Milestone 1 Documentation

Team 1A - Khaled Alfayez, Shaun Davis,
Trinity Merrell, and Logan Smith

CONTENTS:

Register Descriptions

Summary of Registers

Register	Number	Availability	Description
$s*n* *(0-3)*	0-3	Read/write	General purpose registers. The intent is to store long term computation results and save values over functions calls
$t*n* *(0-3)*	4-7	Read/write	General purpose registers, to be used like $s*n* registers. Will NOT be saved over function calls.
$cr	8	Read only	This register holds the result of the most recent computation instruction (arithmetic or logical)
$ra	9	Read/write	Stores the return address of a function
$a*n* (0-1)	10-11	Read/write	These registers are used to store arguments for use in a called function
$v	12	Read/write	This register is for storing the return value from a function
$d	13	Read/write	This register is used for communication between the display and the processor
$st	14	Read/write	Reference to the top bit of the stack
$i*n* (0-1)	15-16	Not available	Used by assembler for pseudoinstructions.
$ex	17	Read only	Cause register for interrupt and exception handling; NOT accessible by regular users
$k*n* (0-1)	18-19	Read/write (while handling exceptions)	OS registers; NOT accessible by regular users
in/output	20	Read (input) Write (output)	The user can only interact with these registers indirectly using some instructions.

Unsafe between procedure calls **Safe between procedure calls**

Assembly

Procedure Call Conventions:

1. Registers $i*n*, $k*n* (where n is 0-1) are reserved for the assembler and operating system and should not be used by user programs or compilers.
2. Registers $a*n* (where n is 0-1) are used to pass arguments to procedures, any other arguments should go on the stack. Register $v is used to return a value from functions.
3. Registers $a*n*, $t*n*, $d, $cr, and $v are temporary and volatile. Expect them to contain different data after a procedure call.
4. $s*n* registers must be backed up on the stack at the beginning of a procedure and restored before returning from the procedure. This preserves values in these registers over procedure calls.
5. $st is the stack register. It points to the top memory location in the stack. If the stack is grown at any time in a procedure, it must be reduced before returning from that procedure.
    a. Memory is allocated to the stack by subtracting from the value in $st. Memory is deallocated from the stack by adding to the value in $st.
6. $ra is the return address of a procedure. Jal will overwrite $ra to be the next instruction, so $ra must ALWAYS be backed up on the stack before a procedure call and restored after returning from the procedure.
7. The instruction jr will return the program to the value in $ra.

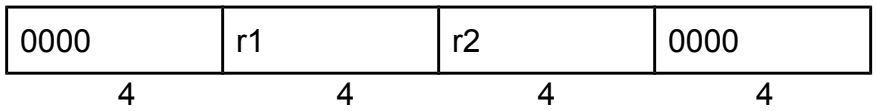
Syntax and Semantics:

Arithmetic and Logical Instructions

Arithmetic and Logical Instructions are C-type instructions (see Machine Language for more information). These instructions take two registers as operands to their computations. Their results are always stored in the specialized $cr register.

Addition:

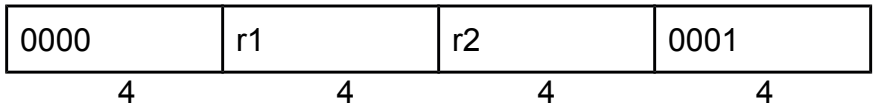
add r1, r2



Stores the sum of r1 and r2 into register $cr

Subtraction:

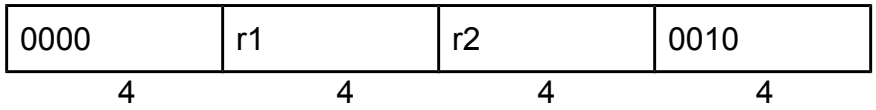
sub r1, r2



Stores the difference between r1 and r2 into register $cr

AND:

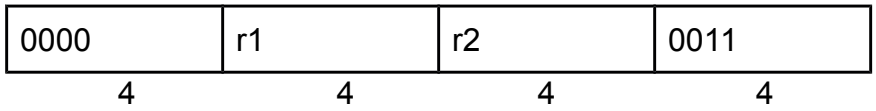
and r1, r2



Stores the logical AND of r1 and r2 into register $cr

OR:

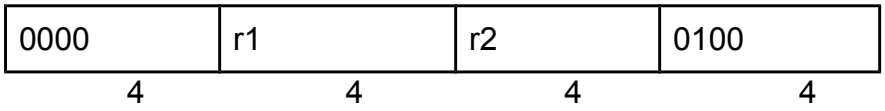
or r1, r2



Stores the logical OR of r1 and r2 into register $cr

NOR:

