ISA Project

W. Clayton Pannell

CPE 531: Intro Computer Architecture

12/01/20

Table of Contents

Summary of Goals

- Minimal cpi low cost RISC ISA
  - signed 16 bit words. max 16 instructions.
- Linear address 2^10 memory bytes, word addressable (2^9 words)
  - 4 bit fors opcode (16 possible ops)
- "Compile" C operations to assembly and machine code:
  - add, subtract, and, or
  - assignment
  - control flow (if-else, while, for, w/ operators ==, !=, >, <=, <, >=)
  - 2's complement signed 16bit int (int a;), 1D array of signed int (int a[11];)

Introduction

This document, and included files, make up a verilog implementation of a low cost, low instruction count Instruction Set Architecture (ISA) that supports common high level (or at least C) language constructs. The instruction set centers around 16 bit signed data and uses 16 bit instructions. No attempt has been made to run this ISA outside of a simulator, much less on live hardware.

Instructions

This Instruction Set Architecture uses only 16 Instructions. Each instruction completes execution in 1 full cycle. It is worth noting that the program counter output for instruction N occurs during the last quarter of instruction N-1's cycle.

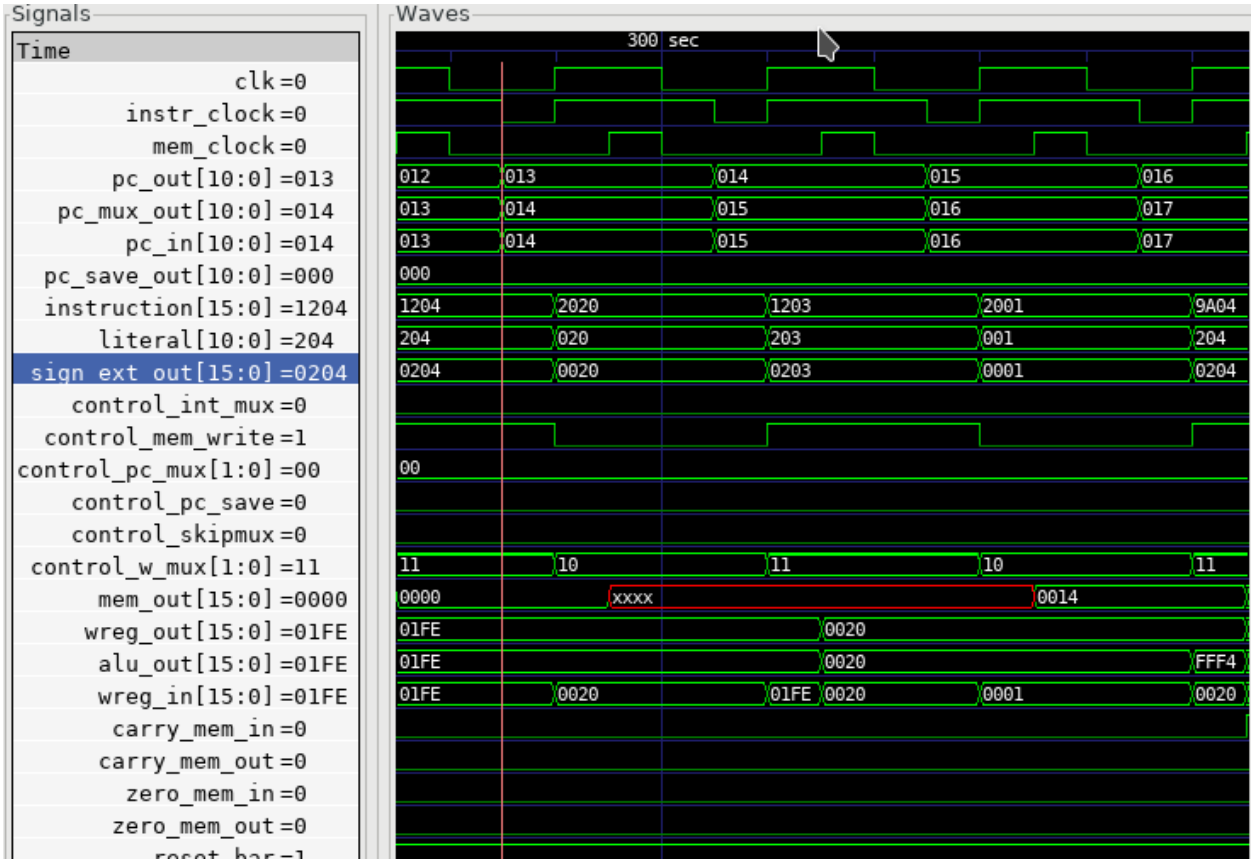

Fig. 1. Verilog Simulation Output

Instruction Word Decoding

This ISA uses a single instruction format. Each instruction consists of a 5 bit opcode and an 11 bit literal value.

