Processor Design

Group V

Ethan Liu
Wenzi Qian
Agnay Srivastava
Emma Letscher

Date

Table of Contents

Table of contents will be created when the document is finalized.

cool!
consider inserting
one during MZ.

Design Principles

The processor design is drawn based off the following:

1. *Make the common case fast*
2. *Good design demands good compromises*
3. *Smaller is faster*

Design

The design aims to do the following:

1. Provide as many general purpose registers as possible
2. Provide as much space for immediate in instructions as possible
3. Support use relative addressing to avoid unnecessarily large immediates (useful for programs that take up lots of space)

The instructions are structured as follows:

1. An instruction is 16 bits. The most significant bit is considered the start of an instruction.
2. The opcode is placed in the most significant bits in a continuous manner. The number of bits varies, but is always non-zero.
3. The registers used are specified in the least significant bits in a continuous manner. An instruction may use 0, 3, 6 or 9 bits for this purpose.
4. Bits between the opcode and the register is used for an immediate. If no registers are used, all bits after the opcode is used for the imeediate.

By default, the immediates will sign extend, with an exception: `LU`.

Because each instruction is 16 bits, and some instructions are intended to have 2 input registers and 1 destination register, it was not feasible to have 16 general purpose registers, as that will require 12 bits in the instruction. We opted to provide the processor with **8 general purpose registers**.

In addition to the general purpose registers, the processor also has three special registers, namely:

1. PC, program counter
2. SP, stack pointer
3. RA, return address

The process also has a 16 bit input and a 16 bit output. The inputs are accessed via instructions `READ` and `WRITE`.

In order to provide the maximum amount of general purpose registers to the programs, these registers are not accessed directly. Instead, there are special instructions (`GETSP`, `SETSP`, `CHGSP`, `CHGSPI`, `GETPC`, `SETPC`, `CHGPC`, `CHGPCI`) that handles interactions with these registers.

The processor also makes the assumption that a program must return to the parent that has called it. We also applied an optimization which combines multiple actions that are always performed when calling procedures into single instructions: `JUMP` and `RETRUN`. These instructions do the following:

`JUMP`:

1. Store current value in return address register at stack position 0
2. Store in return address register the new return address (program counter plus 1)
3. Change program counter by number of words specified by the sign-extended immediate

`RETURN`:

1. Set program counter to value currently in return address register
2. Set return address with value at stack position 0 (*after* stack pointer increment)
3. Change stack pointer by number of words specified by the sign-extended immediate

Note that the "set" actions will not be reflected until the next half cycle, meaning that until then, the outputs of the registers can be considered to be of the old values.

This design does lead to some implications. Position 0 on the caller's stack must be allocated and reserved for storing its parent's return address, and there is no instruction that directly allows access to the return address. Modification is not impossible - calling an empty procedure which modifies the return address written to memory is one way - but we are not making attempts to accomodate that functionality.

For memory access, we implemented two addressing modes:

1. `SET` and `RTV` uses the value in a register directly as the address for value
2. `SETSP` and `RTVSP` adds together the provided immediate (note the immediate is number of words and not bytes) with the stack pointer for the address to operate on.

It is currently assumed that the processor will only run a single program and put out results a single time at the top level, and so, we have implemented a `STOP` instruction to halt the processor and thus have it keep displaying the outputs.

Procedure calling convention for the processor is as follows:

1. Caller saves the a-values relevant for its usage
2. Callee saves and restores any s-values that are modified

Args? ret. addr? jump limits? How to jump/return?

