_E2:
  - R0 is enabled to capture data from memory (usr_en=1, odata_sel=DSEL_DME)
  - R0 value becomes 11
- ST_E3:
  - AIR is enabled to capture ACC value (odata_sel=DSEL_ACC), this makes the pointer points to the next command.

The same goes for the next lines of the program.

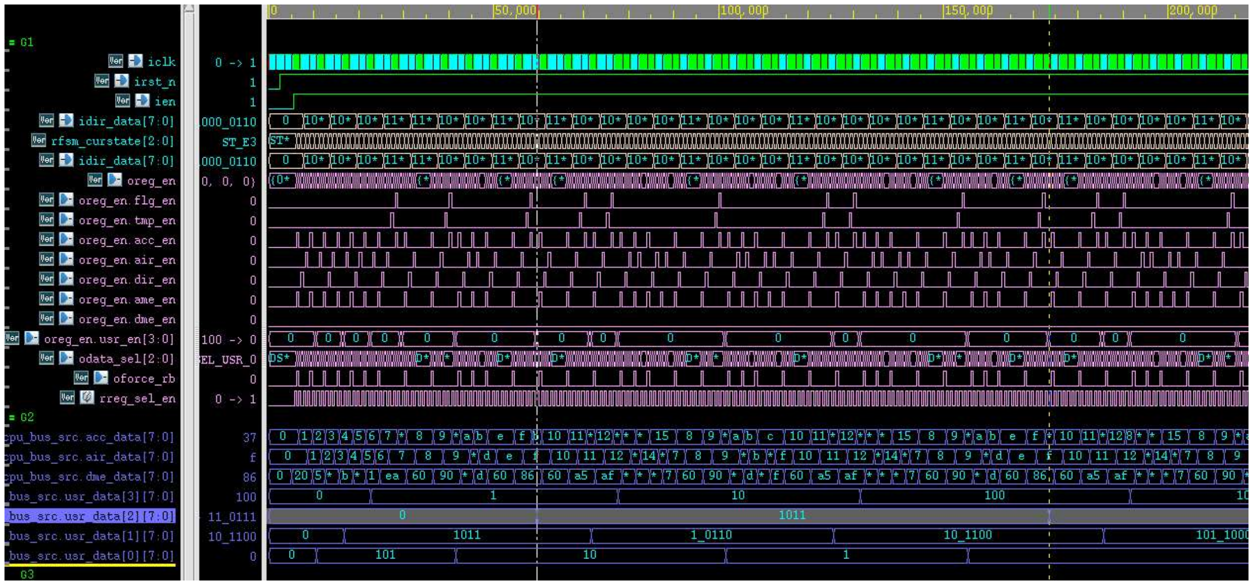
```
JC   L_MUL  # If last bit is 1, add
```

By the first time this command arrives, R0 is 0b101, thus the right shift command makes carry flag to assert. The AIR at this time is expected to capture data in the next memory address.



- ST_E1:
  - AIR is holding value from the state ST_F3, AME is enabled so to get data on that address.
  - ACC is not used in this case.
- ST_E2:
  - AIR is enabled to capture data from memory
  - By this time, AIR "jumps" to another address rather than increases 1 step like normal commands.
- ST_E3: Idle state

During the program runs, R2 updates value twice. The first time is b1011, the second time si b110111 (55 in decimal). Which is expected value of the program.



Fibonacci sequence

The Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones.

In our CPU, the ADD command adds RA and RB, then store the data into RB:

- We always need bigger number is in RA, because that information is used for the next ADD command.
- However, the result is stored in RB, which is the bigger value after the command.

Solution:

- RA ADD RB
- Store RA and RB into 2 addresses
- Swap RA and RB by loading the 2 addresses in to RB and RA
- Repeat the ADD command until carry flag asserts

Program: Fibonacci