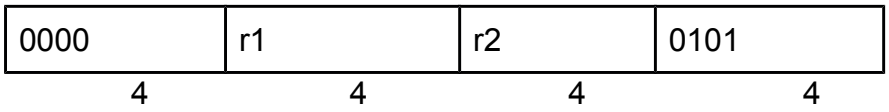
nor r1 ,r2



Stores the logical NOR of r1 and r2 into register $cr

NAND:

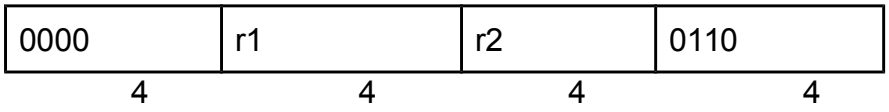
nand r1, r2



Stores the logical NAND of r1 and r2 into register $cr

Exclusive OR:

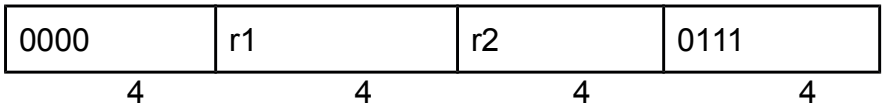
xor r1, r2



Stores the logical Exclusive OR of r1 and r2 into register $cr

Set Less Than:

slt r1, r2



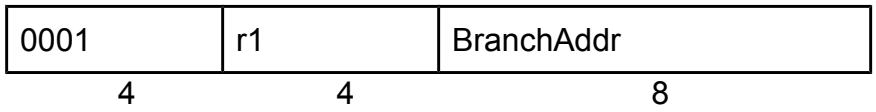
Stores either a 1(True) or a 0(False) in $cr depending on if r1 is less than r2

Branch Instructions

Branch instructions require a comparative value to be computed before execution. The branch instruction will then succeed or fail based on equivalence or inequality.

Branch if equal:

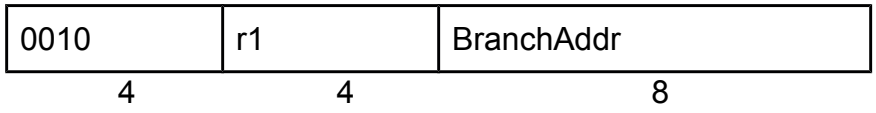
bieq r1, location



Conditional branch to address in immediate if r1 is equal to register $cr

Branch not equal:

bneq r1, location

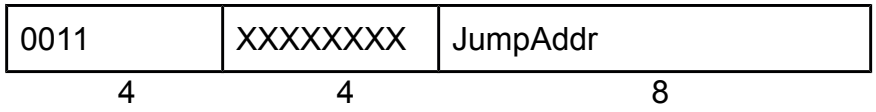


Conditional branch to address in immediate if r1 does not equal register $cr

Jump Instructions

Jump:

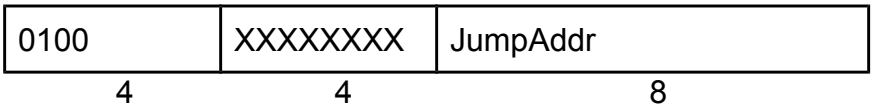
j location



Unconditional jump to the address in immediate

Jump and link:

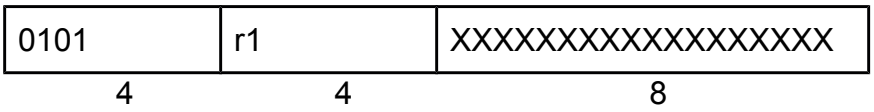
jal location



Unconditional jump to the address in immediate, storing the address of subsequent instruction into register $ra

Jump register:

jr r1



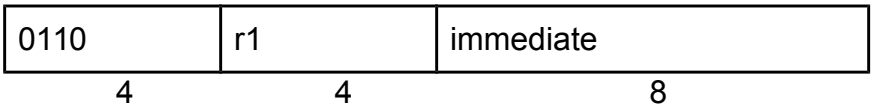
Unconditional jump to the address in r1

Load/Store Instructions

Load/Store instructions often require a value to be computed and stored in $cr prior to execution. Specific requirements are denoted for each instruction.

Load to register:

ltr r1, small



Take an immediate value and store in r1. If immediate is greater than 8-bits, becomes a pseudo instruction utilizing load upper immediate and load lower immediate.

