Document for Project 2

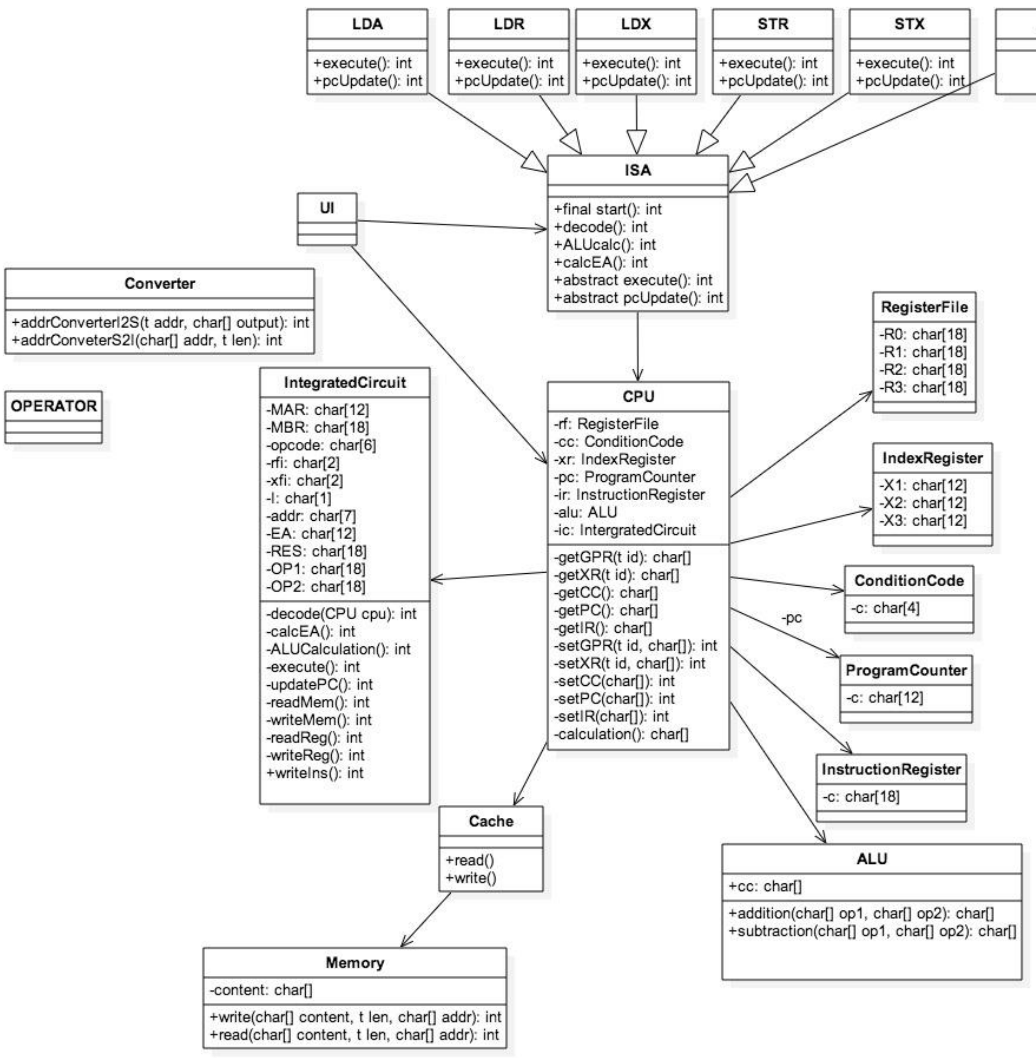
By Group 2

Meng Lin
Siyuan Feng
Rong Wang
Tingting XU

Contents

1 Class specification

The class diagram of the project is showed as following:



1.1 Foreground

Class UI: This is the graphic interface to users. Through UI Users can input instructions, execute instructions, input some data and read/write registers. More details can be found in the user manual.

1.2 Background

Class CPU: This is the main logical interface of the simulator. When the program needs to do something, like reading and writing memory and register, do some arithmetic

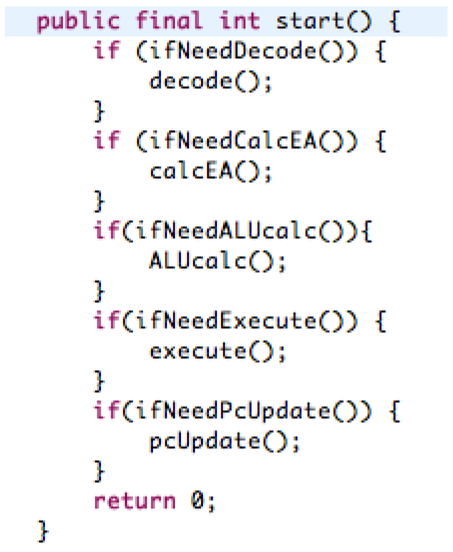
calculations, it should communication with the CPU.
Class Cache: This class realizes the function of a cache. It locates between class Memory and CPU. When it is needed to read or write the memory, the functions of CPU will be invoked. And CPU do these operation through class Cache. The Cache maintains a big block memory which serves as a cache for the Memory. More specification can be found in *2 Cache Specification*.
Class IntegratedCircuit: This class operates just like a physical integrated circuit. There are “wires” inside the IntegratedCircuit. Most of the functions are actually realized here, such as decode, EA calculation, instruction set executions and so on.
Class Memory: This is where all data and instructions are stored. Due to the existence of Cache, the Memory can only be accessed through interfaces in the Cache. There is a large block in the Memory.
Class ALU: This class do all the arithmetic and logical calculations in the simulator, such as addition, subtraction, multiplication and so on.

1.3 Register Classes

RegisterFile: This is the class storing general purpose registers.
IndexRegister: This is the class storing index registers.
ConditionCode: This is the class storing condition code.
ProgramCounter: This is the class storing program counter.
InstructionRegister: This is the class storing instruction register.

1.4 ISA Class

We use template design pattern here. The ISA class is the template. Every instruction class is a subclass of ISA.
The main function of ISA is to provide the instruction template for other classes which actually realize some functions.



