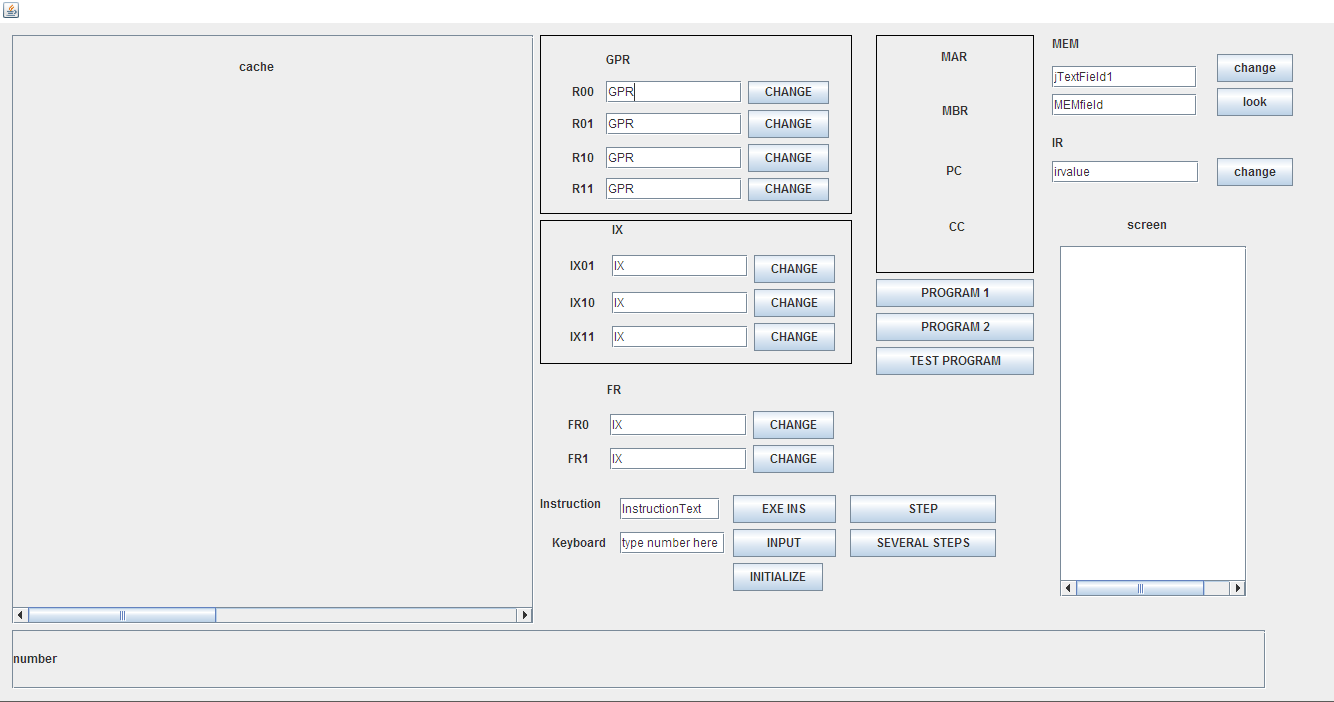
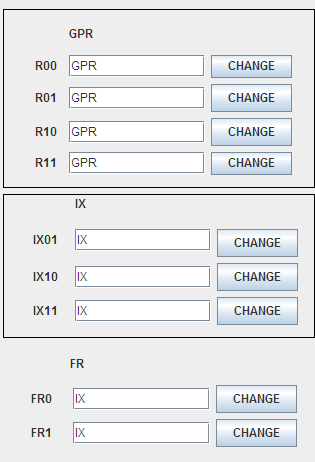
Part 4 of our project introduction

In this part, we implement the float/vector instructions and a simple pipeline in our simulator. Here is the guide to use our simulator and test our new instructions.

1. Simulator



This is our simulator window.



These fields indicates the current value of GPRs, IXs and FRs you can change any register by modifying the value in the text fields and clicking “change” button.



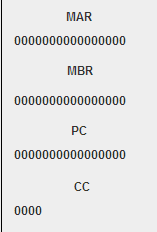
This field is used to enter the instruction that you want to execute right now. by entering the instructions and click “Exe Ins” to execute it. NOTATION: this instruction will replace the current instruction in the IR.



This field is used to input. And the “Initialize” button will initialize the whole program by value every registers, caches and memory to 0.



The “Step” button used to execute one instruction a time, the other button used to execute the whole program until the simulator faces the input instruciton.

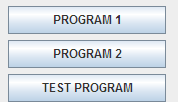


This field indicates the value of MAR, MBR, PC, CC.



The first field used to look at the value of the memory in specific address, and you can alter it and click “ change” to modify the value.

The text field below is used to enter the memory address that you want to look ,and click look to search the value of it.



These three buttons are used to initialize our programs, in this part 4, we only use the “Test program”.

