Segmented Address eXtension - Petranovic Architecture (SAX-PA)

By: Toby Everett

I. Introduction

A. Overview of the Processor

The SAX-PA processor is an extension to the Petranovic Architecture (PA). It is a 4-bit RISC CPU with a segmented Harvard memory model. The CPU is of load-store design. The CPU has 16 4-bit wide general purpose registers. Instruction memory has 8 address lines (256 words) and is 16-bits wide. Data memory has 8 address lines (256 words) and is 4-bits wide. The instruction set supports 15 basic operations including arithmetic, logical, comparison, and conditional and unconditional jumps. All instructions on the CPU take 4 clock cycles to execute.

B. Overview of the Instruction Set

The SAX-PA has 15 instructions. These are broken down into several groups:

1. Conditional Branch: beq
2. Unconditional Branch: jump, jfar
3. Memory Operations: sw, lw, swfar, lwfar
4. Arithmetic: add, addi, sub, slt
5. Logic: and, andi, or, ori

Notice that the unconditional branch and memory operation instructions have a “normal” version and a “far” version. Due to the design of the segmented memory model, the chip can be operated as a PA chip simply by avoiding the jfar, swfar, and lwfar instructions.

C. PA Compatibility

The SAX-PA CPU was designed to supercede the PA CPU as cleanly as possible while retaining full compatibility with the PA CPU. There are differences, but they have been minimized. Specifically, persons wishing to operate the SAX-PA CPU in PA mode should note that the operand order to the Arithmetic and Logic instructions has been changed and that register 0 is now defined to be 0 (similar to the MIPS architecture). However, it is my opinion that the added powers of the SAX-PA CPU will be utilized by anyone wishing to write real code on the chip, as they make life much easier.

D. Design Philosophy

The Segmented Address eXtensions to the Petranovic Architecture were implemented in order to address one of the PA’s primary failings: lack of address space. The PA’s 4 address lines allowed only 16 words of instruction

memory and data memory. This was far too limiting, but was understandable given the limitations of a 4-bit processor. However, by segmenting the memory model and using special on-chip registers to store the segment ID, this limitation has been overcome. The SAX-PA CPU was designed to be as programmer friendly as possible given the limitations of the technology and the restrictions of the PA heritage. The immediate instructions (addi, andi, ori) were added to enable code to be written without storing constants in data memory, simplifying the programmers job. Register 0 was hard-wired to 0 to allow the addi instruction to be used as a set instruction (as well as being useful in general). A quick survey of the available code for the SAX-PA CPU shows that in real programs, addi is the most used instruction and register 0 is the most used register, confirming the choices made in the design phase.

II. A Programmer's Guide to the SAX-PA CPU

A. Register Design

The SAX-PA CPU has 16 4-bit wide general purpose registers (GPR). Register 0 has been hard-wired to always return 0 and to act as a /dev/null (writes to register 0 execute as per normal, but register 0 still returns 0). Apart from this, the only restrictions on register use are those in "Suggested Conventions" on page 3.

B. Memory Model

The SAX-PA has a segmented Harvard memory architecture. The current segment ID's are stored in on chip registers. The only way to change the value of these registers is through the jfar, lwfar, and swfar instructions. All subsequent memory actions using the standard PA instructions will act on memory within the segment set by the last jfar, lwfar, or swfar instruction. Note that there are separate segment registers for instruction memory (Iseg) and data memory (Dseg). Also, the incrementation of the PC register from F to 0 will not increment Iseg. As a result, most code will contain a jfar in address F of the instructions segments. Also, user code to date has made almost no use of the lw and sw instructions, preferring to use the lwfar and swfar. As the execution time of all instructions is the same, it is suggested that swfar and lwfar be used to the exclusion of the lw and sw instructions.