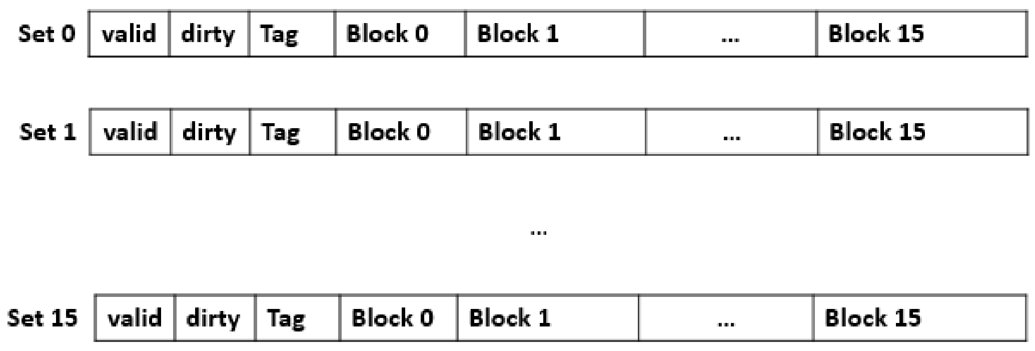
```
public final int start() {
    if (ifNeedDecode()) {
        decode();
    }
    if (ifNeedCalcEA()) {
        calcEA();
    }
    if(ifNeedALUcalc()){
        ALUcalc();
    }
    if(ifNeedExecute()) {
        execute();
    }
    if(ifNeedPcUpdate()) {
        pcUpdate();
    }
    return 0;
}
```

As we can see, ISA provides five stages for an instruction to use. Every instruction can choose to use the stage content provided by ISA, or realizes a stage by itself by overriding.

2 Cache specification

2.1 Cache organization

The cache in our simulator is a Direct-Mapped Cache.



To simplify the realization, we define some terminology ourselves:

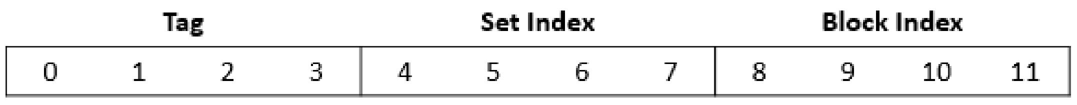
1 sByte = 6 byte
1 sBit = 1 byte
1 word = 3 sByte
1 address = 2 sByte

The parameters of the cache:

Number of sets	16
Number of lines per set	1
Number of blocks per line	16
Number of sBytes per block	1
Cache size	256 sByte

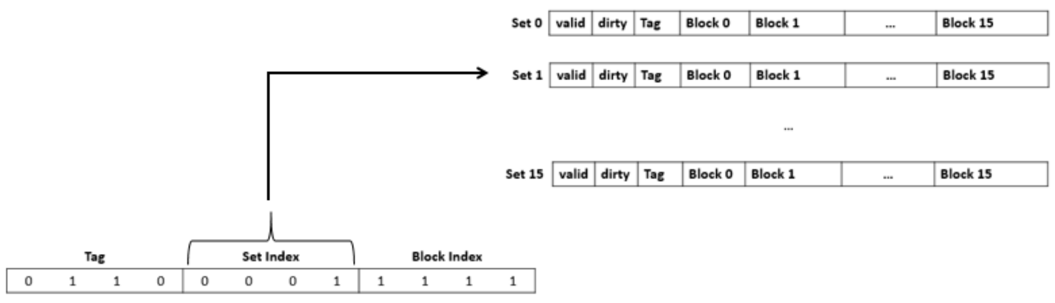
2.2 Address partition

The 12-bit address is organized as following:

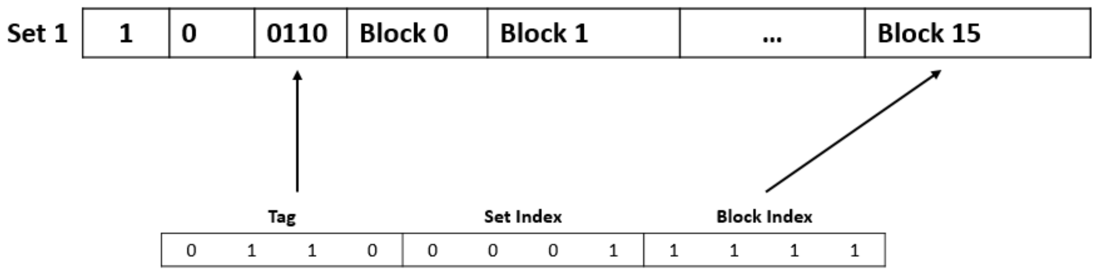


2.3 Cache mapping

2.3.1 Set selection



2.3.2 Line matching and block selection

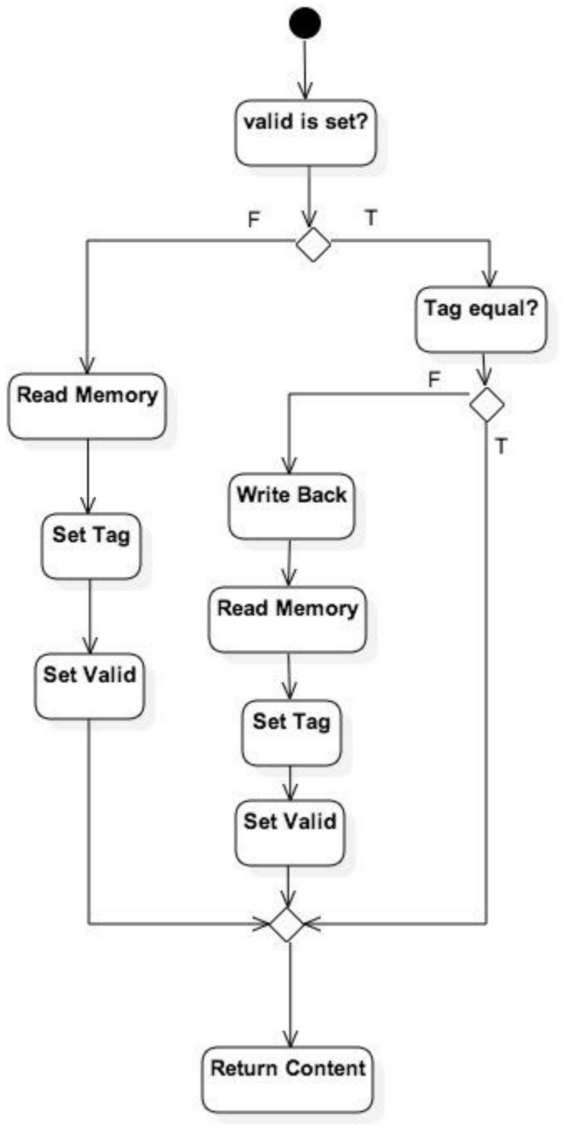


2.3.3 Writing strategy

This writing strategy in our cache is writing back and write allocate.

