Team 2-2 Design Document

Team Members: Greg Wilke, Aditya Kapoor, Charles Penn,
Stephen Dupal, (former member Nathan Marks)

Table of Contents

I. The ANCGS Assembly Language Page 2
Basic Overview Page 2
Assembly Language Commands Page 2
General Purpose Registers Page 4
Special Purpose Registers… Page 5
Converting Assembly Language… Page 5
Programs That Test Commands Page 6

Part I. The ANCGS Assembly Language

Basic Overview:

The ANCGS assembly language is designed for the purpose of operating with a system with a 16-bit data bus and a 16-bit address bus. All commands used in the language have a length of 16 bits in machine code. All values used in ANCGS are two's complement numbers. A word in memory is 16 bits or 2 bytes long and takes two memory spaces. If an instruction is located at 0x0002, the next is at 0x0004.

There are very few differences in writing a program in ANCGS and MIPS. One of the more noticeable differences is the use of the @ symbol to indicate comments instead of the # symbol.

Assembly Language Commands:

All commands in the ANCGS assembly language fall into the following formats, 3-register (3-Reg), 2-register (2-Reg), immediate value (Imm), and jump (Jump). The commands used in the ANCGS assembly language are the following:

Format	Name	Command	Machine Code	Example	Definition
3-Reg	Add	add	0000	add $1, $2, $3	$1 + $2 = $3
3-Reg	Subtract	sub	0001	sub $1, $2, $3	$1 - $2 = $3
3-Reg	AND	and	0010	and $1, $2, $3	Performs logical AND on $1 and $2 and stores result in $3.
3-Reg	OR	or	0011	or $1, $2, $3	Performs logical OR on $1 and $2 and stores result in $3.
Imm	Shift Left Logical	sll	0100	sll $1, x, $3	Shifts bits of $1 left x times and stores result in $3.
2-Reg	Load Word	lw	0101	lw $1, $3	Load value in address indicated in $1 into $3.
2-Reg	Store Word	sw	0110	sw $3, $1	Store value in address indicated in $3 in $1.
Imm	Load Upper Immeadiate	lui	0111	lui x, $3	Load 8-bit immediate value(x) in upper 8-bits of $3.
Imm	Branch Equal	beq	1000	beq $1, $2, L	If $1 = $2 then branch to address L.

Imm	Branch Not Equal	bne	1001	bne $1, $2, L	If $1 ≠ $2 then branch to address L.
3-Reg	Set Greater Than	sgt	1010	sgt $1, $2, $3	If $1 > $2 then set $3 = 1; else $3 = 0.
Jump	Jump Register	jr	1011	jr $1	Jump to address stored in $1 (set PC to value in $1).
Jump	Jump and Link	jal	1100	jal L	Set $ra to address of next command and jump to address L.
Imm	Load Immediate	li	1101	li x, $1	Stores an immediate value(x) 8-bits long into $1 and zeros upper 8-bits.
2-Reg	Special Purpose Load	spl	1110	spl EPC	Loads value from specified SP Register and stores it in $at
2-Reg	Special Purpose Store	sps	1111	sps $1, Display	Takes value in $1 and stores it in the specified SP Register

Notes:

1. There is also one pseudo-code command, load address:

la L, $1 Store memory address of L in $1.

It is accomplished through the following commands:

li x, $1 x = the lower 8 bits of L's address
lui y, $1 y = the upper 8 bits of L's address

2. The jump and link command is a PC-relative command and works like the branching commands, a twelve bit, 2's complement number indicates how far the command the label refers to is from the command following the jump and link command. The address of the command after jal is stored in $ra.