Opcode	Literal
15:11	10:0

The Opcode is further broken down into the Instruction Code and the Destination bit.

Instruction code	Dest (W/M)
15:12	11

Assembly instructions typically consist of the mnemonic, a literal value, and the destination. The literal value is a numeric literal, although preprocessor definitions are highly recommended for variable names. The assembly code:

```
w equ 0
foo equ 0x001
add foo,w
```

would result in:

```
add 0x001,0
```

after preprocessing which would be assembled into the machine code value 0x8001.

Instruction Listing

Instr code	Mnem-onic	Description	Affects Status Regs	Usage
0	mm	move mem/reg to w or self moving into self can be used to check for Zero value of mem/reg	Zero	mm 0x21,w mm 0x22,m
1	mwm	move w into mem/reg		mwm 0x21
2	mlw	move 11bit sign extended literal into W register		mlw 0x01
3	rlm	rotate mem/reg left (through carry) store result in w or mem/reg	Carry	rlm 0x20,w rlm 0x21,m
4	rrm	rotate mem/reg right (through carry) store result in w or mem/reg	Carry	rrm 0x20,w rrm 0x21,m
5	awm	bitwise AND w with mem/reg store result in w or mem/reg	Zero	awm 0x21,w awm 0x21,m
6	owm	bitwise OR w with mem/reg store result in w or mem/reg	Zero	owm 0x21,1 owm 0x21,m
7	xwm	bitwise XOR w with mem/reg store result in w or mem/reg	Zero	xwm 0x21,w xwm 0x21,m
8	add	add w with mem/reg store result in w or mem/reg	Carry Zero	add 0x20,w add 0x21,m
9	sub	subtract w from mem/reg (mem/reg - w) store result in w or mem/reg	Carry Zero	sub 0x20,w sub 0x21,m
A	sms	skip next instruction if value at mem/reg address is nonzero		sms 0x20
B	smc	skip next instruction if value at mem/reg address is zero		smc 0x20
C	gol	goto literal instruction mem address		gol 0x005
D	gow	goto instruction mem address held in w		gow
E	wfi	Halt Program execution until next interrupt		wfi
F	rfi	return from interrupt (restores PC to previous value + 2)		rfi

Instruction Details

The table below shows how the assembly code is translated into machine code. All values are displayed in binary format. The D symbol denotes the Destination bit. The M symbol denotes that the literal value is a data memory address. The P symbol denotes that the literal value is a program memory address. The X symbol denotes that the literal value is a sign extended number. The ? symbol denotes that the value is ignored. The assembler will default to making these values 0. Note that the meanings of different literal values are determined in the instruction decode module. The use of the symbols here is only to better convey understanding. see the instruction decode section for more details. For "real-world" examples see the program.mem file included with this document. This file contains C code that was hand compiled and hand assembled to machine code.