- Write hit
  When writing data to memory, if the memory location is in the cache, the cache will update data in the block. The changes will only be written back to memory when the line where the changes locates is to be evicted from the cache.
- Write miss
  When a write miss happens, the cache will first load the blocks including the memory location to be written and then update the content.

2.4 Cache Activity Diagram

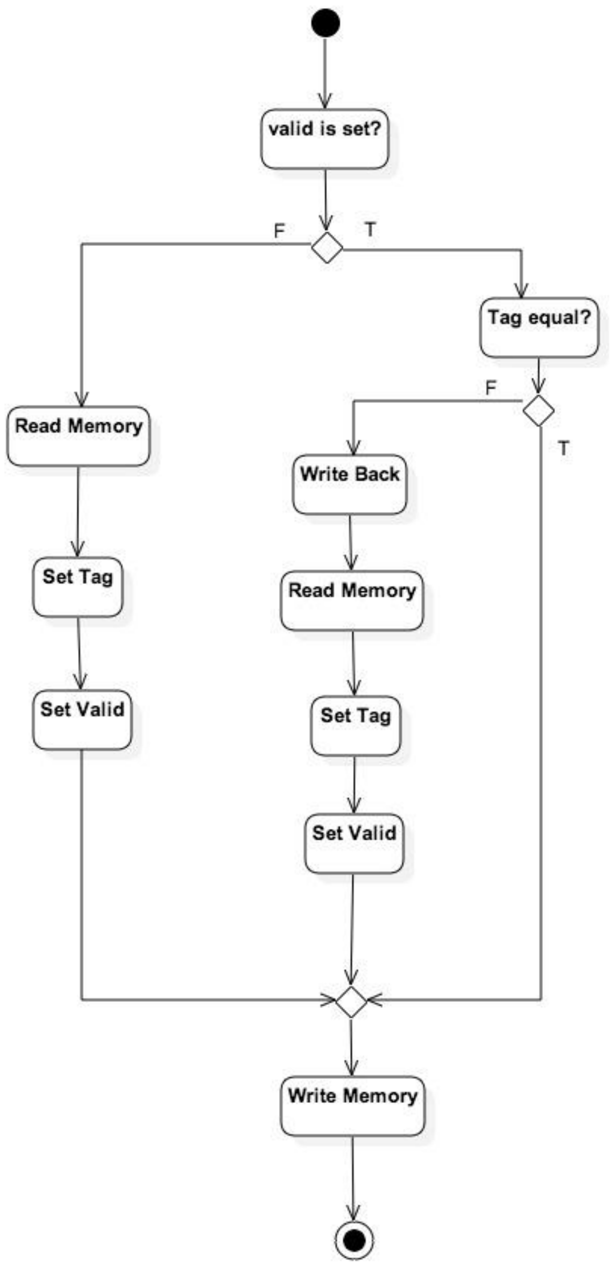


This is the activity diagram for cache read.
First, the valid bit will be check.

- If the valid bit is set, the cache line is valid.
  The tag is check to make sure that the data in the cache is exactly what is requested.
  - If the tag is not matched, the cache line will be written back to the Memory. Cache will then find a collection of blocks which includes the target data and load the blocks into the cache line. Then set the tag and valid bit. At last, Cache can return the request data.
  - If the tag is matched, the cache line contains the target and the Cache can return the request data.

- If the valid bit is not set, the cache line is not valid. New blocks must be loaded.

  Cache will find a collection of blocks which includes the target data and load the blocks into the cache line. Then set the tag and valid bit. At last, Cache can return the request data.



This is the activity diagram for cache write.
First, the valid bit will be check.

- If the valid bit is set, the cache line is valid.
  The tag is check to make sure that the data in the cache is exactly what is requested.
  - If the tag is not matched, the cache line will be written back to the Memory. Cache will then find a collection of blocks which includes the target data and

load the blocks into the cache line. Then set the tag and valid bit. At last, Cache can write data into cache line.

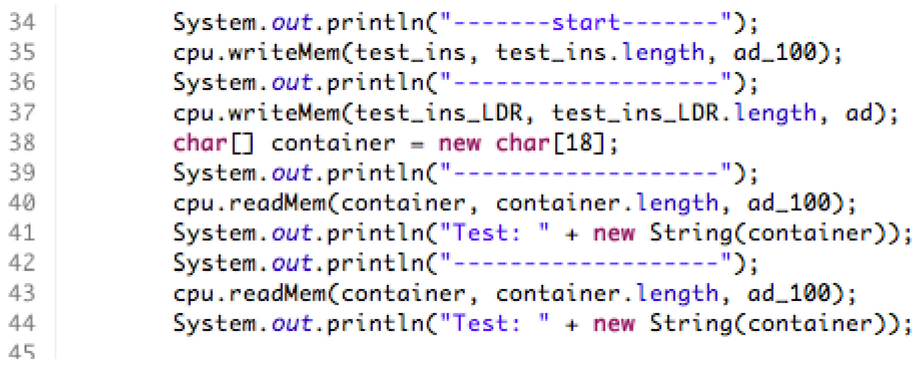
- If the tag is matched, the cache line contains the target and the Cache can write data into cache line.

- If the valid bit is not set, the cache line is not valid. New blocks must be loaded.

    - Cache will find a collection of blocks which includes the target data and load the blocks into the cache line. Then set the tag and valid bit. At last, Cache can write new data into cache line.