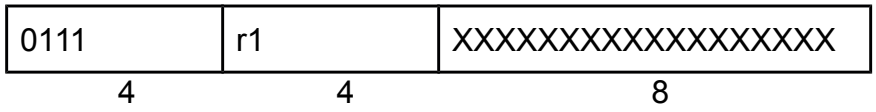
ltr r1, big

Translates to:

lui $i0, upper(big)
lli $i1, lower(big)
or $i0, $i1
ctr r1

Copy to register:

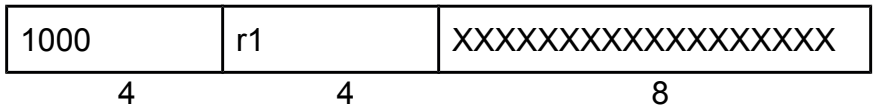
ctr r1



Take a previously computed value from $cr and store in r1

Load word:

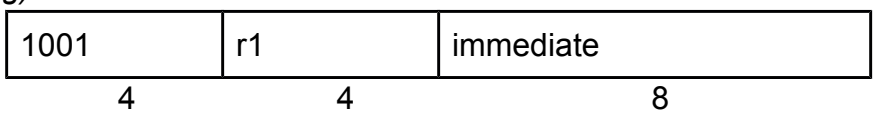
lw r1



Take a value from a previously computed memory address from $cr and store in r1

Load upper immediate:

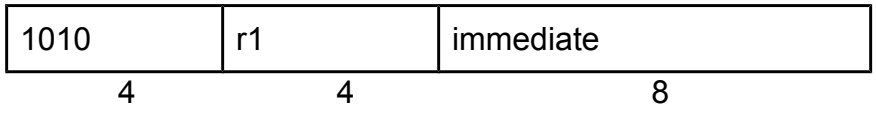
lui r1, upper(big)



Load top half of 16-bit immediate into r1

Load lower immediate:

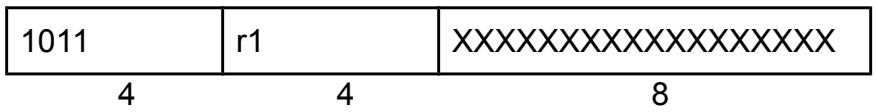
lli r1, lower(big)



Load lower half of 16-bit immediate into r1

Store word:

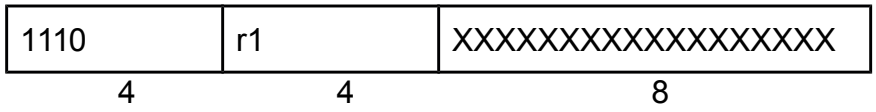
sw r1



Store value in r1 at the previously computed memory address in register $cr.

Read from input:

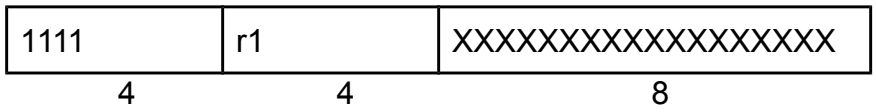
rfi r1



Reads whatever is in the input register and copies it to r1.

Write to Output:

rto r1



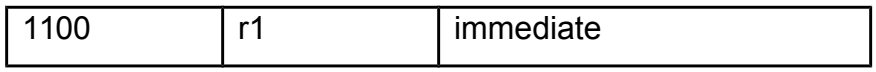
Writes whatever is in the given register to the output register.

Pseudo Instructions

These are instructions that do not actually exist, but that our processor will handle with smaller instructions. All pseudo instructions are I-type. Translations are provided.

OR immediate:

ori r1, big



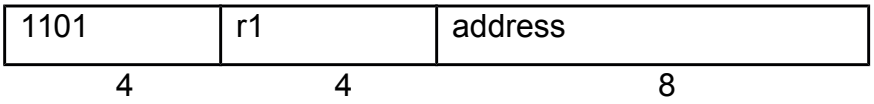
Stores the logical OR of r1 and immediate into register $cr

Translation:

lui $i0, upper(big)
lli $i1, lower(big)
or $i0, $i1
or r1, $cr

Load address:

la r1, address



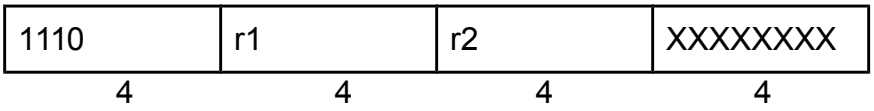
Stores the address in immediate into r1

Translation:

lui $i0, upper(address)
lli $i1, lower(address)
or $i0, $i1
ctr r1

Register to Register:

rtr r1, r2



Moves the value in r2 to r1.

