The PAW Architecture Reference Manual

by Hansen Zhang

For COS375/ELE375

Princeton University

Last Update: 20 September 2015

1. Introduction

The PAW architecture is a simple architecture designed to be easy to implement in a semester course using a minimum of hardware resources. It is derived from the Thumb ISA from Advanced Risc Machines, Ltd.

This manual roughly follows the structure of the *ARM Architecture Reference Manual* and is indebted to that document for many concepts and details of describing the architecture.

2. Programmer's Model

2.1. Data types

PAW processors support the following data types

Byte	8 bits
Halfword	16 bits

- All data operations are performed on halfword quantities; bytes are only relevant for loads and stores.
- Some instructions require "word" alignment. Word alignment is 32-bit alignment, but words are never actually operated on directly.
- PAW instructions are exactly one halfword in size and are aligned on a two-byte boundary.

When these types are described as *unsigned*, the N-bit data value represents an integer in the range 0 to $+2^{N}-1$, using normal binary format. When these types are described as *signed*, the N-bit data value represents an integer in the range -2^{N-1} to $+2^{N-1}-1$, using two's complement format.

The low-order bit of a data type is always considered to be bit 0.

The memory byte ordering is little-endian; byte 0 contains the low-order 8 bits of the halfword at address 0.

2.2. Registers

The PAW processor has a total of 17 registers:

- 16 general-purpose registers named R0-R15. These registers are 16 bits wide.

- 1 status register. This register is also 16 bits wide, but only 4 of the 16 bits need to be implemented.

2.2.1. General purpose registers

The general-purpose registers R0-R15 can be split into three groups.

Registers R0-R7 are the *low registers*. These registers are fully general-purpose and can be used wherever an instruction allows a general-purpose register.

Registers R8-R14 are the *high* registers. These register may only be used by a subset of the instructions. R13 is normally used as a stack pointer and is also known as the SP. R14 is normally used to store subroutine return addresses and is also known as the LP. Note that these uses of R13 and R14 are by convention only; the architecture requires no special use of them.

Register R15 holds the *Program Counter*, or PC. It can be used in place of one of the high registers R8-R14. There are also instructions that implicitly read or write the program counter. When the program counter is read, the value read is always the address of the instruction reading the PC. When the program counter is written, the next instruction executed after the writing instruction is the instruction at the new value of the PC. In other words, writing to the PC causes a branch to occur. Note that whenever a new value is written explicitly to the PC by an instruction, this value overrides/overwrites the "PC+2" value which is normally written at program counter update.

2.2.2. Status registers

The *Status Register* contains the condition code flags. The format of the status register is shown below:

15	3	2	1	0
Unimplemented	N	Z	C	V

The condition code flags are modified by comparison and some of the arithmetic and data transfer instructions. See the individual instruction descriptions to determine which instructions may modify a flag. Some instructions may modify only a subset of the flags.

2.2.2.1. The N flag

This flag is set to bit 15 of the result of the instruction. When the value is interpreted as a two's complement value, then N=1 if the result is negative and N=0 if it is positive or zero.

2.2.2.2. The Z flag

This flag is set to 1 if the result of the instruction is zero. It is set to 0 otherwise.

2.2.2.3. The C flag

This flag is set in one of three ways:

- For an addition instruction, C is set to 1 if the addition produced a carry (i.e. an unsigned overflow) and to 0 otherwise.
- For a comparison instruction, C is set to 1 if the subtraction implied by the comparison did **not** produce a borrow and 0 otherwise. This is equivalent to saying that if the second operand were negated and added to the first operand, there would be a carry out of the addition.

  NOTE: This behavior is the same as that of ARM, but is different from that of other architectures, such as SPARC.
- For a shift instruction, C is set to the last bit shifted out of the value.

2.2.2.4. The V flag

This flag is set only on addition and comparison instructions. It is set to 1 if signed overflow occurred and 0 otherwise.

2.3. Exceptions and reset

On reset, a PAW processor sets the PC to 0 and begins execution from that address.

No exceptions of any kind are detected by a PAW processor.

2.4. Memory and memory-mapped I/O

2.4.1. Byte addressing

The PAW architecture uses a single, flat address space of 2^{16} 8-bit bytes. Byte addresses are treated as unsigned numbers running from 0 to $+2^{16}-1$.

Address calculations are normally performed using ordinary integer instructions. This means that they normally *wrap around* if they overflow or underflow the address space. Essentially all address operations are taken modulo 2^{16}. Programs should not rely on this behavior.

2.4.2. Instruction addressing

Programs should not rely on the sequential execution of the instruction at address 0x0000 after the instruction at address 0xfffe.