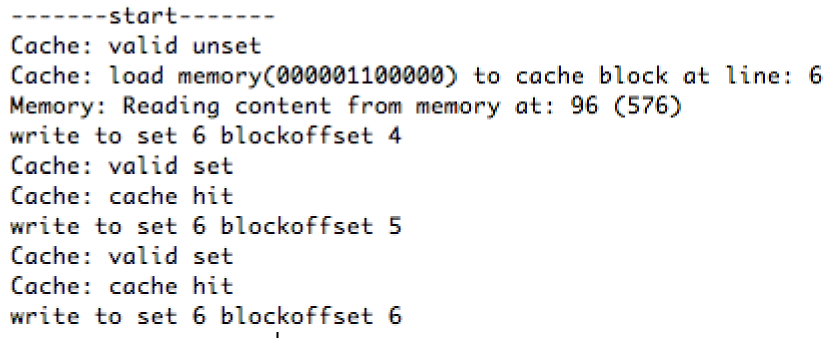
2.5 Example Demonstration

To demonstrate the function of the cache, we design 4 steps. The code is shown as following:



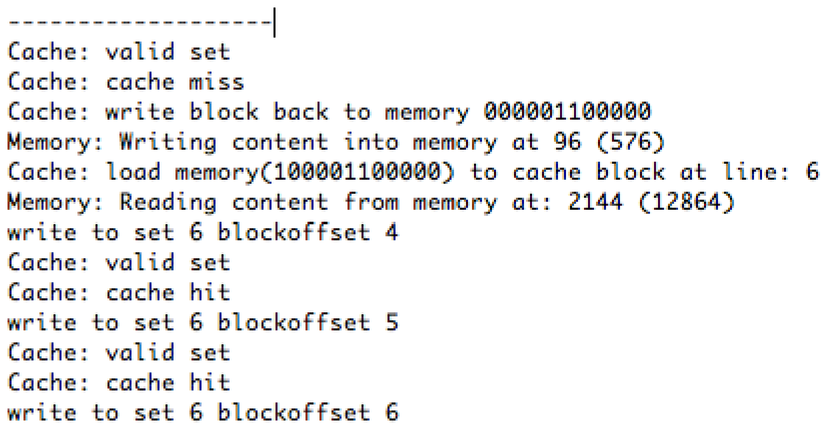
```
        System.out.println("-------start-------");
        cpu.writeMem(test_ins, test_ins.length, ad_100);
        System.out.println("-------------------");
        cpu.writeMem(test_ins_LDR, test_ins_LDR.length, ad);
        char[] container = new char[18];
        System.out.println("-------------------");
        cpu.readMem(container, container.length, ad_100);
        System.out.println("Test: " + new String(container));
        System.out.println("-------------------");
        cpu.readMem(container, container.length, ad_100);
        System.out.println("Test: " + new String(container));
```

Firstly, we write a 3-sByte data at address 0000 0110 0100. Since the cache is empty, thus the valid bit in set 6 is unset and some blocks will be loaded into the cache.



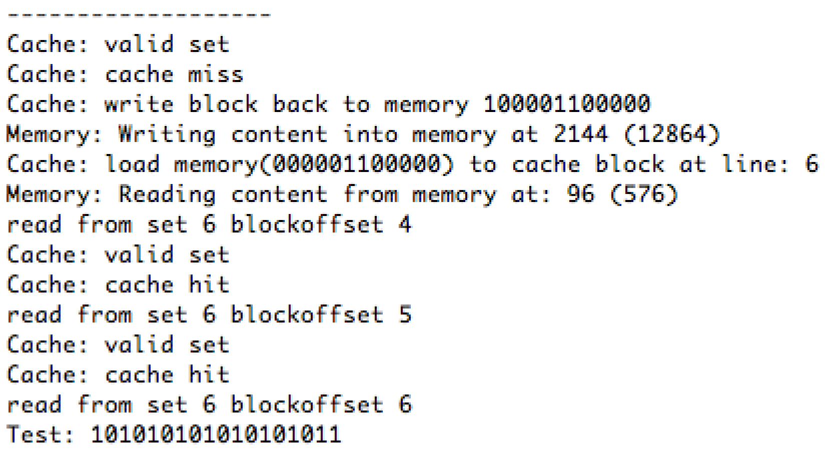
```
-------start-------
Cache: valid unset
Cache: load memory(000001100000) to cache block at line: 6
Memory: Reading content from memory at: 96 (576)
write to set 6 blockoffset 4
Cache: valid set
Cache: cache hit
write to set 6 blockoffset 5
Cache: valid set
Cache: cache hit
write to set 6 blockoffset 6
```

Secondly, we write a 3-sByte data at address 1000 0110 0100. As we can see, the second data are in the same set with the first data's, which is set 6. However, their tag are different. According to the rule of a direct-mapped cache, the line in set 6 will be evicted and new blocks will be loaded.



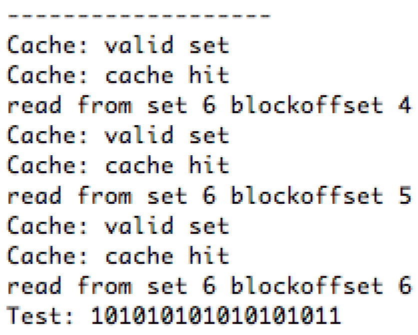
```
-------------------
Cache: valid set
Cache: cache miss
Cache: write block back to memory 000001100000
Memory: Writing content into memory at 96 (576)
Cache: load memory(100001100000) to cache block at line: 6
Memory: Reading content from memory at: 2144 (12864)
write to set 6 blockoffset 4
Cache: valid set
Cache: cache hit
write to set 6 blockoffset 5
Cache: valid set
Cache: cache hit
write to set 6 blockoffset 6
```

Thirdly, we read a 3-sByte data at address 0000 0110 0100. The data has been evicted from cache, so there will be a cache miss. A new replacement of cache lines will happen.



```
-------------------
Cache: valid set
Cache: cache miss
Cache: write block back to memory 100001100000
Memory: Writing content into memory at 2144 (12864)
Cache: load memory(000001100000) to cache block at line: 6
Memory: Reading content from memory at: 96 (576)
read from set 6 blockoffset 4
Cache: valid set
Cache: cache hit
read from set 6 blockoffset 5
Cache: valid set
Cache: cache hit
read from set 6 blockoffset 6
Test: 101010101010101011
```

Finally, we repeat step 3, and read a 3-sByte data at address 0000 0110 0100 again. There should be a cache hit.



```
-------------------
Cache: valid set
Cache: cache hit
read from set 6 blockoffset 4
Cache: valid set
Cache: cache hit
read from set 6 blockoffset 5
Cache: valid set
Cache: cache hit
read from set 6 blockoffset 6
Test: 101010101010101011
```

From the demonstration above, we can see that our cache works correctly.