Asm Format	Instr. Code	Dest.	Literal	Machine Code
mm M,D	0000	D	MMM_MMMM_MMMM	0000_XMMM_MMMM_MMMM
mwm M	0001	?	MMM_MMMM_MMMM	0001_?MMM_MMMM_MMMM
mlw X	0010	?	XXX_XXXX_XXXX	0010_?XXX_XXXX_XXXX
rlm M,D	0011	D	MMM_MMMM_MMMM	0011_DMMM_MMMM_MMMM
rrm M,D	0100	D	MMM_MMMM_MMMM	0100_DMMM_MMMM_MMMM
awm M,D	0101	D	MMM_MMMM_MMMM	0101_DMMM_MMMM_MMMM
owm M,D	0110	D	MMM_MMMM_MMMM	0110_DMMM_MMMM_MMMM
xwm M,D	0111	D	MMM_MMMM_MMMM	0111_DMMM_MMMM_MMMM
add M,D	1000	D	MMM_MMMM_MMMM	1000_DMMM_MMMM_MMMM
sub M,D	1001	D	MMM_MMMM_MMMM	1001_DMMM_MMMM_MMMM
sms M	1010	?	MMM_MMMM_MMMM	1010_?MMM_MMMM_MMMM
smc M	1011	?	MMM_MMMM_MMMM	1011_?MMM_MMMM_MMMM
gol P	1100	?	PPP_PPPP_PPPP	1100_?PPP_PPPP_PPPP
gow	1101	?	???_????_????	1101_????_????_????
wfi	1110	?	???_????_????	1110_????_????_????
rfi	1111	?	???_????_????	1111_????_????_????

Instruction Justification

One of the goals of this project was to use only 16 instructions in the ISA. This restriction required strong justifications for what instructions made it into the ISA. The bare minimum instructions required by this single register architecture to do anything are the memory/register manipulation instructions: mm, mwm, and mlw. These instructions handle moving data into and out of memory, as well as setting up operands for all other instructions. The alternative to not having a way to instantiate a literal value is too grim to consider.

The next easiest instructions to add were the arithmetic instructions: add, sub, awm, owm. These basic instructions were explicitly required to be present. The rotate/shift instructions, rlm and rrm, are needed in order to implement power-of-two mutliplication and divison, which, although not explicitly required, are nearly as ubuiquitous as the basic arithmetic instructions. The xwm (XOR) was also not explicitly needed, but is frequently needed in communications applications, negation, and it rounded out the bitwise boolean operations nicely. A strong contender for it's position was a complement instruction, but xwm could do the same job and more.

The harder decisions to make were the control flow instructions. A literal goto (gol) was needed to make jumps happen, and represents the basis of a function call. A branch or computed goto would also be needed to make function call returns possible. The computed goto (gow) was chosen because it was much easier to use for function call purposes, and doing lookup tables would only slightly more painful than with a branch instruction. Once cost is brought into the equation, gow becomes a much clearer winner since it fills the 4th slot in the 4 way Program Counter Mux. Implementing a branch instruction would require adding another mux between the skip mux and the Adder module.

The sms and smc "skip" instructions pair with the carry and zero ALU status registers to build rudimentary comparison operations (less than, greater than, equal, etc.). These operations are the building block of comparison-based control flow operations (if, else, while, for, etc.). Their inclusion is required, although their operation for this pupose is admittedly painful, especially when dealing with mixed sign operands (see the register section for more detail).

One of the requirements was to have a halt instruction. The wfi instruction implements this, and could be further augmented into a low power sleep mode by disabling any peripherals by piggy-backing off the Int_Mux control line, if needed. Since the ISA now has the ability to interrupt, it needs a way to return from the interrupt. This functionality is provided by the rfi instruction which restores the program counter from the PC Save register.

If the 16 instruction restriction were lifted these are the operations that would be nice to have, in order of importance: increment/decrement memory (easier for loops), skip on less/greater than (easier signed comparison), branch to Wreg value, branch to literal value, add/subtract/and/or/xor Wreg with literal, load indirect memory access value and increment/decrement pointer by literal.