Registers

Register	Alias	Usage
x0	a0	Return Address, Temporary, Return Value
x1	a1	Function Argument, Temporary, Return Value
x2	a2	Function Argument, Temporary, Return Value
x3	a3	Function Argument, Temporary, Return Value

x4	a4	Function Argument, Temporary, Return Value
x5	s0	Saved Value
x6	s1	Saved Value
x7	s2	Saved Value
		Program Counter
		Stack Pointer
		Return Address

The SP, PC, and RA are only accessible via their dedicated instructions

- For PC, `GETPC`, `SETPC`, `CHGPC`, and `CHGPCI`
- For SP, `GETSP`, `SETSP`, `CHGSP`, and `CHGSPI`
- For RA, `JUMP` and `RETURN`

These special register values are not modified directly via other register instructions.

Instruction Set

Instruction	**Desription**
`ADD`	Performs integer addition on the inputs
`SUB`	Performs integer subtraction on the inputs
`AND`	Performs bitwise AND on the inputs
`OR`	Performs bitwise OR on the inputs
`XOR`	Performs bitwise XOR on the inputs
`NAND`	Performs bitwise NAND on the inputs
`NOR`	Performs bitwise NOR on the inputs
`XNOR`	Performs bitwise XNOR on the inputs
`SHIFT`	Perform logical shift on the input
`ADDI`	Adds the register value
`EQ`	Performs a jump if the inputs equal
`LT`	Branches if the first input is less than the second
`NEQ`	Performs a jump if the first input is not equal to the second

GEQ	Performs a jump if the first input is greater or equal to the second
GETSP	Stores SP value to the destination register
SETSP	Sets SP value with value in input 1 register
CHGSP	Offsets SP value by value in input 1 register
CHGSPI	Offsets SP value by value of the sign-extended immediate
GETPC	Stores PC value to the destination register
SETPC	Sets PC value with value in input 1 register
CHGPC	Offsets PC value by value in input 1 register
CHGPCI	Offsets PC value by value of the sign-extended immediate
JUMP	Stores the current RA to position 0 of the stack pointer (It is a calling convention to reserve SP 0 as the return address). Set RA to the next instruction. Offset PC by the value of sign-extended immediate.
RETURN	Sets PC to RA. Release from the stack a number of words equal to the sign-extended immediate value. Restore the old RA from the 0 position of SP.
LU	Sets the upper half of the destination register to be the immediate value, while copying the lower half.
LL	Sets the destination register to be the sign-extended immediate value.
STR	Stores the value of input 2 to the memory address in input 1.
RTV	Gets the value in the memory address stored in input 1 and stores it in the destination register.
STRSP	Stores the value of input 2 to the memory address of SP adding sign-extended immediate value.
RTVSP	Gets the value in the memory address of SP adding sign-extended immediate value and stores it in the destination register.
READ	Read the value from the input line to the destination register.
WRITE	Write the value in input 1 to the output line.

Instruction Formats

Refer to the below tables for where the register specifiers are located in the instructions.

Symbol	T	R	E	M
Meaning	Destination/Output	Input 1	Input 2	Immediate Value

Symbol	O	1	2	3	4	7	A
Meaning	Opcode	Function Code 1	Function Code 2	Function Code 3	Function Code 4	Function Code 7	Function Code 10

```
Type 0RR:   OOOM MMMM MMEE ERRR
Type 1WR:   OOO1 MMMM MMRR RTTT
Type 1R:    OOO1 MMMM MMMM MRRR
Type 1W:    OOO1 MMMM MMMM MTTT
Type 2WR:   OOO1 2MMM MMRR RTTT
Type 2W:    OOO1 2MMM MMMM MTTT
Type 3:     OOO1 23MM MMMM MMMM
Type 4WRR:  OOO1 234E EERR RTTT
Type 7RR:   OOO1 2347 77EE ERRR
Type 7WR:   OOO1 2347 77RR RTTT
Type 10W:   OOO1 2347 77AA ARRR
Type 10R:   OOO1 2347 77AA ATTT
```

This is difficult to read quickly. consider using diagrams like the RISC-V green sheet!