2.4.3. Half-word addressing

The address space is also regarded as consisting of 2^{15} 16-bit words, each of whose address is *halfword-aligned*. Half-word-aligned means that the address is divisible by 2.

The results of an unaligned halfword-sized data access are unpredictable.

Halfwords are stored in little-endian byte order; Byte 0 contains the lower 8 bits of the halfword at address 0.

The load and store halfword instructions enforce an additional alignment restriction: the address must be 32-bit aligned (i.e. divisible by 4). As a result, it is not possible to use these instructions to load or store the halfwords at addresses 2, 6, 10, etc. Two byte loads or stores must be used instead

2.4.4. Memory-mapped I/O

The standard way to perform I/O functions on a PAW system is through memory-mapped I/O. This uses special memory addresses which supply I/O functions when they are loaded from or stored to.

Instruction fetches must not occur from memory-mapped I/O locations.

3. The PAW Instruction Set

3.1. Instruction set encoding

```
     15   14   13   12   11   10    9    8    7    6    5    4    3    2    1    0
 0 | 0  | 0  | 0  | OP=0-31            |            <unused>                | 
 3 | 0  | 0  | 1  | OP=0-3|     Rd      |            Immediate8              | *
 4 | 0  | 1  | 0  | 0  | 0  | 0  |    OP=0-15     |     Rs     |     Rd     | *
 5 | 0  | 1  | 0  | 0  | 0  | 1  | OP=0-3| H1| H2|     Rs     |     Rd     | *
 9 | 0  | 1  | 1  | B  | L  |       Imm5          |     Rb     |     Rd     |
12 | 1  | 0  | 1  | 0  | SP|     Rd      |            Immediate8              |
16 | 1  | 1  | 0  | 1  |      Cond       |       8 Bit Branch Offset          |
```

* - These instructions affect the status register condition code flags.

– All immediates are unsigned except for the branch offset.

3.1.1. Format 0: Miscellaneous operations

OP=0 HALT. The <unused> field should be set to 0.

3.1.2. Format 3: immediate operations

OP=0 MOV Rd, #Immediate8
OP=1 CMP Rd, #Immediate8

3.1.3. Format 4: ALU operations (2 operand)

<OP> Rd, Rs

OP=	0	1	2	3	4	5
	AND	EOR	ASR	TST	NEG	CMP

3.1.4. Format 5: Hi register operations

OP=0 ADD Rd, Rs
OP=1 CMP Rd, Rs (H1 = H2 = 0 is undefined)
OP=2 MOV Rd, Rs

For each one, (H1=1 => Rd = 8..15, H2=1 => Rs = 8..15)

3.1.5. Format 9: Load/Store with immediate offset

STR Rd, [Rb, #Imm5<<2] (L=0, B=0 - 0..124 immediate offset)
LDR Rd, [Rb, #Imm5<<2] (L=1, B=0 - 0..124 immediate offset)

```
STRB    Rd, [Rb, #Imm5]       (L=0, B=1 - 0..31 immediate offset)
LDRB    Rd, [Rb, #Imm5]       (L=1, B=1 - 0..31 immediate offset)
```

3.1.6. Format 12: Load address

SP=0   ADD   Rd, PC, #Immediate8<<2 (0..1020 immediate offset)

3.1.7. Format 16: Conditional branch

- Conditions listed later
- Branch range -256..+254 (offset shifted left by 1)

```
B<CC>   Label
```

3.2. Instruction types

In this section the instructions are classified by type.

3.2.1. Data processing instructions

These instructions are used to perform arithmetic and logical operations on registers and to move data between registers. The instructions are:

- ADD add two operands
- AND - logically "AND" the operands
- ASR - arithmetic shift right of one operand by the value of the other
- CMP - compare two operands
- EOR - exclusive-or the operands
- MOV - move value from one register to another
- NEG - negate one operand
- TST - Perform a logical AND of the operands and set the condition codes

3.2.2. Load and store instructions

These instructions are used to move data from memory to registers and back. Data movement can occur in either byte-size or halfword-size chunks. The instructions are:

- LDR - load a halfword to a register
- STR - store a halfword from a register
- LDRB - load a byte to a register.
- STRB - store a byte to a register.