Architecture

Architecture Overview

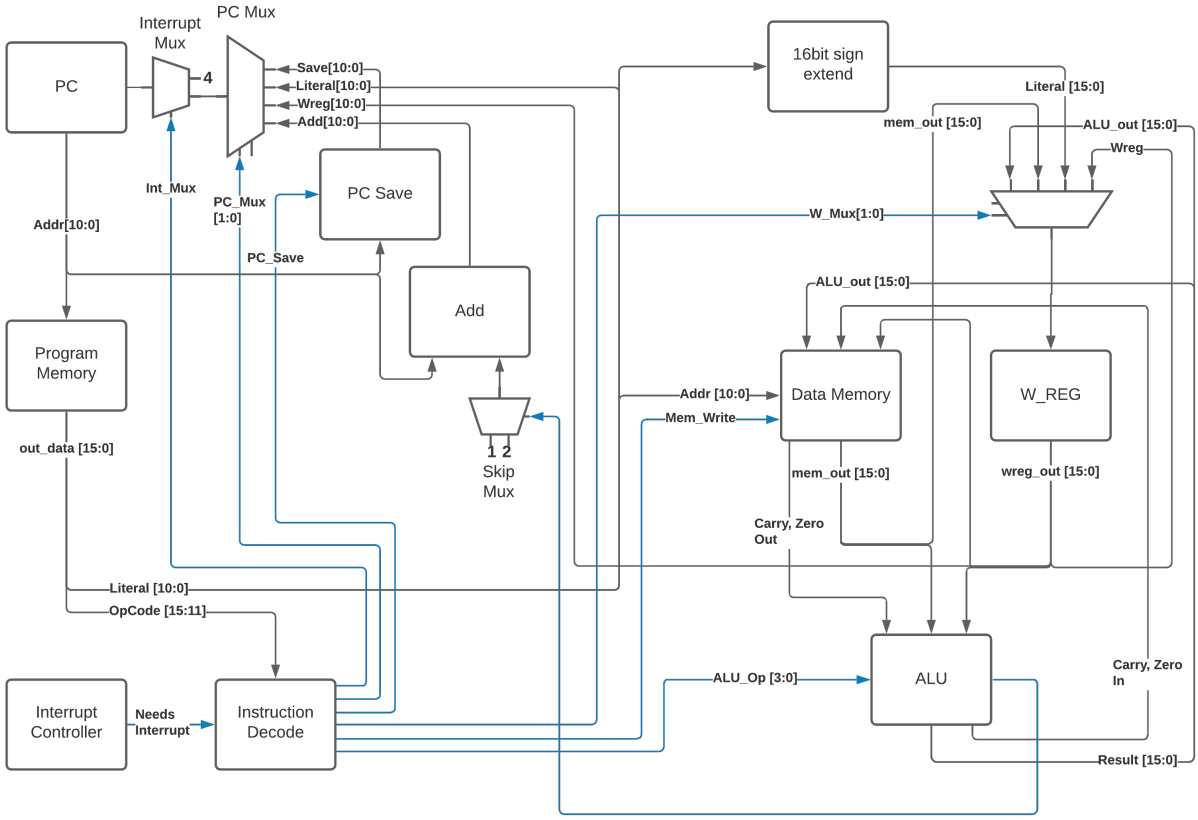

Fig. 2. Architecture Block Diagram. Note that black wires are data, blue wires are controls.

As previously mentioned the only 16 bit register used in the ISA is the Working Register (W_REG) which is hardwired to the second operand position in the ALU. There are also 2 1 bit registers that save the Zero and Carry ALU outputs between operations (and can be read and written through their respective memory mappings, see the Registers section below for more detail). An 11 bit PC_Save register stores the program counter value during interrupts to allow the program to return to normal operation after exiting in the interrupt routine (rfi instruction). Since a criteria for this project was minimal cost (defined by the number of registers and busses used), the small registers can be summed up as being just shy of a full 16 bit register (13 out of 16 bits used), for a total of 2 16 bit registers.

The architecture uses one large 11 bit bus to pass the literal value to the Data Memory module (address), W register (value, via the sign extension block and Wreg input mux), and Program Counter (address, via PC_Mux). This bus technically starts as the 16 bit instruction, but the upper 5 bits immediately branch off into the Instruction Decoder. A smaller 16 bit bus is used to

pass the ALU result to the data memory and the W register. Two very small 16 bit busses interconnect the ALU and Wreg, with one 11 bit leg branching off the Wreg bus to drive the Program Counter (via PC_Mux). For cost accounting it would be reasonable to sum these as somewhere between 3 and 4 busses, given that the 11 bit busses have to travel the furthest and interconnect several modules, whereas the 16 bit busses only connect amongst the data memory, w register (sometimes through a mux), and ALU. It is worth noting that adding peripherals would need to connect to the data memory module via at least 1 additional 16 bit bus.

Instruction Decoder

The Instruction Decode module determines the control register outputs based on the Opcode portion of the instruction. Both the Instruction code and the Destination bit portions of the Opcode are used in this determination. The table below enumerates the Instruction Decode module's outputs. As mentioned in the Instruction Details section, some operations ignore the destination bit of the opcode. Values marked with 'x' indicate that the input value is ignored, the default value produced by the assembler is zero. For operations that do use the destination bit, a value of 0 indicates that the result be stored in the W register whereas a value of 1 indicates the result is to be stored in the data memory.