3. The jal, beq and bne commands become pseudo-code when they have to branch to an instruction that is farther away than the 4-bits reserved in beq and bne and 12-bits in jal can handle.

beq $1, $2, L

The operation is then accomplished through the following commands:

```
bne     $1, $2, 0x3     @This would be beq if the pseudo-command was bne.
li      x, $at          @Basically, this is load address of L.
lui     y, $at
jr      $at             @Jumps to instruction.
```

The pseudo-code is similar for jal except the branch command is replaced with:

```
li      x, $at          @Loads address of command after code pseudo-command.
lui     y, $at
sub     $ra, $ra, $ra   @Clears old values in $ra.
add     $ra, $at, $ra   @Stores address in $ra
```

General Purpose Registers:

There are 16 general purpose registers. There purposes are arranged in the following chart (the addresses are in binary):

Register	Name	Address	Function
$0	$zero	0000	This register is permanently set to 0.
$1	$t0	0001	This register is for use by the program for storing values.
$2	$t1	0010	This register is for use by the program for storing values.
$3	$t2	0011	This register is for use by the program for storing values.
$4	$t3	0100	This register is for use by the program for storing values.
$5	$t4	0101	This register is for use by the program for storing values.
$6	$t5	0110	This register is for use by the program for storing values.
$7	$t6	0111	This register is for use by the program for storing values.
$8	$s0	1000	This register is for use by the program for saving values in a program by a method besides main.
$9	$s1	1001	This register is for use by the program for saving values in a program by a method besides main.
$10	$v0	1010	This register is for use by the program for saving values that are put out by a method.
$11	$v1	1011	This register is for use by the program for saving values that are put out by a method.
$12	$a0	1100	This register is for use by the program for saving values that will be used in another method.
$13	$a1	1101	This register is for use by the program for saving values that will be used in another method.
$14	$ra	1110	This stores the address of the next instruction after a jal instruction.
$15	$at	1111	This is for use by the assembler to handle pseudo-code instructions.

Special Purpose Registers and Interrupt Handling:

All special purpose registers are 16-bits long. These are the following special purpose registers used in the processor:

Name	Symbol	Machine Code	Definition
Program Counter	PC	none	This register keeps track of which instruction is currently being executed.
Exception Program Counter	EPC	0000	This register keeps track of the instruction that the main program was at when an exception occurred.
Instruction Register	IR	none	This register stores the instruction currently being used.
Display Register	DIS	0001	This register stores values to be displayed to the user.
Data Register	DR	0010	This register stores a value input by the user.
Interrupt Type Register	IT	0011	The type of interrupt is stored in this register as number.

There are two types of interrupts that ANCGS handles, the interrupt that indicates that someone is loading a value into the Data Register (interrupt 1) and the interrupt that indicates that someone is telling the processor to activate the main program (interrupt 2). The values that indicate these interrupts (1 and 2) are stored in the Interrupt Type Register. When the user attempts to load a number, IT is set to 1 and then a method in the interrupt shifts the current value in $at over 4 bits and inserts the new four bit value into bits 0:3 of $at.

The program that handles the interrupts is placed at the end of the relatively prime program and is located on page 11.

Converting Assembly Language into Machine Language:

As stated earlier, all commands in ANCGS are 16-bits long. The first four bits contain the opcode and tells the machine what it is supposed to do. The next twelve bits are for the registers to acquire data from or transfer data to. Here are the examples of translated commands:

add	$1	$2	$3
0000	0001	0010	0011

So the binary string for this command is 0000000100100011.
The first two registers listed in the command are the registers that hold the numbers to add together. The result of the fuction is placed in the third register listed in the command. This example is how 3-register commands are translated.

li	27	$t5
1101	00001111	0110

So the binary string for this command is 1101000011110110.
If any command requires something to be stored in a register, the address of that register is always placed in the last four bits of a command. This is how immediate commands with only one register are handled.

lw	$1		$3
0101	0001	0000	0011

So the binary string for this load word command is 0101000100000011.
Since the command only needs two registers and the address of any register that has something written to it in the command is placed in the last four bits, the third group of bits (7:4) is filled with the value 0. This is how all 2-register commands are translated.

jr	$1		
1011	0001	0000	0000

So the binary string of this jump register command is 1011000100000000.
Since the command only needs one register and nothing is being stored in a register, the address of the register is placed in bits 11:8 and the last eight bits are set to 0. This is how the jump commands are translated except that all of the last twelve bits are used in jal.