C. The Instruction Set

Mnemonic	OP Code	OP 1	OP 2	OP 3	Behavior
beq	1	Rs	Rt	Offset	if ($Rs=$Rt) then $PC=$PC+Offset+1
jump	2	Rs	N/A	Offset	$PC=$Rs+Offset
jfar	3	Rs	Rt	Offset	$Iseg=$Rt; $PC=$Rs+Offset
sw	4	Rs	Rd	Offset	Mem[$Rs+Offset]=$Rd
lw	5	Rs	Rd	Offset	$Rd=Mem[$Rd+Offset]
swfar	6	Rs	Rd	Rt	$Dseg=$Rt; Mem[$Rs]=$Rd
lwfar	7	Rs	Rd	Rt	$Dseg=$Rt; $Rd=Mem[$Rs]
add	8	Rs	Rd	Rt	$Rd=$Rs + $Rt
addi	9	Rs	Rd	Value	$Rd=$Rs + Value
sub	A	Rs	Rd	Rt	$Rd=$Rs - $Rt
slt	B	Rs	Rd	Rt	$Rd=($Rs > $Rt)
and	C	Rs	Rd	Rt	$Rd=$Rs & $Rt
andi	D	Rs	Rd	Value	$Rd=$Rs & Value
or	E	Rs	Rd	Rt	$Rd=$Rs # $Rt
ori	F	Rs	Rd	Value	$Rd=$Rs # Value

D. Suggested Conventions

In the interest of promoting the development of a usable procedure library, the designer of the SAX-PA CPU would like to propose some conventions on register usage, procedure calls, and the like.

1. Return Addresses

   As every procedure call requires some way to return, it is suggested that the return instruction segment ID be stored in $E and the return instruction address be stored in $F. To return, the instruction jfar $F, $E, 0 can be executed.

2. The Stack

   It is suggested that a simple stack be maintained in data segment 0 for the purposes of storing return addresses before executing a procedure call. This

stack should not be used for passing values to procedures as this will fill up the stack to quickly. Values should be passed in some other segment and the segment address should be passed in a register. The position of the head of the stack should be stored in $D (and as such, $D should only be modified when pushing or popping values onto or off of the stack). The start of the stack is considered to be 0,0.

3. Other Considerations

   As of yet, there is no convention on callee/caller save registers. All procedures should be published along with a list of registers that will/may be modified in the body of that procedure.

E. A Sample Program

This program demonstrates how the addition of the "immediate instructions" (addi, andi, ori) along with the new segmented, Harvard memory architecture (which provides for 256 4-bit words of data memory and 256 16-bit words of instruction memory) enables the processor to tackle problems that were previously impossible, namely addition of two 64-bit numbers.

This is the setup code for the procedure call.

```
00 addi     $0, $A, 1   Sets A to 1
01 addi     $0, $B, 2   Sets B to 2
02 addi     $0, $C, 3   Sets C to 3
03 addi     $0, $D, 0   Sets D to 0
04 addi     $0, $E, 0   Sets E to 0
05 addi     $0, $F, 8   Sets F to 8
06 addi     $0, $1, 2   Puts 2 in 1
07 jfar     $0, $1, 0   Jumps to 20
08 jump     $0, $0, 8   Infinite loop
```

This is the main procedure. It presumes that the data segment ID's for A, B, and C are stored in $A, $B, and $C. It also presumes a standard stack and return address in $D, $E and $F.

```
20 addi     $D, $D, 1   Increments the stack head
21 swfar    $D, $E, 0   Pushes E onto the stack
22 addi     $D, $D, 1   Increments the stack again
23 swfar    $D, $F, 0   Pushes F onto the stack
24 addi     $0, $E, 2   Sets return segment to 2
25 addi     $0, $F, C   Sets return address to C
26 addi     $0, $9, 0   Sets position to 0 (first dec will
                        go to F)
27 addi     $0, $8, 0   Sets carry to 0
28 addi     $0, $1, 5   Sets 1 to 5
29 addi     $9, $9, F   Decrements position
2A addi     $0, $2, 4   Sets 2 to 4
2B jfar     $0, $2, 0   Calls the 4-bit slice adder
2C beq      $0, $9, 1   Skips the jump if we are at 0
2D jump     $0, $0, 9   Jumps to 29
2E addi     $0, $2, 3   Puts 3 in 2
```

```
2F jfar     $0, $2, 0   Jumps to 30
30 lwfar    $D, $F, 0   Pops F off the stack
31 addi     $D, $D, F   Decrements the stack head
32 lwfar    $D, $E, 0   Pops E off the stack
33 addi     $D, $D, F   Decrements the stack again
34 jfar     $F, $E, 0   Returns
```