Instr code	Mnem-onic	Dest	W_Mux [1:0]	Mem_Write	PC_Mux [1:0]	PC_Save	Int_Mux	ALU_Op [3:0]
0	mm	0	MEM	0	ADD	0	0	ZeroTest
0	mm	1	WREG	1	ADD	0	0	ZeroTest
1	mwm	x	WREG	1	ADD	0	0	Nop
2	mlw	x	LIT	0	ADD	0	0	Nop
3	rlm	0	ALU	0	ADD	0	0	RotL
3	rlm	1	WREG	1	ADD	0	0	RotL
4	rrm	0	ALU	0	ADD	0	0	RotR
4	rrm	1	WREG	1	ADD	0	0	RotR
5	awm	0	ALU	0	ADD	0	0	And
5	awm	1	WREG	1	ADD	0	0	And
6	owm	0	ALU	0	ADD	0	0	Or
6	owm	1	WREG	1	ADD	0	0	Or
7	xwm	0	ALU	0	ADD	0	0	Xor
7	xwm	1	WREG	1	ADD	0	0	Xor
8	add	0	ALU	0	ADD	0	0	Add
8	add	1	WREG	1	ADD	0	0	Add
9	sub	0	ALU	0	ADD	0	0	Sub
9	sub	1	WREG	1	ADD	0	0	Sub
A	sms	x	WREG	0	ADD	0	0	PCZero
B	smc	x	WREG	0	ADD	0	0	PCZerobar
C	gol	x	WREG	0	LIT	0	0	Nop
D	gow	x	WREG	0	WREG	0	0	Nop
E	wfi	x	WREG	0	SAVE	1	0	Nop
F	rfi	x	WREG	0	SAVE	0	0	Nop

For readability and understandability, variables were used for the ALU, W_Mux, and PC_mux values. The enumeration for the ALU_Op values can be found in the ALU section below. The Enumerations for W_Mux and PC_Mux are as follows:

W_Mux	value		PC_Mux	value
ALU	0		ADD	0
MEM	1		WREG	1
LIT	2		LIT	2
WREG	3		SAVE	3

The ALU

ALU inputs:

1. operation control input (4 bits)
2. Carry status register (1 bit)
3. Zero status register (1 bit)
4. Memory output (signed 16 bit)
5. W Register output (signed 16 bit)

ALU outputs:

1. Program Counter control signal (Skip_Mux, 1 bit)
2. Carry Status Register (1 bit)
3. Zero status register (1 bit)
4. Operation result (signed 16 bit)

The carry and zero bits are status registers. These status bits can be used by both the ALU and by users (they are mapped in data memory) to make decisions about the state of arithmatic. For example, if performing 32bit addition in software, the carry bit will be monitored by the program to determine when the lower byte has overflowed, necessitating an increment of the high bytes. The carry bit is also used as an inverted borrow bit for subtraction, allowing the program to determine that an operation underflowed in order to compare magnitude of the two values (<, >). Likewise, a set Zero bit after subtraction indicates equality of the subtracted values. See the Register Section for more information.

ALU Instructions

Status bits pass through unless listed in the affects Status box

Op Code	Operation	Description	Used By	Affects Status	PC Skip
0x0	RotL	Shift Mem 1 bit left, The bit in the carry position before the operation is shifted into the LSB. The MSB is shifted out, into the carry bit. W Unused.	rlm	Carry	0
0x1	RotR	Shift Mem 1 bit right, The bit in the carry position before the operation is shifted into the MSB. The LSB is shifted out, into the carry bit. W Unused.	rrm	Carry	0
0x2	Add	Adds W to Mem, Carry value is value of 17th bit of result (stripped to 16 bit output), Zero set if result is 0.	add	Carry Zero	0
0x3	Sub	Subtracts W from Mem (Mem - W), Carry cleared if result is negative, Zero set if result is 0.	sub	Carry Zero	0
0x4	And	Bitwise AND W and Mem, zero set if result is 0	awm	Zero	0
0x5	Or	Bitwise inclusive OR W and Mem, zero set if result is 0	owm	Zero	0
0x6	Xor	Bitwise exclusive OR W and Mem, zero set if result is 0	xwm	Zero	0
0x7	ZeroTest	Passes Mem to result, Zero set if Mem is 0	mm	Zero	
0x8	PCZero	Sets PC_Skip if Mem is nonzero, else clear	sms		?
0x9	PCZerobar	Sets PC_Skip if Mem is zero, else clear	smc		?
0xA-F	Nop	Passes W to result, No other operation is performed	mwm mlw gol gow wfi rfi		0