<SE> indicates sign extension

Mnemonic	Opcode	F1	F2	F3	F4	F7	F10	Type	Description
ADD	000	0	0	0	0			4WRR	Dest = Input 1 + Input 2
AND	000	0	0	0	1			4WRR	Dest = Input 1 AND Input 2
OR	000	0	0	1	0			4WRR	Dest = Input 1 OR Input 2
XOR	000	0	0	1	1			4WRR	Dest = Input 1 XOR Input 2
SUB	000	0	1	0	0			4WRR	Dest = Input 1 - Input 2
NAND	000	0	1	0	1			4WRR	Dest = Input 1 NAND Input 2
NOR	000	0	1	1	0			4WRR	Dest = Input 1 NOR Input 2
XNOR	000	0	1	1	1			4WRR	Dest = Input 1 XNOR Input 2

SHIFT	001	1	0					2WR	Shift input 1 <SE> IMMEDIATE bits left and store in output. Fill with zeroes.
ADDI	001	0						1WR	Dest = Input 1 + <SE>IMM
EQ	100							0RR	If Input 1 = Input 2 PC = PC + <SE>IMM << 1
LT	101							0RR	If Input 1 < Input 2 PC = PC + <SE>IMM << 1
NEQ	110							0RR	If Input 1 != Input 2 PC = PC + <SE>IMM << 1
GEQ	111							0RR	If Input 1 >= Input 2 PC = PC + <SE>IMM << 1
GETSP	011	1	0	1	0	000	000	10R	Dest = SP
SETSP	011	1	0	1	0	100	001	10W	SP = Input 1
CHGSP	011	1	0	1	0	100	000	10W	SP = SP + Input 1
CHGSPI	011	1	1	0				3	SP = SP + <SE>IMM
GETPC	011	1	0	1	1	000	000	10R	Dest = PC
SETPC	011	1	0	1	1	100	001	10W	PC = Input 1
CHGPC	011	1	0	1	1	100	000	10W	PC = PC + Input 1
CHGPCI	011	1	1	1				3	PC = PC + <SE> IMM
JUMP	001	1	1	1				3	MEM[SP] = RA RA = PC+2 PC = PC + <SE>IMM << 1
RETURN	001	1	1	0				3	PC = RA SP = SP + <SE>IMM << 1 RA = MEM[SP]*
LU	000	1	0					2W	Dest[0:7] = IMMEDIATE[0:7] Dest[8:15] = Dest[8:15]
LL	000	1	1					2W	Dest = <SE>IMM
STR	011	1	0	0	0	010		7RR	MEM[Input 1] = Input 2
RTV	011	1	0	0	0	011		7WR	Dest = MEM[Input 1]

STRSP	010	0						1R	MEM[SP + <SE>IMM << 1] = Input1
RTVSP	010	1						1W	Dest = MEM[SP + <SE>IMM]
READ	011	1	0	0	0	100	000	10W	Output = IN
WRITE	011	1	0	0	0	100	001	10R	OUT = Input 1

* Note: RA uses the SP after addition as address to retrieve old RA.

Psuedo-Instructions

```
LI: Load Immediate
      Converts to LL, or LI and LU if the immediate is bigger than 8 bits.
STOP: Permanently halt the program until reset
      Can be implemented a number of ways: "EQ x0 x0 0" is one.
```

Instruction Assembly Examples

Type 0RR

EQ a0 a3 5

```
1000 0001 0100 0011
OOOM MMMM MMEE ERRR
```

Format & Breakdown:

```
OOO MMMMMMM EEE RRR
100 0000101 000 011
|   |       |   |
|   |       |   Input 2: 0b011 or x3
|   |       |
|   |       Input 1: 0b000 or x0
|   |
|   <SE> Immediate: 0b0000101 or 5
|
Opcode: 0b100
```

Type 1WR

ADDI x3 x5 -11

```
0010 1101 0110 1011
```

Format & Breakdown:

```
001 0 MMMMMM RRR TTT
001 0 110101 101 011
|   | |      |   |
|   | |      |   Output: 0b011 or x3
|   | |      |
|   | |      Input 1: 0b101 or x5
|   | |
|   | <SE> Immediate: 0b110101 or -11
|   |
|   F1: 0b0
|
Opcode: 001
```