1. Float/vector instructions

The exponent part is represented by the shift code.

|  |  |  |
| --- | --- | --- |
| **OpCode** | **Instruction** | **Description** |
| 033 | FADD fr, x, address[,I] | Floating Add Memory To Register  c(fr)  c(fr) + c(EA)  c(fr)  c(fr) + c(c(EA)), if I bit set  fr must be 0 or 1.  OVERFLOW may be set |
| 034 | FSUB fr, x, address[,I] | Floating Subtract Memory From Register  c(fr)  c(fr) - c(EA)  c(fr)  c(fr) - c(c(EA)), if I bit set  fr must be 0 or 1  UNDERFLOW may be set |
| 035 | VADD fr, x, address[,I] | Vector Add  fr contains the length of the vectors  c(EA) or c(c(EA)), if I bit set, is address of first vector  c(EA+1) or c(c(EA+1)), if I bit set, is address of the second vector  Let V1 be vector at address; Let V2 be vector at address+1  Then, V1[i] = V1[i]+ V2[i], i = 1, c(fr). |
| 053 | VSUB fr, x, address[,I] | Vector Subtract  fr contains the length of the vectors  c(EA) or c(c(EA)), if I bit set is address of first vector  c(EA+1) or c(c(EA+1)), if I bit set is address of the second vector  Let V1 be vector at address; Let V2 be vector at address+1  Then, V1[i] = V1[i] - V2[i], i = 1, c(fr). |
| 054 | CNVRT r, x, address[,I] | Convert to Fixed/FloatingPoint:  If F = 0, convert c(EA) in float point format to a integer number in normal format and store in r.  If F = 1, convert c(EA) to a floating point number and store in FR0.  The r register contains the value of F before the instruction is executed. |
| 50 | LDFR fr, x, address [,i] | Load Floating Register From Memory, fr = 0..1  fr  c(EA), c(EA+1)  fr <- c(c(EA), c(EA)+1), if I bit set |
| 51 | STFR fr, x, address [,i] | Store Floating Register To Memory, fr = 0..1  EA, EA+1  c(fr)  c(EA), c(EA)+1 <- c(fr), if I-bit set |

Note: the red part in the table is different with the project description document.

Our test program:

**Initialization:**

**Register:**

R0=0;R1=1;

FR1=10;

IX1=200;

pc=1000;

**Memory:**

M(10)=300;

M(11)=400;

M(200)=4.1;

M(201)=4.0;

M(202)=7;

M(203)=6.0;

**v1:**

M(300)=15;

M(301)=16;

...

M(310)=25;

**V2:**

M(400)=35;

M(401)=36;

...

M(410)=45;

**Instruciton:**

starts from M(1000);

LDFR FR0,1,0[0]; //FR0=4.1(Float point format)

FADD FR0,1,1[0]; //FR0=8.1(FPF)

STFR FR0,1,4[0]; //M(204)=8.1(FPF)

FSUB FR0,1,3[0]; //FR0=2.1(FPF)

STFR FR0,1,5[0]; //M(205)=2.1(FPF)

VADD FR1,0,10[0]; //vector length = fr1=10, M(300)=50,M(301)=52,...M(310)=70(Normal Format)

VSUB FR1,0,10[0]; //M(300)=15,M(301)=16...M(310)=25(NF)

CNVRT R0,1,0[0]; //R0 = 4 (NF), convert FP 4.1 to a normal integer.

CNVRT R1,1,1[0]; //FR0 = 4.0(FP), convert an integer to a floating point.

**Machine code:**

1010000001000000

0110110001000001

1010010001000100

0111000001000011

1010010001000101

0111010100001010

1010110100001010

1011000001000000

1011000101000010

To test very instruction works.

**Step1:**

After the first instruction we can see the value of fr0 has changed

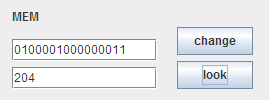


**Next step2**



It change to the 8.1 (floating point format)

**Step3:**



The value of M(204) has been changed to 8.1(FPT)

**Step4:**



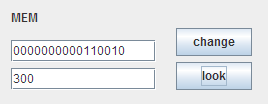
The fr0 has change to 8.1-6.0=2.1(FPT)

**Step5:**



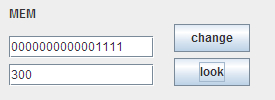
The fr0 stores into M(205)

**Step6:**



The vector length is 10 which stored in fr1, and memory 300-310 which valued 15-25 add 35-45, so their value are changed.

**Step7:**



Their value are changed to the original value because It sub the same vectors they added before.

**Step8:**



R0 change from 0 to 4, because the M(200) values 4.1 the CNVRT converts the float point number to a integer number and stores it into r.