Translation:

ltr $i0, 0
add $r2, $i0
ctr r1

Example 1 - relPrime and Euclid's algorithm:

```
relPrimeSetup:
        ltr $t0 -8
        add $st $t0
        sw  $ra
        add $st $t0
        sw  $s0
        ltr $t1 2       # m = 2
        rtr $a1 $t1     # m in $a1
        rtr $s0 $a1     # m in $s0
        rfi $a0

relPrimeLoop:
        ltr $t0 1
        ltr $t1 0
        add $t1 $t0
        jal gcd
```

```
        bieq $v cleanup # if gcd(n, m) == 1 jump to cleanup

        # body of while loop
        ltr $t0 1
        add $s0 $t0   # m + 1
        ctr $s0                 # m = m + 1
        rtr $a1 $s0
        j relPrimeLoop

gcd:
        ltr $t0 0
        add $t0 $t0
        bneq $a0 subOne    # if a != 0, go to subOne
        rtr $v $a1
        jr $ra   # return to loop

subOne:
        slt $a1 $a0
        ltr $t0 1
        bieq $t0 subTwo      # if (a > b) go to subTwo
        sub $a1 $a0  # b - a
        ctr $a1          # b = b - a
        rtr $v $a0       # return a
        jr $ra

subTwo:
        sub $a0 $a1  # a - b
        ctr $a0          # a = a- b
        rtr $v  $a0      # return a
        jr $ra

cleanup:
        rtr $v $a1       # return m
        ltr $t0 0
        add $st $t0
        lw $s0
        ltr $t0 8
        add $st $t0
        lw $ra
        add $st $t0
        jr $ra
```

Example 2 - Common operations:

Table of Common Operations

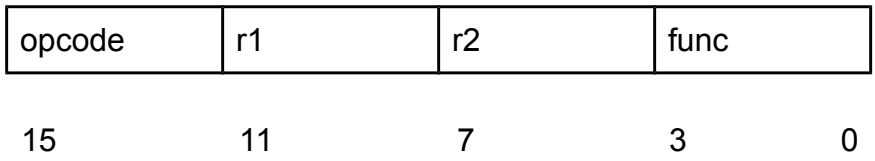
	SAPA 1.0	Description
Load Address	*la $s2, 0x4EF6*	Loading an address is a pseudoinstruction. Refer to *Pseudoinstruction* for full details on how *la* works.
Iterations	*…* *ltr $s1, 15* *ltr $t0, 1* *ltr $t1, 0* *ltr $t2, 3* *loop: slt $s1, $t1* *bieq $t0, exit* *add $t1, $t2* *ctr $t1* *j loop* *exit: ...*	This *for-loop* keeps adding 3 to register $s3 (0) until $t1 becomes greater than $s1 (15), resulting with a 18 stored in $t1 after the loop has exited.
Branches	*See above*	The example above utilizes *slt* and *bieq* to create a branch on greater than, which isn't an instruction itself but can be created using multiple instructions.
Reading from inputs	*rfi $s1*	In this example, this function reads whatever is in the input register and copies it to the given register.
Writing to outputs	*func: …* *wto $s1* *...*	This function writes whatever is in the given register to the output register.
Reading from display reg	*ltr $t0, 0* *add $d, $t0* *ctr $s0*	Here, we read the data in the display reg $d into $cr and copy the data to a general purpose register ($s0).
Writing to display reg	*ltr $d, 415* *out*	In this example, we write a value into $d and then output it into a 16-bit LCD screen.

Machine Language

Instruction Types:

Basic Instruction Formats:

C-type, Computation types (register to register)



C-type instructions are used for register to register computations. They handle arithmetic and logical computations such as add, sub, and, or, etc. These instructions share a single opcode, and are distinguishable by their func code.

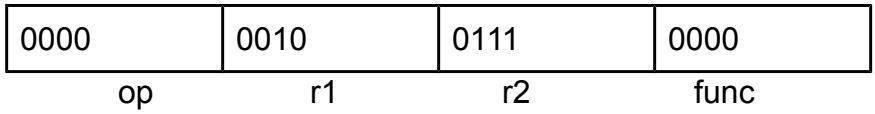
I-type, Immediate types (register to data, register to memory)

I-type instructions are used for register to data and register to memory computations. These instructions handle storing data from registers into memory and loading data into registers from memory, immediate values, and the $cr register. They also handle control flow such as branches and jumps that are necessary for loops and procedure calls.