Type 1R

STRSP s0 1
0100 0000 0000 1101

Format & Breakdown:
```
OOO 1 MMMMMMMMM RRR
010 0 000000001 101
|   | |         |
|   | |         Input 2: 0b101 or x5 (s0)
|   | |
|   | <SE> Immediate:
|   |
|   F1: 0b0
|
Opcode: 0b010
```

Type 1W

RTVSP s1 2
0101 0000 0001 0110

Format & Breakdown:
```
OOO 1 MMMMMMMMM TTT
010 1 000000010 110
|   | |         |
|   | |         Output: 0b110 or x6 (s1)
|   | |
|   | <SE> Immediate: 0b000000010 or 2 0b000000001 or 2 word, equivalent to 4 bytes.
|   |
|   F1: 0b1
|
Opcode: 0b010
```

Type 2W

LL a3 17
0001 1000 0000 1011

Format & Breakdown:
```
OOO 1 2 MMMMMMMM TTT
000 1 1 00010001 011
|   | | |        |
|   | | |        Output: 0b011 or x3 (a3)
|   | | |
|   | | <SE> Immediate: 0b00010001 or 17
|   | |
|   | F2: 0b1
|   |
|   F1: 0b1
|
Opcode: 0b000
```

Type 2WR

SHIFT a1 s2 8
0011 0010 0011 1001

Format & Breakdown:
```
OOO 1 2 MMMMM RRR TTT
001 1 0 01000 111 001
|   | | |     |   |
|   | | |     |   Output: 0b001 or x1 (a1)
|   | | |     |
|   | | |     Input 1: 0b111 or x7 (s2)
|   | | |
|   | | <SE> IMMEDIATE: 0b01000 or 8
|   | |
|   | F2: 0b0
|   |
|   F1: 0b1
|
Opcode: 0b001
```

Type 3

JUMP 8
0011 1100 0000 1000

Format & Breakdown:
```
OOO 1 2 3 MMMMMMMMMM
001 1 1 1 0000001000
|   | | | |
|   | | | <SE> Immediate: 0b000001000 or 8
|   | | |
|   | | F3: 0b1
|   | |
|   | F2: 0b1
|   |
|   F1: 0b1
|
Opcode: 0b001
```