Data Memory Unit

The data memory unit interfaces with the on-chip SRAM memory. This implementation is equipped with 512 16 bit words, totaling 1KByte of memory. The memory is word addressable only. For example memory addresses 0x000 and 0x001 contain two different 16bit words, as opposed to two bytes comprising a 16 bit word.

Indirect Memory Access

The data memory unit includes the Indirect Memory Access peripheral. This peripheral allows programmatic access to data memory, as opposed to compile-time only literals. In other words, array offsets can be computed at run-time, for example:

```
// array_var[i] = 32;
mlw array_var        // load address of array_var
mwf inda             // store address of array_var in in IMA pointer
mm i,w               // load value of i
add inda,m           // index i words into the array
mlw .32              // load value of 32
mwm indv             // store 32 at array_var[i]
```

Registers

Registers are memory mapped to 16 bit values and are word addressable (only) for user/program access through the data memory unit's interface, starting from address 0x200. The first 5 words (addresses) are reserved for core registers and the Indirect Memory Access core peripheral registers. The remainder of the address space would hold peripheral control registers, if implemented.

Wreg

- Working Register (or W register)
- 16bit register
- Memory mapped to 0x200
- When accessed through the memory, this register is read-only (writes are ignored).
- This register is used as a data input to ALU and is usually the second operand in arithmetic operations (see ALU and Instruction sections for more detail). Most operations can optionally store the result in the Wreg instead of in memory.

Carry

- 1bit register
- Memory mapped to 0x201
- When accessed through the memory, the least significant bit is mapped to the register.
  - Upper 15 bits are read as 0
  - Writes to the upper 15 bits are ignored
- Used in and set by some ALU operations. For example:
  - addition carry (set high on addition overflow, set low otherwise).
  - subtraction borrow (inverted, set low on borrow)
  - rotate input/output
- Note : when subtracting unsigned or positive signed values, a clear Carry (borrow occurred) indicates that the value in W was greater than the value in Memory. If both signs are negative, then this logic is inverted. When dealing with mixed signs, the meaning of carry is determined by the position of the signed value. If working with

signed numbers and no "compile-time" knowledge of the value's sign is available, then the program will have to determine the signed-ness of the operands. Fortunately, in the case of mixed signs, the sign bit will determine which operand is greater.

Zero

- 1bit register
- Memory mapped to 0x202
- When accessed through the memory, the least significant bit is mapped to the register.
  - Upper 15 bits are read as 0
  - Writes to the upper 15 bits are ignored
- Set by some ALU operations. For operations that affect the zero bit:
  - set to 1 when the result is 0
  - set to 0 when the result is nonzero

Indv

- Indirect Memory Access Peripheral, Value Register
- 16bit register
- Memory mapped to 0x203
- Accessible only through memory interface. Full read and write support.
- Holds value of memory location pointed to by Inda

Inda

- Indirect Memory Access Peripheral, Address Register.
- 9 bit register
- Memory mapped to 0x204
- Accessible only through memory interface.
  - Upper 7 bits are read as 0.
  - Writes to upper 7 bits are ignored.
- The value stored in Inda is the memory address pointer for Indv

Program Memory Unit

The program memory is user accessible only during programming. The ISA contains no method to modify program memory values, although a peripheral could be implemented for that purpose. The verilog simulation loads the program memory from the the program.mem file included with this document. The program memory is word addressable and contains 512 16-bit words (1KByte). The program memory is addressed by the program counter which can be controlled in various ways through the instruction set.

Programming

For an example program see the program.c, program.asm, and program.mem files included with

this document. The text of program.c and program.asm have been included as an appendix to this document.