3.2.3. Branch instructions

These instructions are used to alter the control flow of the program. Branches occur based upon the values of the condition code flags in the Status Register. The branch condition names are:

condition code	*encoding*	*meaning*	*Calculation*
eq	0000	Equal	Z=1
ne	0001	Not equal	Z=0
cs	0010	Carry set/ unsigned higher or same	C=1
cc	0011	Carry clear/ unsigned lower	C=0
mi	0100	Minus/Negative	N=1
pl	0101	Plus/Positive or zero	N=0
vs	0110	Overflow	V=1
vc	0111	No overflow	V=0
hi	1000	Unsigned higher	C= 1 && Z=0
ls	1001	Unsigned lower or same	C = 0 || Z = 1
ge	1010	Signed >=	(N==V)
lt	1011	Signed <	(N != V)
gt	1100	Signed >	Z=0 && (N==V)
le	1101	Signed <=	Z=1 || (N!=V)

3.2.4. Miscellaneous instructions

There is one instruction in this class: the HALT instruction.

3.3. Instruction descriptions

Detailed descriptions of each instruction follow in alphabetical order. The instruction operation is described in a C-like pseudocode which uses indentation instead of braces to show statement nesting. The individual steps of the operation are defined as if they were to occur sequentially. In other words, when one step modifies Rd and a later step sets the N condition code to Rd[15], the **new** value of Rd is used to set the N condition code. Hardware, of course, may perform the steps in parallel as long as the sequential meaning of the steps is preserved

The notation [*hb*:*lb*] means bits *hb* to *lb*, inclusive of the register or intermediate result.

3.3.1. ADD(1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	H1	H2	Rs			Rd		

This form of ADD adds the value of one register to the value of a second register and stores the result in the first register. None, one, or both of the registers may be high registers. This instruction sets the condition code flags based upon the result.

Syntax

```
ADD  <Rd>, <Rs>
```

where:

- <Rd> specifies the register containing the first value and also the destination register. It can be any of R0 to R15. The register number is encoded in the instruction in H1 (most significant bit) and Rd (remaining 3 bits).
- <Rs> specifies the register containing the second value. It can be any of R0 to R15. The register number is encoded in the instruction in H2 (most significant bit) and Rs (remaining 3 bits).

Operation

```
Rd = Rd + Rs
N Flag = Rd[15]
Z Flag = if (Rd == 0) then 1 else 0
C Flag = if (carry out) then 1 else 0
V flag = if (signed overflow) then 1 else 0
```

3.3.2. ADD(2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd			Immediate8							

This form of ADD adds an unsigned immediate value to the PC and writes the resulting PC-relative address to a destination register. The immediate value can be any multiple of 4 in the range 0 to 1020. The condition flags are not affected.

Syntax

```
ADD  <Rd>, PC, #<Immediate8><<2
```

where:

- <Rd> specifies the destination register. It can be any of R0 to R7.
- PC indicates PC-relative addressing (note: R15 is also correct here)
- <Immediate8> Specifies an unsigned 8-bit immediate value that is shifted left by 2 and added to the value of the PC.

Operation

```
Rd = (PC AND 0xFFFC) + (Immediate8 << 2)
```

3.3.3. AND

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rs			Rd		

The AND (Logical AND) instruction performs a bitwise AND of the values in two registers and stores the result in the first. The condition flags are updated based upon the result.

Syntax

```
AND  <Rd>, <Rs>
```

where:

- <Rd> specifies the register containing the first value and also the destination register. It can be any of R0 to R7.
- <Rs> specifies the register containing the second value. It can be any of R0 to R7.

Operation

```
Rd = (Rd and Rs)
N Flag = Rd[15]
Z Flag = if (Rd == 0) then 1 else 0
C Flag = unaffected
V flag = unaffected
```

3.3.4. ASR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0	Rs			Rd		

The ASR (Arithmetic Shift Right) instruction is used to provide the signed value of a register divided by a variable power of 2. The condition flags are updated based upon the result.

Syntax

```
ASR  <Rd>, <Rs>
```

where:

- <Rd> specifies the register containing the value to be shifted and also the destination register. It can be any of R0 to R7.
- <Rs> specifies the register containing the value of the shift amount. It can be any of R0 to R7.

Operation

```
if Rs[7:0] == 0 then
     C flag = unaffected
     Rd = unaffected
else if Rs[7:0] < 16 then
     C flag = Rd[Rs[7:0] - 1]
     Rd = Rd >> Rs[7:0]
else
     C flag = Rd[15]
     if Rd[15] == 0 then
          Rd = 0
     else
          Rd = 0xffff
N Flag = Rd[15]
Z Flag = if (Rd == 0) then 1 else 0
V flag = unaffected
```

3.3.5. B

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Cond				BranchOffset							

The B (branch) instruction is used to provide a conditional branch to a target address.