Type 4WRR

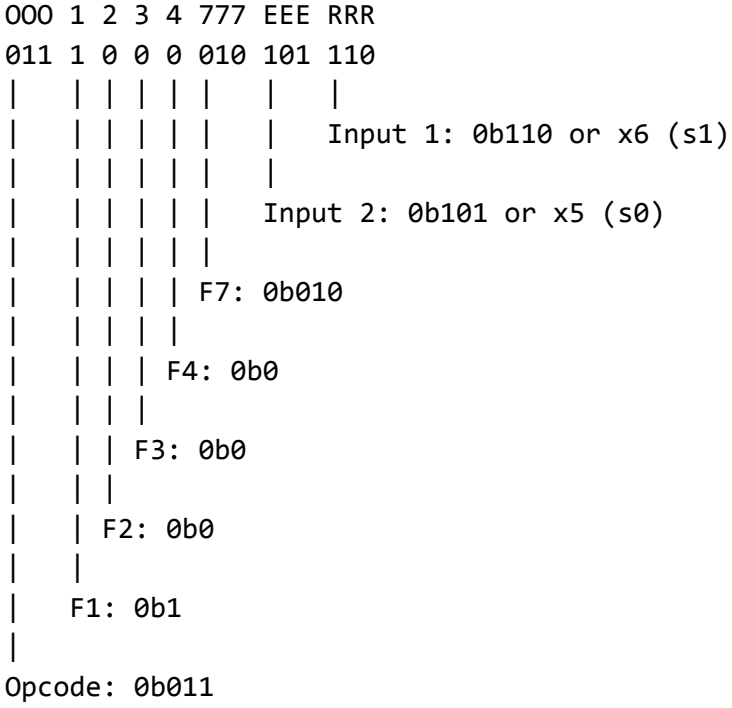
ADD x1 x3 x5
0000 0010 0101 1101

Format & Breakdown:
```
OOO 1 2 3 4 EEE RRR TTT
000 0 0 0 0 101 011 001
|   | | | | |   |   |
|   | | | | |   |   Output: 0b001 or x1
|   | | | | |   |
|   | | | | |   Input 1: 0b011 or x3
|   | | | | |
|   | | | | Input 2: 0b101 or x5
|   | | | |
|   | | | F4: 0b0
|   | | |
|   | | F3: 0b0
|   | |
|   | F2: 0b0
|   |
|   F1: 0b0
|
Opcode: 0b000
```

Type 7RR

STR s1 s0
0111 0000 1010 1110

Format & Breakdown:

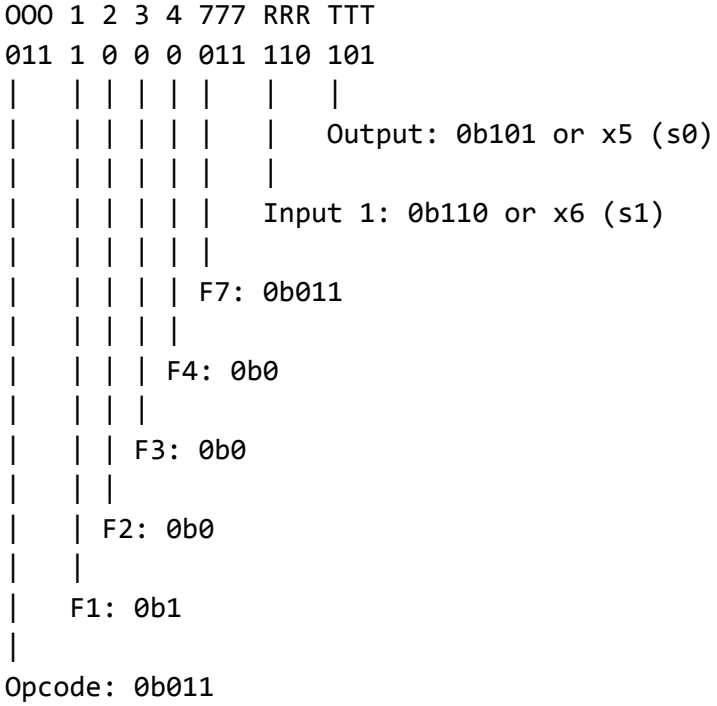


Type 7WR

RTV s0 s1

0111 0000 1111 0101

Format & Breakdown:



Type 10W

SETSP a0

0111 0101 0000 1000

Format & Breakdown:

```
OOO 1 2 3 4 777 AAA RRR
011 1 0 1 0 100 001 000
|   | | | | |   |   |
|   | | | | |   |   Input 1: 0b000 or x0 (a0)
|   | | | | |   |
|   | | | | |   F10: 0b001
|   | | | | |
|   | | | | F7: 0b100
|   | | | |
|   | | | F4: 0b0
|   | | |
|   | | F3: 0b1
|   | |
|   | F2: 0b0
|   |
|   F1: 0b1
|
Opcode: 0b011
```

Type 10R

GETSP a2

0111 0100 0000 0010

Format & Breakdown:

```
OOO 1 2 3 4 777 AAA TTT
011 1 0 1 0 000 000 010
|   | | | | |   |   |
|   | | | | |   |   Output: 0b010 or x2 (a2)
|   | | | | |   |
|   | | | | |   F10: 0b000
|   | | | | |
|   | | | | F7: 0b000
|   | | | |
|   | | | F4: 0b0
|   | | |
|   | | F3: 0b1
|   | |
|   | F2: 0b0
|   |
|   F1: 0b1
|
Opcode: 0b011
```