3 Program one

Program 1: A program that reads 20 numbers (integers) from the keyboard, prints the numbers to the console printer, requests a number from the user, and searches the 20 numbers read in for the number closest to the number entered by the user. Print the number entered by the user and the number closest to that number. Your numbers should not be 1…10, but distributed over the range of 1 … 65,536. Therefore, as you read a character in, you need to check it is a digit, convert it to a number, and assemble the integer.

3.1 Assembly language design

The whole process is separated into 4 stages.
To read 20 numbers from the keyboard, we need to do a 20-time loop. We use SOB to realize the loop. We first set the R3 to 20, then do the IN operation, and use SOB based on the value in R3. If the value in R3 is greater than 0 after decreased by 1, the program will go back to instruction IN and continue.

```
AIR 3 20
AIX 3 100
IN 1 0
STR 1 3 0 100
AIX 3 3
SOB 3 0 0 9
```

After inputting 20 numbers, we output these numbers by using SOB.

```
AIR 3 20
AIX 3 100
IN 1 0
STR 1 3 0 100
AIX 3 3
SOB 3 0 0 9
AIX 2 100
AIR 3 20
AIX 1 100
LDR 1 1 0 100
OUT 1 1
SIX 1 3
SOB 3 0 0 30
```

Then we get an input from user by calling IN once, and store the number at location 120 into the memory.

```
IN 1 0
STR 1 0 0 120
```

To find the number that is closest to the input from user among the 20 number, we need to do a 20-time loop, calculate the difference between the user input and each

of the 20 number. When one difference is calculated, we need to compare it with the former one. If it is smaller, we change the candidate for the closet number, and continue the compare until all of the 20 numbers are compared.
We store the closest number at location X2+94, and the difference at location X2+97.
JCC is used here to compare two difference. If the one calculated is smaller than the smallest difference up to now, JCC jump to codes that change the smallest number and smallest difference we have observed so far, and then come back.
SOB is used here to realize the loop, just as described in step 1.

```
STR 1 0 0 120
LDR 1 3 0 100
STR 1 2 0 94
SMR 1 0 0 120
STR 1 2 0 97
SIR 3 1
SIX 3 3
LDR 1 3 0 100
SMR 1 0 0 120
SMR 1 2 0 97
JCC 1 0 0 90
SOB 3 0 0 69
JMP 0 0 105
LDR 1 3 0 100
STR 1 2 0 94
SMR 1 0 0 120
STR 1 2 0 97
JMP 0 0 84
```

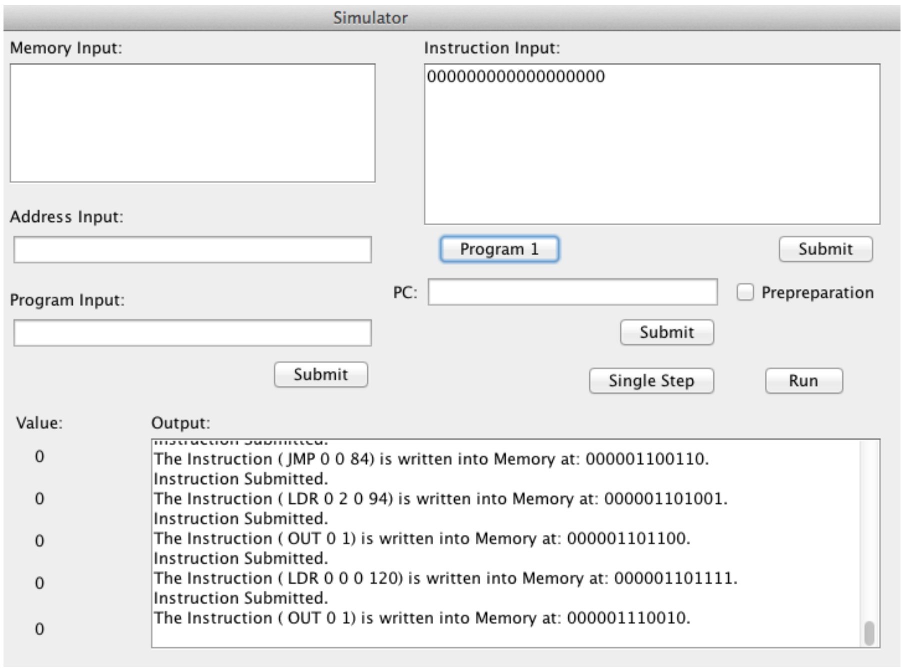
Finally, we output the closest number which is at X2+94 and the number user input which is at 120.

```
LDR 0 2 0 94
OUT 0 1
LDR 0 0 0 120
OUT 0 1
```

3.2 Output

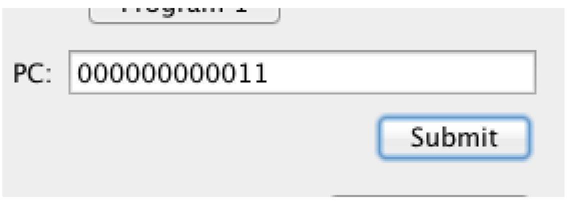
3.2.1 Process

To input assembly codes quickly into memory, we design a button on the UI. You can quickly input the codes into memory (or you can input the 38 codes manually).



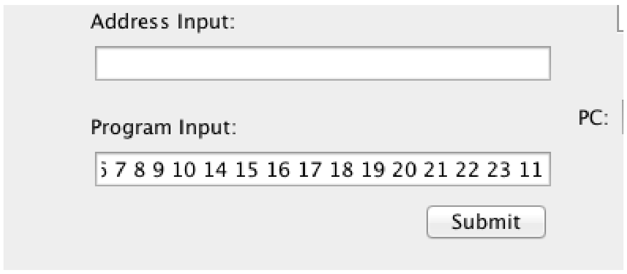
```
The Instruction ( JMP 0 0 84) is written into Memory at: 000001100110.
Instruction Submitted.
The Instruction ( LDR 0 2 0 94) is written into Memory at: 000001101001.
Instruction Submitted.
The Instruction ( OUT 0 1) is written into Memory at: 000001101100.
Instruction Submitted.
The Instruction ( LDR 0 0 0 120) is written into Memory at: 000001101111.
Instruction Submitted.
The Instruction ( OUT 0 1) is written into Memory at: 000001110010.
```

Then set the PC to the right place.