This routine adds two 4 bit slices and takes care of Carry type stuff. This routine presumes the following:

Register $1 has a 5 in it
Register $8 has the carry-in from the previous step in it
Register $9 has out current position in the mess in it
Register $A has the data segment for A in it
Register $B has the data segment for B in it
Register $C has the data segment for C in it
Register $E has the return instruction segment in it
Register $F has the return instruction $PC in it

```
40 lwfar    $9, $2, $A  Gets current 4-bit slice of A
41 lwfar    $9, $3, $B  Gets current 4-bit slice of B
42 andi     $2, $4, 7   Puts 3 lsb's of A in 4
43 andi     $3, $5, 7   Puts 3 lsb's of B in 5
44 add      $4, $4, $5  Puts low bits of A+B in 4
45 add      $4, $4, $8  Adds the carry in
46 slt      $0, $5, $4  Grabs the high bit on lsb A+B+ carry
                        into 5
47 andi     $4, $4, 7   Puts the lsb's of A+B+carry into 4
48 slt      $0, $2, $2  Is A<0 (i.e. is the 4th bit of A
                        high?)
49 slt      $0, $3, $3  Is B<0 (i.e. is the 4th bit of B
                        high?)
4A add      $2, $2, $3  Adds high bits of A and B
4B add      $2, $2, $5  Adds in high bit of the 3 bit add
4C andi     $2, $3, 2   Grabs 2nd bit of 2
4D andi     $2, $2, 1   Grabs 1st bit of 2
4E slt      $3, $8, $0  Is the 2nd bit > 0 (put that in
                        Carry)
4F jfar     $0, $1, 0   Jumps to next segment
50 beq      $0, $2, 1   If the 1st bit of 2 is zero, skip
                        the next step
51 ori      $4, $4, 8   Add a msb to A+B+Carry
52 swfar    $9, $4, $C  Put the mess into the current 4-bit
                        slice of C
53 jfar     $F, $E, 0   Return, we are finished with this
                        subroutine.
```

A quick analysis of the code raises some interesting issues. Register 0 which is hardwired to return 0, gets a heavy work out. It gets called 28 times, out of 113 register calls (Almost exactly 25% of the time, and remember that $0 is

almost never written to).

In addition, some of the most used instructions were the ones that were added along with the segmented memory model:

Mnemonic	Uses		Mnemonic	Uses
add	4		beq	2
addi	19		jump	2
sub	0		jfar	6
slt	4		sw	0
and	0		lw	0
andi	5		swfar	3
or	0		lwfar	4
ori	1			