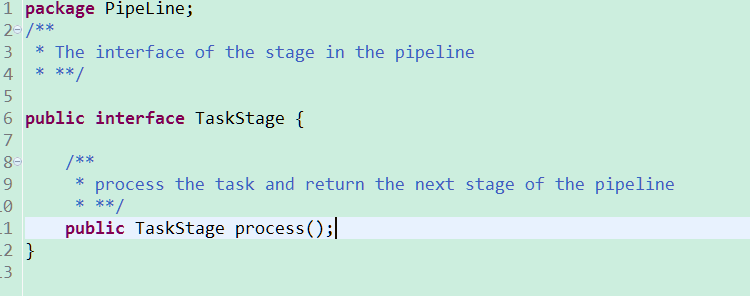
**Step9:**

****

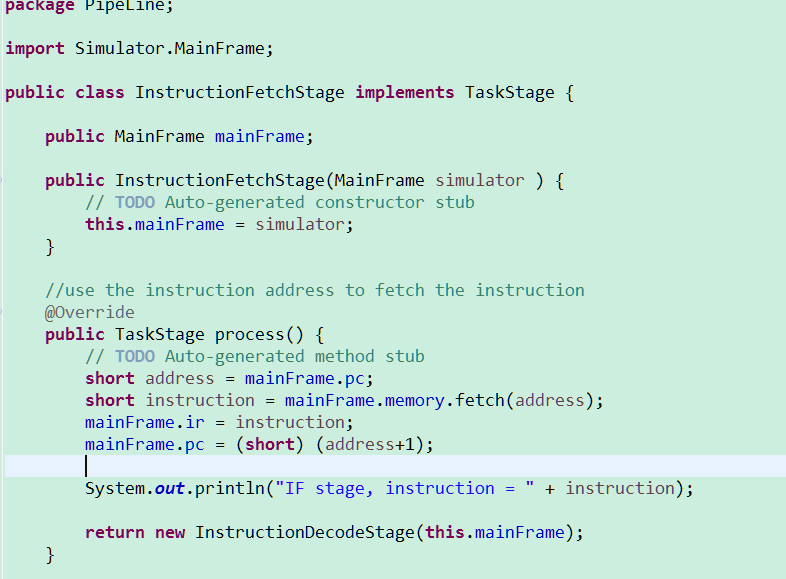
In this step, the CNVRT instruction switch the M(202)=7 in normal format into 7.0 in float point format and store in FR0

1. Simple pipeline

There are 3 stages in the pipeline, instruction fetching, instruction decoding, instruction executing. The simulator maintains three queue to schedule the tasks, they are the *firstStageQueue*,the *secondStageQueue* and the *thirdStageQueue*. The simulator creates three sub-threads and each sub-thread manage a queue. The sub-threads take the tasks in the queues and execute it and then put the returned value into the queue of next stage.

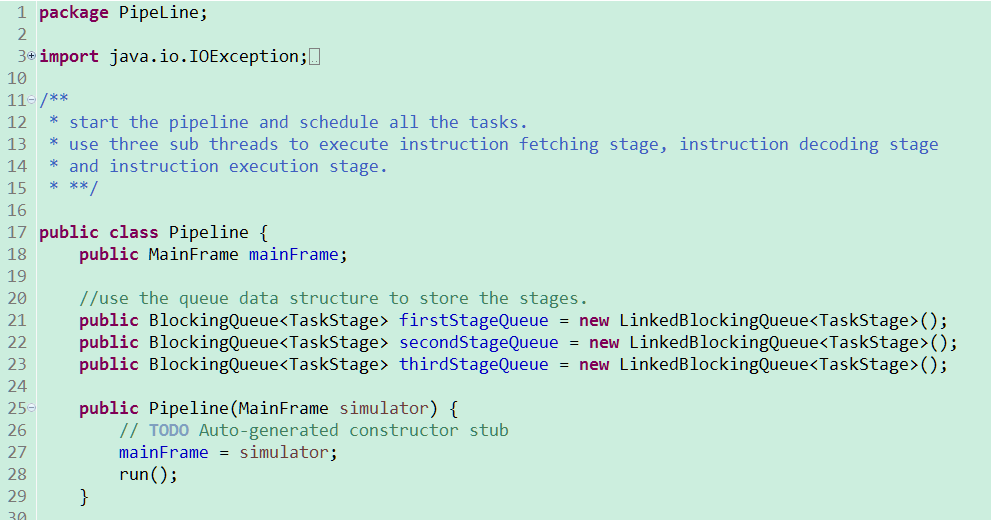


The above picture shows the *TaskStage* interface. The instruction fetching stage, the instruction decoding stage and the instruction executing stage will implement the interface.



This is the class of instruction fetching stage. In this stage, the simulator get the instruction through the instruction address and then let the program counter register add 1. And then the process function create the next stage and return it to the task scheduler. The scheduler will put the returned value into the queue of next stage. The structure of code of three stages are similar. So I will not show the code of rest of the two stages.

The *Pipeline* Class is the scheduler of the pipeline. There are three queues in the class.



The content of the queue are tasks which are not processed. The scheduler creates sub-threads to process the tasks. The following picture is the code of the sub-thread of the first stage.



The sub-thread will take one task from its queue and then process it. If it completes the task, it will get the returned value of the next stage. Then it puts the task of the next stage into the queue of next stage.

The following picture shows the pipeline work normally.

1. 