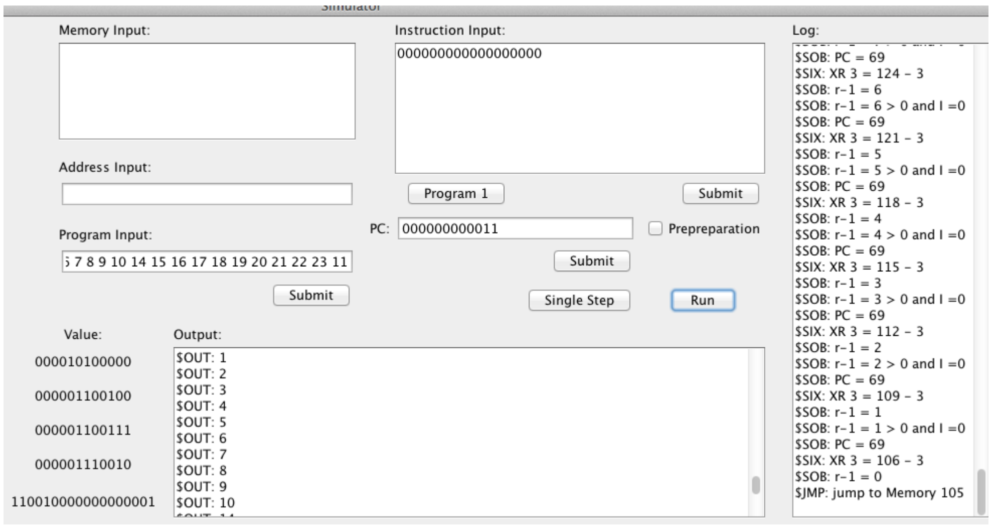
PC: 000000000011

To run the program, you can choose to run step by step, or run at once.
There are two ways to input numbers.
The first is write 21 numbers in the Program Input box, and press Submit.
The second is to check the Prepreparation Box. Then, 21 random numbers will be produced and send them to the program. This is the default way. We use the first way.
The input is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23. The requested number is 11. Thus the closest number should be 10.



Program Input: 5 7 8 9 10 14 15 16 17 18 19 20 21 22 23 11

After press “Run” button, the result will be shown in the Output Panel, and some information will be shown in the Log Panel.



Instruction Input: 000000000000000000

Program Input: 5 7 8 9 10 14 15 16 17 18 19 20 21 22 23 11

PC: 000000000011

Value:

```
000010100000
000001100100
000001100111
000001110010
110010000000000001
```

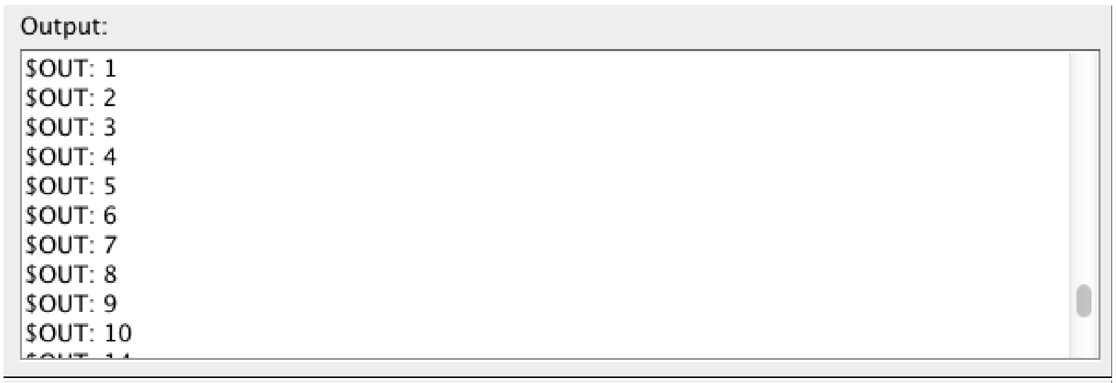
Output:

```
$OUT: 1
$OUT: 2
$OUT: 3
$OUT: 4
$OUT: 5
$OUT: 6
$OUT: 7
$OUT: 8
$OUT: 9
$OUT: 10
```

Log:

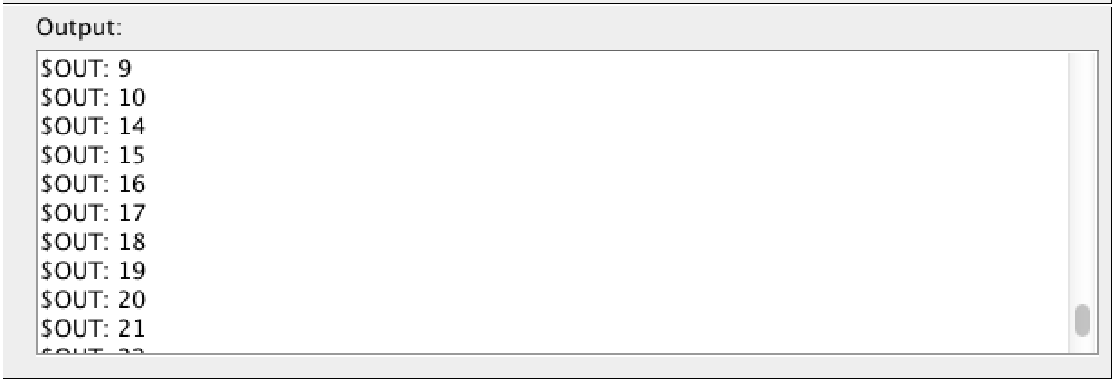
```
$SOB: PC = 69
$SIX: XR 3 = 124 – 3
$SOB: r–1 = 6
$SOB: r–1 = 6 > 0 and I =0
$SOB: PC = 69
$SIX: XR 3 = 121 – 3
$SOB: r–1 = 5
$SOB: r–1 = 5 > 0 and I =0
$SOB: PC = 69
$SIX: XR 3 = 118 – 3
$SOB: r–1 = 4
$SOB: r–1 = 4 > 0 and I =0
$SOB: PC = 69
$SIX: XR 3 = 115 – 3
$SOB: r–1 = 3
$SOB: r–1 = 3 > 0 and I =0
$SOB: PC = 69
$SIX: XR 3 = 112 – 3
$SOB: r–1 = 2
$SOB: r–1 = 2 > 0 and I =0
$SOB: PC = 69
$SIX: XR 3 = 109 – 3
$SOB: r–1 = 1
$SOB: r–1 = 1 > 0 and I =0
$SOB: PC = 69
$SIX: XR 3 = 106 – 3
$SOB: r–1 = 0
$JMP: jump to Memory 105
```