beq	$1	$2	L
1011	0001	0010	1111

So the binary string of the branch equal command is 1011000100101111.
The last four bits in the beq and bne commands indicate how many instructions L is away from the instruction after the branch command. In the above example, L is -8 instructions away from the instruction after beq (you go up the page 8 instruction, PC = PC – 16). This is how the immediate commands that deal with branching are handled when they are not pseudo-commands.

sll	$1	7	$3
0100	0001	0111	0011

So the binary string of the shift left logical command is 0100000101110011.
As shown in this example the program can only shift at most 17 places. This is how immediate commands with two registers are handled.

The Programs that Test All Commands:

The program for determining relatively prime numbers and the interrupt service routine.

Addr	Machine Language	ANCGS	ANCGS Comments
		main:	
0000	0000 0000 0000 0100	add $zero, $zero, $t3	@ loads 0 into $3

0002	0000 0000 0000 0101	add $zero, $zero, $t4	@ ensures that $t4 is set to 0
0004	1000 0000 0100 1001	beq $zero, $t3, main	@ returns to the top of main if $3 is still 0
0006	1000 0000 0101 1100	beq $zero, $t4, main	@ returns to the top of main if $3 is still 0
0008	0000 1111 0000 1000	add $at, $s0, $s0	@ stores the value of the first input whose relatively prime number is to be found
000a	1101 0000 0010 1001	li 2, $s1	@ stores the value of B in $s1
000c	1101 0000 0001 0010	li 1, $t1	@ stores the value of 1 in $t1
		loop:	
000e	0000 0000 1000 0011	add $zero, $s0, $t2	@ stores the value of A in $t2
0010	0000 0000 1001 0100	add $zero, $s1, $t3	@ stores the value of B in $t3
0012	1100 0000 0000 1001	jal gcd	
0014	1001 1010 0010 0010	bne $v0, $t1, subloop	@ if the gcd is not 1 it runs subloop
0016	1111 1001 0000 0011	sps $s1, DIS	@ special purpose store to the display special purpose register
0018	1011 1110 0000 0000	jr $zero	@ ends the program, returning to the interrupt
		subloop:	
001a	0000 1001 0010 1001	add $s1, $t1, $s1	@ increments b by 1
001c	1101 0000 1110 0110	la loop, $t5	@ stores the address of loop in $t5
001e	0111 0000 0000 0110		
0020	1011 0110 0000 0000	jr $t5	@ runs the loop calling the gcd function
		gcd:	
0022	0000 0000 0000 0101	add $zero, $zero, $t4	@ ensures that $t4 is set to 0
0024	1010 0100 0000 0101	sgt $t3, $zero, $t4	@ sets $t4 to 1 if $t3 > $t0
0026	1000 0101 0010 0010	beq $t4, $t1, gcdloop	@ if $t4 is set to 1, i.e. b is greater than 0, go to gcdloop
0028	0000 0000 0011 1010	add $zero, $t2, $v0	@ sets $v0 to the value of a
002a	1011 1110 0000	jr $ra	

	0000		
		gcdloop:	
002c	0000 0000 0000 0101	add $zero, $zero, $t4	@ sets $t4 to 0
002e	1010 0100 0011 0101	sgt $t3, $t2, $t4	@ sets $t4 to 1 if B is greater than A
0030	1000 0101 0010 0100	beq $t4, $t1, setvalloop	@ if B > A (i.e. $t4 = 1), go to setvalloop
0032	0001 0011 0100 0011	sub $t2, $t3, $t2	@ performs a = (a-b)
0034	1101 0010 0010 0110	la gcd, $t5	@ stores the address of gcd in $t5
0036	0111 0000 0000 0110		
0038	1011 0110 0000 0000	jr $t5	@ runs the gcd loop again (i.e. tests whether b >0)
		setvalloop:	
003a	0001 0101 0110 0110	sub $t4, $t5, $t5	@ clears the value of $t5 to 0
003c	0000 0000 0011 0110	add $zero, $t2, $t5	@ sets the temporary register $t5 to the value of A
003e	0000 0000 0100 0011	add $zero, $t3, $t2	@ a=b
0040	0000 0000 0110 0100	add $zero, $t5, $t3	@ b=original a
0042	1101 0010 0010 0110	la gcd, $t5	@ stores the address of gcd in $t5
0044	0111 0000 0000 0110		
0046	1011 0110 0000 0000	jr $t5	@ runs the gcd loop again (i.e. tests whether b >0)