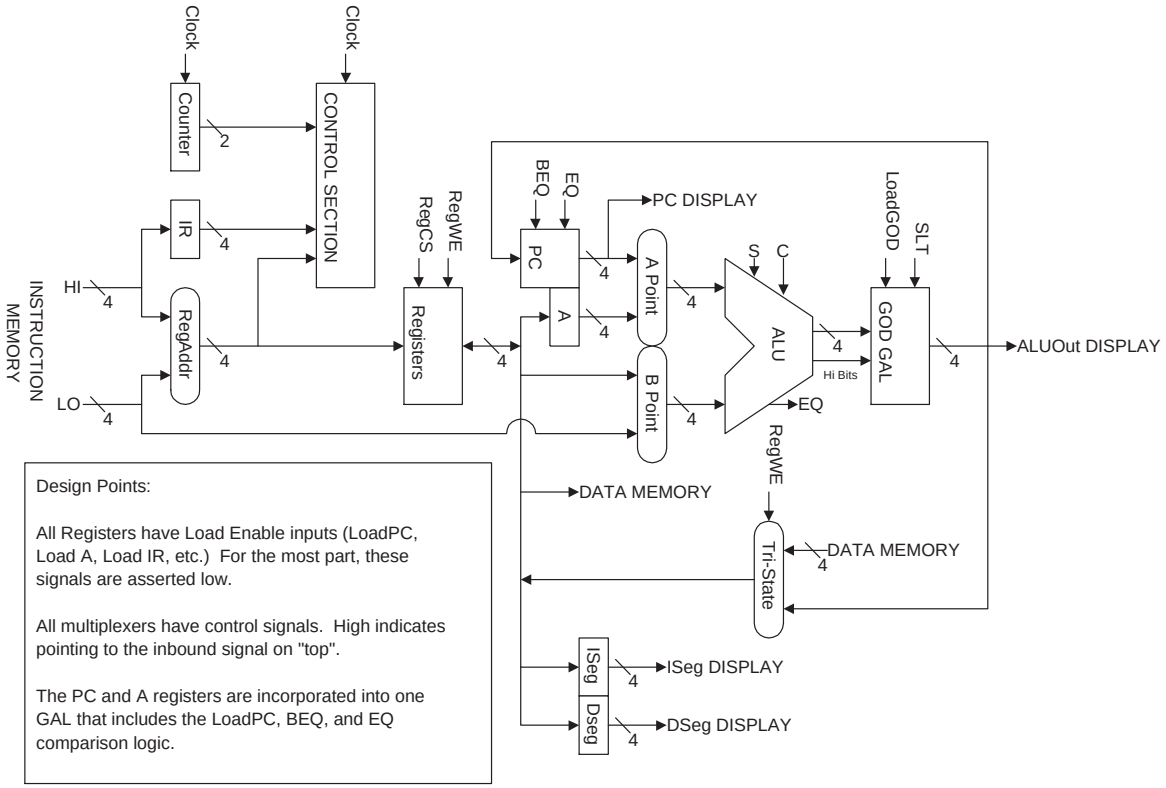
Notice how useful addi is. Out of 50 instructions, it is used 19 times. Also, notice that swfar and lwfar are used exclusively in place of sw and lw. There are places in the code where sw and lw could be substituted (the second operation in a stack store or restore), but as they take the same number of clock cycles, I just used the far versions for simplicity and ease of reading. Nonetheless, sw and lw should be maintained as they offer base addressing within a segment, which could be very handy when working with small datasets. jfar gets a pretty good workout (it is used within procedures to move from one segment to the next, and is also instrumental in calling procedures and returning from them). Another interesting point is that the immediate versions of the logic operations are the only ones used in this code. Nonetheless, removing the non-immediate versions would be a mistake, as when they are needed, replicating them in terms of other operations would be rather painful.

III. Implementation of the SAX-PA CPU

A. Overview

The implementation of the SAX-PA CPU was done using GAL's and other commonly available discrete logic chips. In order to test the implementation, a human-interface was designed that handles instruction memory and data memory as well as providing a way to view the contents of the registers.

B. The Design



C. The Control Signals:

Signal	Description
LoadAIR	Goes low when $A and $IR should be loaded.
LoadGOD	Goes low when GOD GAL should be loaded.
RegAddr	Controls which quarter-word the registers get their address from.
LoadPC	Goes low when $PC should be loaded.
LoadISeg	Goes low when $ISeg should be loaded.
LoadDSeg	Goes low when $Dseg should be loaded.
A Point	Controls where the A input of the ALU gets it's input from.
B Point	Controls where the B input of the ALU gets it's input from.
S3,S2,S1,S0	Control the ALU. (M on the ALU is pinned high).

Signal	Description
C	Carry-in to the ALU.
SLT	High when GOD GAL should do a Set Less Than.
RegWE	High when the registers should be written to. This signal is the only one which is dependant on the state of the clock (only low when the clock is). Also, it is passed through a rising edge triggered low-pulse generator before it is used.
RegCS	High when the Register chip should be disabled. High whenever all the address lines on the Register chip are low.
RegInput	Controls where the registers get their data from during the write cycle. Also used to signify data memory reads.
BEQ	High when $PC should load if EQ is also high.
DataWrt	Not used anywhere in the CPU. Used to signify data memory writes.

For more information on the control signals and their relationship with the IR and the Counter, see Attachment I.

D. The GAL's

Eight GAL's were used in the design, programmed from 6 different ABL files.

1. STROBER

   This chip is used in the human memory interface. It is used to strobe the key-pad column wires. It also takes the 4 row wires as inputs and uses those to detect key-presses, pin the strobe, and outputs a debounced key-press signal.