Rules for Translating Assembly to Machine Language:

Arithmetic and logical instructions are directly translated from their assembly to the machine language. For example, the following assembly would translate accordingly into binary:

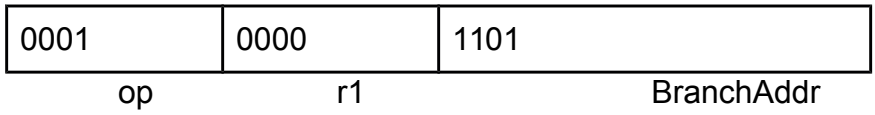
add $s2, $t3



Registers are directly translated from their respective numbers in the registry (see chart in Registers).

Branch instructions are PC-relative, meaning they use an 8-bit offset that allows a user to jump to 2^7-1 instructions forward or 2^7 backward. A translation may appear as below (where “loop” is 3 instructions above bieq):

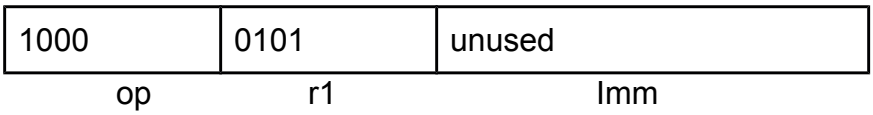
bieq $s0, loop



Jump instructions use an 8-bit immediate with the top 8 bits of the PC concatenated to create a 16-bit address. Jump instructions use this address for the new PC address to jump to, allowing us to jump 2^7-1 instructions forward or 2^7 backward.

Load/store instructions are directly translated from like arithmetic and logical instructions. The following assembly would translate accordingly:

lw $t0



For the assembly to machine language translation of all instructions, refer to the chart below.

Key:
8 = $cr, 9 = $ra

Instruction	Type	Verilog	Description of bits and rules			
Add	C	R[8] = R[r1] + R[r2]	0000	r1	r2	0000
Sub	C	R[8] = R[r1] - R[r2]	0000	r1	r2	0001
AND	C	R[8] = R[r1] & R[r2]	0000	r1	r2	0010
OR	C	R[8] = R[r1] || R[r2]	0000	r1	r2	0011
NOR	C	R[8] = ~(R[r1] || R[r2])	0000	r1	r2	0100
Set Less Than	C	R[8] = (R[r1] < R[r2]) ? 1 : 0	0000	r1	r2	0101
Branch Equal	I	If (R[8] = R[r1]) PC = PC + 4 + BranchAddr	0001	r1	BranchAddr	
Branch Not Equal	I	If (R[8] != R[r1]) PC = PC + 4 + BranchAddr	0010	r1	BranchAddr	
Jump	I	PC = JumpAddr	0011	XXX	JumpAddr	
Jump and Link	I	R[9] = PC + 8 PC = JumpAddr	0100	r1	JumpAddr	
Jump Register	I	PC = R[r1]	0101	r1	XXX	

Load to Register	I	R[r1] = SignExtImm	0110	r1	Immediate	
Copy to Register	I	R[r1] = R[8]	0111	r1	XXX	
Load Word	I	R[r1] = M[R[8]]	1000	r1	XXX	
Load Upper Immediate	I	R[r1] = {imm, 8'b0}	1001	r1	Immediate	
Load Lower Immediate	I	R[r1] = {8'b0, imm}	1010	r1	Immediate	
Store Word	I	M[R[8]] = R[r1]	1011	r1	XXX	
ORi	I	R[8] = r1 | ZeroExtImm	1100	r1	Immediate	
Load Address	I	R[r1] = SignExtImm	1101	r1	Immediate	
Register to Register	C	R[r1] = R[r2]	1110	r1	r2	XXX