Single Input RelPrime

We demonstrate the workings of our processor with the following instruction:

```
void main() {
      int result = relPrime(30);
}

int relPrime(int n) {
   int m= 2;
   while (gcd(n, m) != 1) {  // n is the input from the outside world
     m = m + 1;
   }
   return m;
}

int gcd(int a, int b) {
  if (a == 0) {
    return b;
  }

  while (b != 0) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }

  return a;
}
```

Single Input RelPrime: Assembled

Note: The implementation here uses different memory architecture than specified for the final implementation of the processor. The instruction and data memory are separate.

Address	Assembly	Machine Code	Comments
0x0000	li a0 48	0001 1001 1000 0000	
0x0002	setsp a0	0111 0101 0000 1000	Initializing stack pointer to 96 bytes (48 words)
0x0004	li a0 6	0001 1000 1111 0000	Input 6
0x0006	jump 2	0011 1100 0000 0010	
0x0008	stop	1000 0000 0000 0000	Assembled as `eq x0 x0 0`; halts the program indefinitely.
			Start of RelPrime
0x000A	chgspi -3	0111 1011 1111 1101	Allocate 3 words of stack space for the program
			Saves on stack:
			(0) Relprime parent's return address. Saving this is automated by instruction `JUMP`.
0x000C	strsp s0 1	0100 0000 0000 1101	(1) Original Value in S0
0x000E	strsp s1 2	0100 0000 0001 0110	(2) Original Value in S1
0x0010	addi s0 a0 0	0010 0000 0000 0101	s0 stores input
0x0012	li s1 2	0001 1000 0001 0110	s1 stores current tested number
0x0016	li a3 1	0001 1000 0000 1011	# Load in the constant 1
0x001A	addi a0 s0 0	0010 0000 0010 1000	Prior to procedure call, store input a0 in s0
0x001E	addi a1 s1 0	0010 0000 0011 0001	Prior to procedure call, store input a1 in s1
0x0022	jump 8	0011 1100 0000	Go forward 8 words to call GCD

		1000	
0x0026	eq a0 a3 3	1000 0001 0100 0011	After return, the GCD is in a0. If the GCD equals 1, the number currently tested (s1) is relatively prime to our target. In this case, we jump forward 3 words to return
0x002A	addi s1 s1 1	0010 0000 0111 0110	Otherwise, we need to keep searching. Increment the current tested number.
0x002E	chgpci -5	0111 1111 1111 1011	Go back 5 words to the start of loop
0x0032	addi a0 s1 0	0010 0000 0011 0000	Put return value in a0
0x0036	rtvsp s0 1	0101 0000 0000 1101	Restore original value in s0
0x003A	rtvsp s1 2	0101 0000 0001 0110	Restore original value in s1
0x003E	return 3	0011 1000 0000 0110	Return to parent (main) and moves stack pointer by 3 words
0x0042	li a2 0	0001 1000 0000 0110	Load constant 0 for comparison
0x0046	neq a2 a0 3	1100 0000 1101 0000	If a0 is already zero, simply return a1.
0x004A	addi a0 a1 0	0010 0000 0000 1000	Load a1 into a0 for return
0x004E	return 0	0011 1000 0000 0000	Return to parent (RelPrime). No stack space was allocated, so none needs returned.
0x0052	eq a1 a2 6	1000 0001 1000 1100	If b equals 0, the loop is done. Jump to the end.
0x0056	geq a1 a0 3	1110 0001 1000 1001	Go to the else clause if a is less than or equal to b. Otherwise, proceed to then clause.
0x005A	sub a0 a0 a1	0000 1000 0100 0000	Subtract b from a
0x005E	chgpci -3	0111 1111 1111 1101	Repeat loop
0x0062	sub a1 a1 a0	0000 1000 0000 1001	Subtract a from b
0x0066	chgpci -5	0111 1111 1111 1011	Repeat loop

is this gcd?