		interrupt:	
0080	1110 0011 0000 0001	spl IT, $t0	@ gets the interrupt type from the special purpose Interrupt Type Register
0082	1101 0000 0001 0010	li 1, $t1	@ loads the value 1 to $t1
0084	1000 0001 0010 0010	beq $t0, $t1, loadval	@ jumps to loadval if $t0 = $t1
0086	1101 0000 0010 0010	li 2, $t1	@ loads the value 2 to $t1

0088	1001 0001 0010 0101	bne $t0, $t1 return	@ if $t0 does not equal $t1, it jumps to return
		loadval:	
008a	0100 1111 0100 1111	sll, $at, 4, $at	@ multiplies $at by 16 (shifts numbers over 4 bits)
008c	1110 0010 0000 0001	spl DR, $t0	@ gets the input from the user which was stored in the special purpose Data Register
008e	0011 1111 0001 1111	or $at, $t0, $at	@ or's the values in $at and $t0
0090	1110 0000 0000 0001	spl EPC, $t0	@ loads the value from EPC to $t0
0092	1011 0001 0000 0000	jr $t0	@ returns to main
		return:	
0094	1110 0011 0000 0100	spl IT, $t3	@ stores the interrupt type from the Interrupt Type Register in $t3
0096	1110 0000 0000 0001	spl EPC, $t0	@ loads the value from EPC to $t0
0098	1011 0001 0000 0000	jr $t0	@ returns to main

Addr	Machine Language	ANCGS	ANCGS Comments
		main:	
0000	0000 0000 0000 0100	add $zero, $zero, $t3	@ loads 0 into $3
0002	0000 0000 0000 0101	add $zero, $zero, $t4	@ ensures that $t4 is set to 0
0004	1000 0000 0100 1001	beq $zero, $t3, main	@ returns to the top of main if $3 is still 0
0006	1000 0000 0101 1100	beq $zero, $t4, main	@ returns to the top of main if $3 is still 0
0008	0000 1111 0000 1000	add $at, $s0, $s0	@ stores the value of the first input whose relatively prime number is to be found
000a	1101 0000 0010 1001	li 2, $s1	@ stores the value of B in $s1
000c	1101 0000 0001 0010	li 1, $t1	@ stores the value of 1 in $t1
		loop:	
000e	0000 0000 1000 0011	add $zero, $s0, $t2	@ stores the value of A in $t2
0010	0000 0000 1001 0100	add $zero, $s1, $t3	@ stores the value of B in $t3
0012	1100 0000 0000 1001	jal gcd	
0014	1001 1010 0010 0010	bne $v0, $t1, subloop	@ if the gcd is not 1 it runs subloop
0016	1111 1001 0000 0011	sps $s1, DIS	@ special purpose store to the display special purpose register