Calling Convention

There is no enforced calling convention.

For writing assembly, If the function is called from more than one place it is recommended to use the W register to pass the return address (PC + 2) (callee saved if the W register is needed). However, it is just as valid to implement a call stack and use W to pass the first parameter. If memory use allows, further parameters can be passed using fixed memory locations either shared amongst all functions or per-function. If the function is only called from one place then gol can be used to return and the W register can be used to pass the first argument and the return value.

For C compilers, it is recommended to setup a stack as part of the runtime starting from 0x1FF, moving up (numerically down). Use this stack to pass the return address and function parameters. The caller handles loading and cleaning the stack before and after calls. The order of arguments will depend upon the compiler, but the calling convention used in the samples provided is push the return address followed by the arguments from right to left, and then the return value.

Conclusion

This document and included files form a working low cost Instruction Set Architecture. The design successfully “runs” the included program that covers common C language constructs in a simulator. The simulated hardware and program have been painstakingly checked for accuracy of input and output at each sub-step of each instruction.

Special Thanks

A special thanks to Dr. Gaede for lending me a stack of Verilog and Digital Design books and pointing me in the right direction. This project would not have been possible without this help.

Appendix A: Tools used

- The verilog files were “compiled” using Icarus Verilog, a popular free open source software project.
- Waveforms were created from the simulation’s output VCD files using GTKWave
- GNU Make was used to script the build operations. This allowed quickly switching between building the top-level verilog file and the unit-test testbench verilog files. The file named “Makefile” contains the build instructions used by Make.
- The Architecture Block Diagram model was built using Lucid Charts, a web-based flowcharting tool.

Appendix B: Example Program

An example program is included with the project deliverables. The c code is contained in program.c, program.asm shows the c code hand compiled into assembly, and program.mem contains the hand assembled machine code. For completeness the text of the .c and .asm files are included below.

program.c

```
int add(int *by_ref, int by_val) {
  *by_ref = *by_ref + by_val;
  return by_val;
}

int i = 0;
int j = 20;
int k = 0;
int l[10];

void main(void) {
  for (i = 0; i < 10; i++) {
    l[i] = add(&j, i);
  }
  i = i - j;
  if (i >= 10) {
    j = 0xaa;
  } else {
    j = 0x55;
  }
  k = 0x55a9;
  while ((j & k) != 0) {
    k++;
```

```
  }
  i = (j | k) == -1;
  i = j > k;
  i = j <= k;
}
```