Syntax

```
B<cond>   <target address>
```

where:

- <cond> is the condition under which the instruction is executed. See the table in the *Branch Instructions* section.
- <target_address> specifies the address to branch to. The branch target address is calculated by:
    1. Shifting the 8-bit signed offset field of the instruction left by one bit.
    2. Sign-extending the result to 16 bits.
    3. Adding this to the contents of the PC.

    The instruction can therefore specify an offset of -256 to +254 bytes.

Operation

```
if condition true then
     PC = PC + (sign extend (BranchOffset<<1))
else
     PC = PC + 2 (as with any non-branch instruction)
```

3.3.6. CMP(1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rd			Immediate8							

This form of the CMP (Compare) compares a register with an 8-bit unsigned immediate value. The condition flags are updated based upon the result of subtracting the immediate value from the register

Syntax

```
CMP  <Rd>, # <Immediate8>
```

where:

- <Rd> specifies the register for comparison. It can be any of R0 to R7.
- <Immediate8> specifies the unsigned 8-bit immediate value for comparison in the range 0 to 255.

Operation

```
alu_out = Rd - Immediate8
N Flag = alu_out[15]
Z Flag = if (alu_out == 0) then 1 else 0
C Flag = if (borrow from last bit) then 0 else 1
V flag = if (signed overflow) then 1 else 0
```

3.3.7. CMP(2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rs			Rd		

This form of the CMP (Compare) compares two register values. Both registers must be low registers. The condition flags are updated based upon the result of subtracting the second register from the first.

Syntax

```
CMP  <Rd>, <Rs>
```

where:

- <Rd> specifies the register containing the first value. It can be any of R0 to R7.
- <Rs> specifies the register containing the second value. It can be any of R0 to R7.

Operation

```
alu_out = Rd - Rs
N Flag = alu_out[15]
Z Flag = if (alu_out == 0) then 1 else 0
C Flag = if (borrow from last bit) then 0 else 1
V flag = if (signed overflow) then 1 else 0
```

3.3.8. CMP(3)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	H1	H2	Rs			Rd		

This form of the CMP (Compare) compares two register values. One or both of the register values is a high register. The condition flags are updated based upon the result of subtracting the second register from the first.

Syntax

```
CMP  <Rd>, <Rs>
```

where:

- <Rd> specifies the register containing the first value. It can be any of R0 to R15. The register number is encoded in the instruction in H1 (most significant bit) and Rd (remaining 3 bits).
- <Rs> specifies the register containing the second value. It can be any of R0 to R15. The register number is encoded in the instruction in H2 (most significant bit) and Rs (remaining 3 bits).

Operation

```
alu_out = Rd - Rs
N Flag = alu_out[15]
Z Flag = if (alu_out == 0) then 1 else 0
C Flag = if (borrow from last bit) then 0 else 1
V flag = if (signed overflow) then 1 else 0
```

3.3.9. EOR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	1	Rs			Rd		

The EOR (Logical Exclusive-Or) instruction performs a bitwise exclusive or of the values in two registers and stores the result in the first. The condition flags are updated based upon the result.

Syntax

```
EOR  <Rd>, <Rs>
```

where:

- <Rd> specifies the register containing the first value and also the destination register. It can be any of R0 to R7.
- <Rs> specifies the register containing the second value. It can be any of R0 to R7.

Operation

```
Rd = (Rd EOR Rs)
N Flag = Rd[15]
Z Flag = if (Rd == 0) then 1 else 0
C Flag = unaffected
V flag = unaffected
```

3.3.10. HALT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The HALT instruction causes the processor to cease fetching and executing instructions.

Syntax

```
HALT
```

Operation

```
Stop fetching and executing instructions.
```

3.3.11. LDR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	Imm5					Rb			Rd		

The LDR instruction allows 16-bits of memory data to be loaded into a general-purpose register. The condition code flags are not affected by this instruction.

The memory address must be divisible by 4. This implies that this instruction **cannot** be used to load halfwords from the two upper-order bytes of words. Two LDRB instructions must be used instead.

Syntax

```
LDR  <Rd>, [ <Rb>, # <Imm5> <<2 ]
```

where:

- <Rd> specifies the destination register. It can be any of R0 to R7.
- <Rb> specifies the register containing the base memory address. It can be any of R0 to R7.
- <Imm5> specifies a five-bit unsigned immediate value to be multiplied by 4 and added to the value of <Rb> to form the memory address. This yields an effective address range from the value of <Rb> to the value of <Rb> + 124.

Operation

```
address = Rb + (Imm5 << 2)
if (address[1:0] == 0b00)
     data = Memory[address+1] , Memory[address]
else
     data = UNPREDICTABLE
Rd = data
```

3.3.12. LDRB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	Imm5					Rb			Rd		

The LDR instruction allows 8-bits of memory data to be loaded into a general-purpose register. The byte is not sign-extended. The condition code flags are not affected by this instruction.