0018	1011 1110 0000 0000	jr $zero	@ ends the program, returning to the interrupt
		subloop:	
001a	0000 1001 0010 1001	add $s1, $t1, $s1	@ increments b by 1
001c	1101 0000 1110 0110	la loop, $t5	@ stores the address of loop in $t5
001e	0111 0000 0000 0110		
0020	1011 0110 0000 0000	jr $t5	@ runs the loop calling the gcd function
		gcd:	
0022	0000 0000 0000 0101	add $zero, $zero, $t4	@ ensures that $t4 is set to 0
0024	1010 0100 0000 0101	sgt $t3, $zero, $t4	@ sets $t4 to 1 if $t3 > $t0
0026	1000 0101 0010 0010	beq $t4, $t1, gcdloop	@ if $t4 is set to 1, i.e. b is greater than 0, go to gcdloop
0028	0000 0000 0011 1010	add $zero, $t2, $v0	@ sets $v0 to the value of a
002a	1011 1110 0000 0000	jr $ra	
		gcdloop:	
002c	0000 0000 0000 0101	add $zero, $zero, $t4	@ sets $t4 to 0
002e	1010 0100 0011 0101	sgt $t3, $t2, $t4	@ sets $t4 to 1 if B is greater than A
0030	1000 0101 0010 0100	beq $t4, $t1, setvalloop	@ if B > A (i.e. $t4 = 1), go to setvalloop
0032	0001 0011 0100 0011	sub $t2, $t3, $t2	@ performs a = (a-b)
0034	1101 0010 0010 0110	la gcd, $t5	@ stores the address of gcd in $t5
0036	0111 0000 0000 0110		
0038	1011 0110 0000 0000	jr $t5	@ runs the gcd loop again (i.e. tests whether b >0)
		setvalloop:	
003a	0001 0101 0110 0110	sub $t4, $t5, $t5	@ clears the value of $t5 to 0
003c	0000 0000 0011 0110	add $zero, $t2, $t5	@ sets the temporary register $t5 to the value of A
003e	0000 0000 0100 0011	add $zero, $t3, $t2	@ a=b
0040	0000 0000 0110 0100	add $zero, $t5, $t3	@ b=original a
0042	1101 0010 0010 0110	la gcd, $t5	@ stores the address of gcd in $t5
0044	0111 0000 0000 0110		
0046	1011 0110 0000 0000	jr $t5	@ runs the gcd loop again (i.e. tests whether b >0)

		interrupt:	
0080	1110 0011 0000 0001	spl IT, $t0	@ gets the interrupt type from the special purpose Interrupt Type

			Register
0082	1101 0000 0001 0010	li 1, $t1	@ loads the value 1 to $t1
0084	1000 0001 0010 0010	beq $t0, $t1, loadval	@ jumps to loadval if $t0 = $t1
0086	1101 0000 0010 0010	li 2, $t1	@ loads the value 2 to $t1
0088	1001 0001 0010 0101	bne $t0, $t1 return	@ if $t0 does not equal $t1, it jumps to return
		loadval:	
008a	0100 1111 0100 1111	sll, $at, 4, $at	@ multiplies $at by 16 (shifts numbers over 4 bits)
008c	1110 0010 0000 0001	spl DR, $t0	@ gets the input from the user which was stored in the special purpose Data Register
008e	0011 1111 0001 1111	or $at, $t0, $at	@ or's the values in $at and $t0
0090	1110 0000 0000 0001	spl EPC, $t0	@ loads the value from EPC to $t0
0092	1011 0001 0000 0000	jr $t0	@ returns to main
		return:	
0094	1110 0011 0000 0100	spl IT, $t3	@ stores the interrupt type from the Interrupt Type Register in $t3
0096	1110 0000 0000 0001	spl EPC, $t0	@ loads the value from EPC to $t0
0098	1011 0001 0000 0000	jr $t0	@ returns to main

```
@ SillyProgram
@
@ Author: Greg Wilke, 1/6/04
@
@ This program was created to test the commands in our assembly language not used in our
@ relatively prime program.

        .text
        .globl main

main:
        li      7, $1           @Loads the value 7 into register $1.
        lui     127, $4         @Loads the value of 100 into the upper 8 bits of $4.
        sll     $1, 1, $2       @Shifts the value in $1 left 1 bit and stores the result in $2.
        and     $1, $2, $3      @$1 AND $2, the result is stored in $3.
        or      $1, $4, $4      @$1 OR $4, the result is stored in $4.
```

Machine Code Translation:

```
        0111 0111 1111 0100             0x0000

        1101 0000 0111 0001             0x0002
```

0100 0001 0001 0010	0x0004
0010 0001 0010 0011	0x0006
0011 0001 0100 0100	0x0008