program.asm

```
STACKPTR equ 0x1FF

WREG equ 0x200  // W Register
CARRY equ 0x201  // Carry Register
ZERO equ 0x202  // Zero Register
INDV equ 0x203  // Indirect Value Register
INDA equ 0x204  // Indirect Pointer Register

W equ 0
w equ 0
M equ 1
m equ 1

ORG 0x0000
RESET: gol STARTUP

ORG 0x0004
INTERRUPT_VECTOR: rfi  // Interrupts not used

ORG 0x0005
STARTUP:
// initialize stack
      mlw (STACKPTR - 1)  // 0x1FE is top of stack
      mwm STACKPTR
//int i = 0;
      i equ 0x0000
      mlw .0
      mwm i

//int j = 20;
      j equ 0x0001
      mlw .20
      mwm j

//int k = 0;
      k equ 0x0002
      mlw .0
      mwm k

//int l[10];
      l equ 0x0003
      l_end equ (l + 10 - 1) // l_end = 0x000C
      gol MAIN
```

```
//void main(void) {
ORG 0x0010
MAIN:
//  for (i = 0; i < 10; i++) {
//    l[i] = add(&j, i);
//  }
      MAIN_TEMP_0 equ 0x000D
      MAIN_TEMP_1 equ 0x000E
      mm STACKPTR,w // get top of stack
      mwm MAIN_TEMP_0 // save original stack position
      mwm INDA  // point the Indirect access at the stack
      mlw MAIN_ADD_RETURN // return address
      mwm INDV
      mlw .1
      sub INDA,M // next stack address
      // load args right to left
      mm i,w
      mwm INDV
      mlw .1
      sub INDA,M // next stack address
      mlw j // &j
      mwm INDV
      mm INDA,w
      mwm STACKPTR
      gol ADD

MAIN_ADD_RETURN:
      mm STACKPTR,w
      mwm INDA
      mm INDV,w  // load return value
      mwm MAIN_TEMP_1  // save return value
      mm MAIN_TEMP_0,w // load previous STACKPTR value
      mwm STACKPTR  // "pop" function args off stack
      mlw l
      mwm INDA
      mm i,w
      add INDA,m
      mm MAIN_TEMP_1,w
      mwm INDV

      // i < 10 ?
      mlw .1
      add i,m
      mlw .10
      sub i,w
      sms CARRY  // skip if carry set
      gol MAIN  // i < 10

      // out of for loop
//  i = i - j;
      mm j,w
```

```
      sub i,m

//  if (i >= 10) {
//    j = 0xaa;
//  } else {
//    j = 0x55;
//  }
      mlw .10
      mwm MAIN_TEMP_1
      rlm i,w
      smc CARRY // If carry set, number is negative
      // borrow check doesn't work with mixed signs. Since
      // 10 is a positive literal the "compiler" knows that if i is
      // negative,
      // i < 10
      gol MAIN_I_SIGN_NEG

      mm i,w
      sub MAIN_TEMP_1,w
      mlw 0x55  // assume not
      sms CARRY
      mlw 0xaa  // i > 10
      smc ZERO
      mlw 0xaa // i == 10
      gol MAIN_I_SIGN_END

// If i is negative, it's obviously less than 10
MAIN_I_SIGN_NEG:
      mlw 0x55

MAIN_I_SIGN_END:
      mwm j

//  k = 0x55a9;
      mlw (0x55a9 >> 5) // staying under sign ext.
      add k,m  // could use mwm, but we know the memory is zeroed, and
               // this assures that carry is cleared in one op
      rlm k,m // 1
      rlm k,m // 2
      rlm k,m // 3
      rlm k,m // 4
      rlm k,m // 5
      mlw 0x9
      add k,m

//  while ((j & k) != 0) {
//    k++;
//  }
      mm k,w
      awm j,w
      smc ZERO
      gol MAIN_WHILE_LOOP_END // (j & k) == 0
```

```
MAIN_WHILE_LOOP:
      mlw .1
      add k,m
      mm k,w
      awm j,w
      sms ZERO
      gol MAIN_WHILE_LOOP

MAIN_WHILE_LOOP_END:
//  i = (j | k) == -1;
      mm k,w
      owm j,w
      mwm MAIN_TEMP_1
      mlw .-1
      sub MAIN_TEMP_1,w
      mm ZERO,w  // if temp == -1, zero = 1, else 0
      mwm i;

//  i = j > k;
      mm j,w
      sub k,w  // CARRY clear if Wreg > Mem
      mlw 1  // assume true
      smc CARRY
      mlw 0
      mwm i

//  i = j <= k;
      mm j,w
      sub k,w  // CARRY set if Wreg <= Mem
      mm CARRY,w
      mwm i

END_OF_PROGRAM:
      wfi // effectively a halt if there's no interrupt
            // and/or the interrupt handler just does rfi

ADD:
// return addr = *(STACKPTR + 2)
// by_ref = *(STACKPTR)
// by_val = *(STACKPTR + 1)
// return value = *(STACKPTR - 1)

//    int add(int *by_ref, int by_val) {
      ADD_TEMP_0 equ 0x000F

      mm STACKPTR,w
      mwm INDA
      mlw .1
      add INDA,m  // by_val
      mm INDV,w
      mwm ADD_TEMP_0 // by_val
      mm STACKPTR, w
```

```
      mwm INDA,m  // by_ref
      mm INDV,w  // by_ref pointer value loaded in W
      mwm INDA  // by_ref pointer value loaded into indirect
      mm ADD_TEMP_0,w  // by_val
//      *by_ref = *by_ref + by_val;
      add INDV,m

//      return by_val;
//    }
      mm STACKPTR,w
      mwm INDA,m
      mlw .1
      sub INDA,m  // return value
      mm ADD_TEMP_0,w
      mwm INDV
      mm INDA,w
      mwm STACKPTR // update STACKPTR
      mlw .3
      add INDA,m  // return adress
      mm INDV,w
      gow  // return
```