Machine Language Translations:

relPrimeSetup:

ltr $t0 -4

0110	0100	11111100

add $st $t0

0000	1110	0100	0000

sw $ra

1011	1001	xxxxxxxx

add $st $t0

0000	1110	0100	0000

sw $s0

1011	0000	xxxxxxxx

ltr $t1 2 # m = 2

0110	0101	00000010

rtr $a1 $t1 # m in $a1

0110	1111	00000000

0000	0101	1111	0000

0111	1011	xxxxxxxx

rtr $s0 $a1 # m in $s0

0110	1111	00000000

0000	1011	1111	0000

0111	0000	xxxxxxxx

rfi $a0

1110	1010	xxxxxxxx

relPrimeLoop:

ltr $t0 1

0110	0100	00000001

ltr $t1 0

0110	0101	00000000

add $t1 $t0

0000	0101	0100	0000

jal gcd

0100	xxxx	01010100

bieq $v cleanup # if gcd(n, m) == 1 jump to cleanup

0001	1100	1010000

body of while loop

ltr $t0 1

0110	0100	00000001

add $s0 $t0 # m + 1

0000	0000	0100	0000

ctr $s0 # m = m + 1

0111	0000	xxxxxxxx

rtr $a1 $s0

0110	1111	00000000

0000	0000	1111	0000

0111	1011	xxxxxxxx

j relPrimeLoop

0011	xxxx	00101000

gcd:

ltr $t0 0

0110	0100	00000000

add $t0 $t0

0000	0100	0100	0000

bneq $a0 subOne # if a != 0, go to subOne

0010	1010	01101100

rtr $v $a1

0110	1111	00000000

0000	1011	1111	0000

0111	1100	xxxxxxxx

jr $ra # return to loop

1010	1001	xxxxxxxx

subOne:

slt $a1 $a0

0000	1011	1010	0111

ltr $t0 1

0110	0100	00000001

bieq $t0 subTwo # if (a > b) go to subTwo

0001	0100	10001100

sub $a1 $a0 # b - a

0000	1011	1010	0001

ctr $a1 # b = b - a

0111	1011	xxxxxxxx

rtr $v $a0 # return a

0110	1111	00000000

0000	1010	1111	0000

0111	1100	xxxxxxxx

jr $ra

0101	1001	xxxxxxxx

subTwo:

sub $a0 $a1 # a - b

0000	1010	1011	0001

ctr $a0 # a = a- b

0111	1010	xxxxxxxx

rtr $v $a0 # return a

0110	1111	00000000

0000	1010	1111	0000

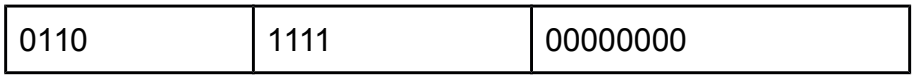
0111	1100	xxxxxxxx

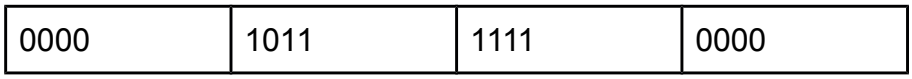
jr $ra

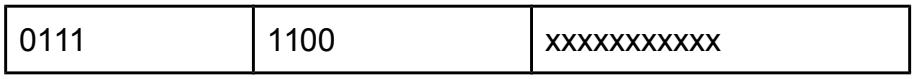
0101	1001	xxxxxxxx

cleanup:

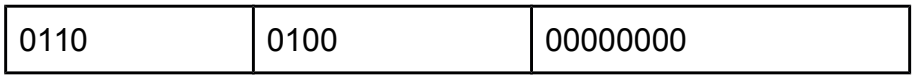
restore $s0 from stack
rtr $v $a1 # return m



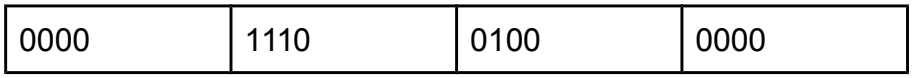




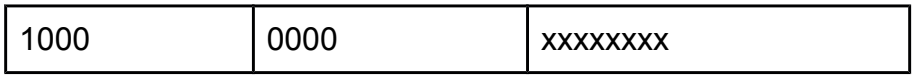
ltr $t0 0



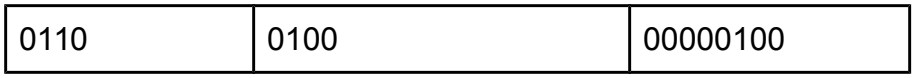
add $st $t0



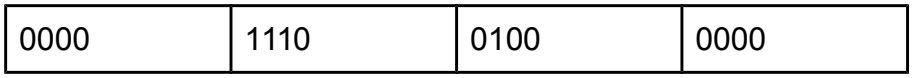
lw $s0



ltr $t0 4

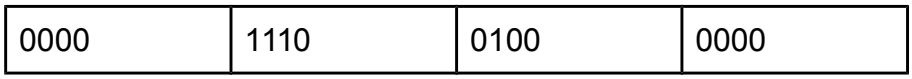


add $st $t0



lw $ra

add $st $t0



jr $ra