2. DCODBANK

   This chip decodes the output from the keypad and stores both the current key-press and the last key-press. It uses the debounced key-press signal from the STROBER chip as a clock.

3. MULTIHCD

   This is a one chip implementation of a multiplexed Hex 7-Segment LED driver. It takes a clock and two 4-wire buses as inputs and outputs the inverted clock and driver signals for the 7 segments. The driver signals do not have resistors on them, so each of them must pass through a resistor before hitting the 7-segment display. This chip was used once for each of the 3 Dual 7-Segment LED displays.

4. COUNTER

   This chip implements a 2 bit counter as well as outputting a host of control signals for the CPU. It also handles the RegCS signal.

5. PCNA

   This chip handles the PC and A registers. It deals with the rather complex handling of LoadPC, BEQ, and EQ to decide whether or not to load the PC register.

6. GODGAL

   This chip handles the ALUOut register as well as SLT decision making. To assist in the SLT decision making, the high bits of A and B are passed to it along with the output from the ALU and the control signals LoadGOD and SLT.

E. Other Implementation Details

The gory details can be had by examining the implementation. A few

1. Register 0 Implementation

   Register 0 was implemented by using the CS pin on the chip. By raising the CS pin high, the chip enters a high impedance move. Pull-down resistors were used to generate the appropriate low signals for a 0. The RegCS pin was generated by the COUNTER GAL by inverting the result of OR'ing all the address lines on the Register chip.

2. ALU Details

   The ALU output pin EQ was found to be either low or in high impedance mode. A pull-up resistor was used to generate a high or low signal from this. Also, by appropriate use of the carry input, it was found that all the logic functions required could be implemented in the arithmetic mode, so the mode pin was hooked high.

3. Data Memory Implementation

   Data memory output is implemented using a bank of 4 LED's and data memory input is implemented using 4 DIP switches.

4. Register Viewing

   When the CPU is in the first half of cycle 0, the Data memory output LED's can be used to view the contents of any register simply by hitting the appropriate key on the keypad. This does not affect the state of the CPU.

F. Test Program

Last, but not least, a test program is provided that utilize all of the instructions in the CPU. Notice that, in addition to the memory location, the 2-byte equivalent for the instruction is included.

```
00 90 43 addi     $0, $4, 3   Put 3 in register 4
01 90 11 addi     $0, $1, 1   Put 1 in register 1
02 81 21 add      $1, $2, $1  Put 2 in register 2
03 84 51 add      $4, $5, $1  Put 4 in register 5
04 A5 51 sub      $5, $5, $1  Subtract 1 from register 5
05 B4 65 slt      $4, $6, $5  Set register 6 if register 5 <
                              register 4
06 16 0D beq      $6, $0, -3  Branch back to 04 if register
                              6 = 0
07 41 22 sw       $1, $2, 2   Put 2 in memory location 0,3
08 51 32 lw       $1, $3, 2   Put 2 from memory location 0,3
                              in register 3
09 61 32 swfar    $1, $3, $2  Put 2 in memory location 2,1
0A 51 40 lw       $1, $4, 0   Put 2 from memory location 2,1
                              in register 4
0B 71 40 lwfar    $1, $4, $0  Put 7 from memory location 0,1
                              in register 4
0C D4 5D andi     $4, $5, 13  Put 5 in register 5
0D E3 65 or       $3, $6, $5  Or 2 with 5(7), put in
                              register 6
0E C6 75 and      $6, $7, $5  And 5 with 7(5), put in
                              register 7
0F 30 10 jfar     $0, $1, 0   Jump to memory location 1,0
10 F5 88 ori      $5, $8, 8   Or 5 with 8(13), put in
                              register 8
11 90 F5 addi     $0, $F, 5   Put 5 in register F
12 90 E1 addi     $0, $E, 1   Put 1 in register E
13 90 94 addi     $0, $9, 4   Put 4 in register 9
14 30 92 jfar     $0, $9, 2   Jump to 42 (subroutine call)
15 20 05 jump     $0, $0, 5   Jump to 15

42 20 05 jump     $0, $0, 5   Jump to 45
45 3F E0 jfar     $F, $E, 0   Return to the calling
                              procedure
```