3.2.2 Output result



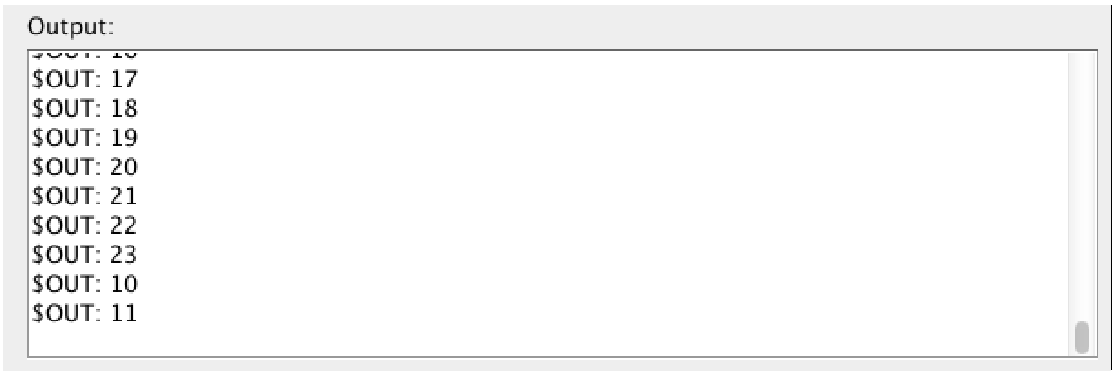
Output:

```
$OUT: 1
$OUT: 2
$OUT: 3
$OUT: 4
$OUT: 5
$OUT: 6
$OUT: 7
$OUT: 8
$OUT: 9
$OUT: 10
```



Output:

```
$OUT: 9
$OUT: 10
$OUT: 14
$OUT: 15
$OUT: 16
$OUT: 17
$OUT: 18
$OUT: 19
$OUT: 20
$OUT: 21
```

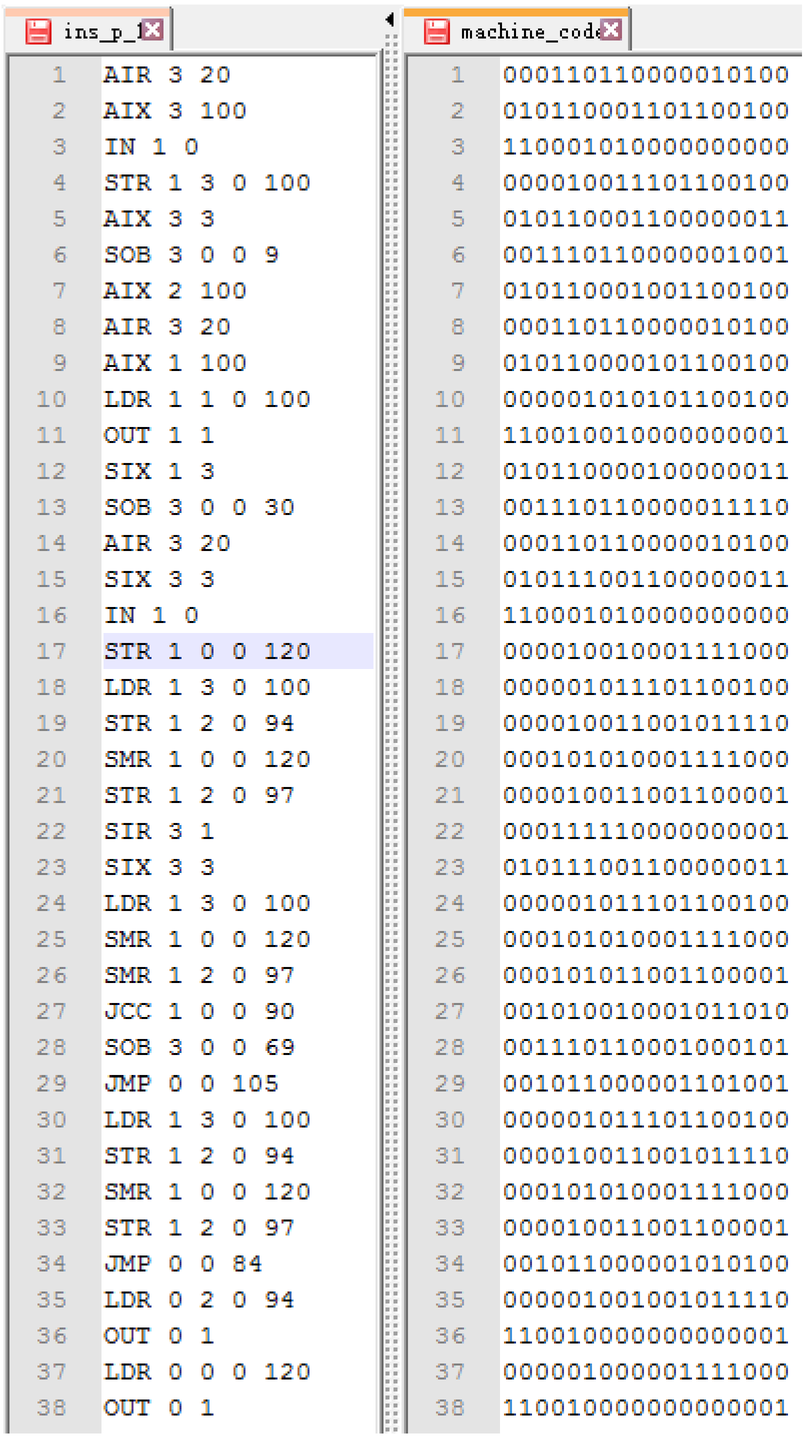

Output:

```
$OUT: 17
$OUT: 18
$OUT: 19
$OUT: 20
$OUT: 21
$OUT: 22
$OUT: 23
$OUT: 10
$OUT: 11
```

As we can see, 20 numbers are:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23.
The one that is closest to the requested number is 10.
The requested number is 11.