0x006A	return 0	0011 1000 0000 0000	Return to parent (RelPrime). No stack space was allocated, so none needs returned.

Performance

We are choosing to measure performance based on the execution speed of programs, including relprime. To do this, the instructions are all planned to be single cycle, and optimization specifically for procedure calling is made.

Memory Map

12 bit addresses?

Start Byte	End Byte	Purpose
`0x000`	`0x01F`	Reserved
`0x020`	`0x1FF`	Instruction
`0x200`	`0x400`	Data

The “Reserved” section is for initializations necessary, such as instruction that stores the initialization of sp and pc. Not all space may be used.

Examples of Common Operations

Iteration

The following code segment is roughly equivalent to this Java code:

```
int x = 7;
for(int i = 0; i<10; i++){
   x = x + 2;
}
```

```
================================================================================
=

# a0 = x, initialized to 7
li a0 7

# a1 = 10
li a1 10

# a2 = i, initialized to 0
li a2 0

# if i>= 10, we are done, jump forward 4 instructions
geq a2 a1 4

# a += 2
addi a0 a0 2

# i++
addi a2 a2 1

# Go to start of loop
chgpci -3
```

Loading Large Immediates

```
int number = 12007;

12007 = 0b0010111011100111
================================================================================
=

# Loads lower 8 bits into a0 (0b11100111 = 231)
ll a0 231
```

```
# Loads upper 8 bits into a0 (0b00101110 = 46)
lu a0 46
```

Conditionals (if/else/then)

The following assembly is roughly equivalent to this Java code:

```
if(x >= y){
      x++;
}
else {
      x–;
}

================================================================================
=

# Assume the value of x is stored in a0 and the value of y is stored in a1

# If x is less than y, add 3 to program counter (else block)
lt a0 a1 3

# x++
addi a0 a0 1

# Skip over else block
chgpci 2

# x--
addi a0 a0 -1
```

Assemble me

Procedure Calling

```
void main() {
	int product = mult(9, 15);
}

int mult(int a, int b) {
	int result = 0;
	for(int i = 0; i < b; i++){
		result+=a;
	}
	return result;
}

================================================================================
=

# Main
li a0 9
li a1 15
jump 2
stop

# Multiplication
# Arguments in a0 and a1
# Result in a2
# i in a3
# Nothing on stack
li a2 0
li a3 0

# If i >= b, leave loop and return
geq a3 a1 4

# Result += a
add a2 a2 a0

# i++
addi a3 a3 1

# Loop back
chgpci -3

# Store return value in a0
addi a0 a2 0
return 0
```

Backup & Restore of Saved Registers

```
void main() {
	int seriesSum = getSeriesSum(7);
}

int getSeriesSum(int in) {
	if (in == 0) {
		return in;
	}
	int myNum = in;
	in = in - 1;
	return (myNum + getSeriesSum(in));
}
```

Note that this is a flawed program that does not work when the input is negative. This is reflected in the assembled program as well.

```
================================================================================
=

# Main
li a0 7
jump 2
stop

# Get Series Sum
# Arguments in a0 and a1
# Result in s0
# i in s1
# Stores on stack:
# (0) Return Address
# (1) Previous value in s0

li a1 0

# Skip the if block if input does not equal 0, otherwise, return the input.
neq a0 a1 2
return 0

# Allocate space on stack
chgspi -2

# Back up s0
strsp s0 1

# Back up myNum
addi s0 a0 0
```

```
# in = in - 1
addi a0 a0 -1

# Call itself
jump -6

# Add myNum to return value
add a0 a0 s0

# Put back old s0
rtvsp s0 1

# Go to parent
return 2
```