3.2.3 Machine codes

The machine codes produced by our simulator is

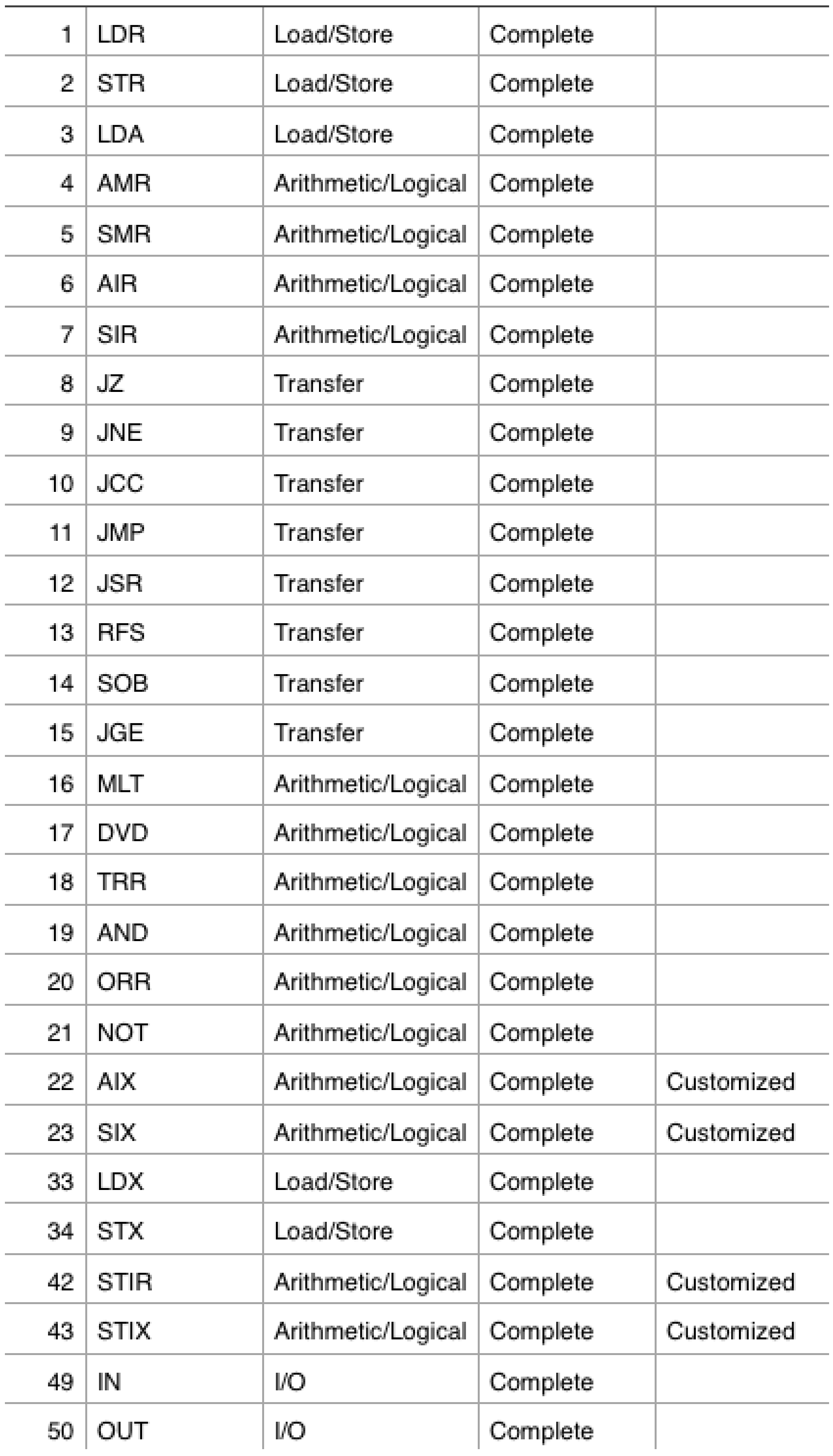


```
AIR 3 20
AIX 3 100
IN 1 0
STR 1 3 0 100
AIX 3 3
SOB 3 0 0 9
AIX 2 100
AIR 3 20
AIX 1 100
LDR 1 1 0 100
OUT 1 1
SIX 1 3
SOB 3 0 0 30
AIR 3 20
SIX 3 3
IN 1 0
STR 1 0 0 120
LDR 1 3 0 100
STR 1 2 0 94
SMR 1 0 0 120
STR 1 2 0 97
SIR 3 1
SIX 3 3
LDR 1 3 0 100
SMR 1 0 0 120
SMR 1 2 0 97
JCC 1 0 0 90
SOB 3 0 0 69
JMP 0 0 105
LDR 1 3 0 100
STR 1 2 0 94
SMR 1 0 0 120
STR 1 2 0 97
JMP 0 0 84
LDR 0 2 0 94
OUT 0 1
LDR 0 0 0 120
OUT 0 1
```

```
0001101100000010100
0101100011011001 00
```

4 Instruction Schedule

These are the instruction we have realized so far. Among them, there are 4 instructions added by ourselves. They are SIX (Subtract Immediate from Index register), AIX (Add Immediate from Index register), STIR (Store Immediate to register) and STIX (Store Immediate to index register). Most of them are used in the program one, as you can see them in the assembly codes.



1	LDR	Load/Store	Complete	
2	STR	Load/Store	Complete	
3	LDA	Load/Store	Complete	
4	AMR	Arithmetic/Logical	Complete	
5	SMR	Arithmetic/Logical	Complete	
6	AIR	Arithmetic/Logical	Complete	
7	SIR	Arithmetic/Logical	Complete	
8	JZ	Transfer	Complete	
9	JNE	Transfer	Complete	
10	JCC	Transfer	Complete	
11	JMP	Transfer	Complete	
12	JSR	Transfer	Complete	
13	RFS	Transfer	Complete	
14	SOB	Transfer	Complete	
15	JGE	Transfer	Complete	
16	MLT	Arithmetic/Logical	Complete	
17	DVD	Arithmetic/Logical	Complete	
18	TRR	Arithmetic/Logical	Complete	
19	AND	Arithmetic/Logical	Complete	
20	ORR	Arithmetic/Logical	Complete	
21	NOT	Arithmetic/Logical	Complete	
22	AIX	Arithmetic/Logical	Complete	Customized
23	SIX	Arithmetic/Logical	Complete	Customized
33	LDX	Load/Store	Complete	
34	STX	Load/Store	Complete	
42	STIR	Arithmetic/Logical	Complete	Customized
43	STIX	Arithmetic/Logical	Complete	Customized
49	IN	I/O	Complete	
50	OUT	I/O	Complete	