Syntax

```
LDRB <Rd>, [ <Rb>, # <Imm5>]
```

where:

- <Rd> specifies the destination register. It can be any of R0 to R7.
- <Rb> specifies the register containing the base memory address. It can be any of R0 to R7.
- <Imm5> specifies a five-bit unsigned immediate value in the range 0 to 31 to be added to the value of <Rb> to form the memory address.

Operation

```
address = Rb + (Imm5)
Rd = Memory[address]
```

3.3.13. MOV(1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			Immediate8							

This form of the MOV(Move) moves an 8-bit unsigned immediate value into a register. The condition flags are updated based upon the result.

Syntax

```
MOV  <Rd>, # <Immediate8>
```

where:

- <Rd> specifies the destination register. It can be any of R0 to R7.
- <Immediate8> specifies the unsigned 8-bit immediate value in the range 0 to 255.

Operation

```
Rd = Immediate8
N Flag = 0
Z Flag = if (Rd == 0) then 1 else 0
C Flag = unaffected
V flag = unaffected
```

3.3.14. MOV(2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	H1	H2	Rs			Rd		

This form of MOV IMove) moves the value of the second register into the first register. None, one, or both of the registers may be high registers. This instruction sets the condition code flags based upon the result.

Syntax

```
MOV  <Rd>, <Rs>
```

where:

- <Rd> specifies the register containing the first value and also the destination register. It can be any of R0 to R15. The register number is encoded in the instruction in H1 (most significant bit) and Rd (remaining 3 bits).
- <Rs> specifies the register containing the second value. It can be any of R0 to R15. The register number is encoded in the instruction in H2 (most significant bit) and Rs (remaining 3 bits).

Operation

```
Rd = Rs
N Flag = Rd[15]
Z Flag = if (Rd == 0) then 1 else 0
C Flag = unaffected
V flag = unaffected
```

3.3.15. NEG

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0	Rs			Rd		

The NEG (Negates) instruction negates the value of one register and stores the result in a second register. The condition flags are updated based upon the result.

Syntax

```
NEG  <Rd>, <Rs>
```

where:

- <Rd> specifies the destination register. It can be any of R0 to R7.
- <Rs> specifies the register containing the value. It can be any of R0 to R7.

Operation

```
Rd = 0 - Rs
N Flag = Rd[15]
Z Flag = if (Rd == 0) then 1 else 0
C Flag = if (borrowed from last bit in 0-Rs) then 0 else 1.
V flag = if (signed overflow in 0-Rs) then 1 else 0.
```

3.3.16. STR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	Imm5					Rb			Rd		

The STR instruction allows 16-bits of data to be stored to memory from a general-purpose register. The condition code flags are not affected by this instruction.

The memory address must be divisible by 4. This implies that this instruction **cannot** be used to store halfwords into the two upper-order bytes of words. Two STRB instructions must be used instead.

Syntax

```
STR  <Rd>, [ <Rb>, # <Imm5> <<2 ]
```

where:

- <Rd> specifies the register containing the data to store. It can be any of R0 to R7.
- <Rb> specifies the register containing the base memory address. It can be any of R0 to R7.
- <Imm5> specifies a five-bit unsigned immediate value to be multiplied by 4 and added to the value of <Rb> to form the memory address. This yields an effective address range from the value of <Rb> to the value of <Rb> + 124.

Operation

```
address = Rb + (Imm5 << 2)
if (address[1:0] == 0b00)
     Memory[address+1]= Rd[15:8]
     Memory[address] = Rd[7:0]
else
     Memory[address+1] = UNPREDICTABLE
     Memory[address] = UNPREDICTABLE
```

3.3.17. STRB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	Imm5					Rb			Rd		

The STR instruction allows 8-bits of data to be stored to memory from a general-purpose register. The byte is not sign-extended. The condition code flags are not affected by this instruction.

Syntax

```
STRB <Rd>, [ <Rb>, # <Imm5>]
```

where:

- <Rd> specifies the register containing the data to be stored. It can be any of R0 to R7.
- <Rb> specifies the register containing the base memory address. It can be any of R0 to R7.
- <Imm5> specifies a five-bit unsigned immediate value in the range 0 to 31 to be added to the value of <Rb> to form the memory address.

Operation

```
address = Rb + (Imm5)
Memory[address] = Rd[7:0]
```

3.3.18. TST

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Rs			Rd		

The TST (Test) instruction performs a bitwise AND of the values in two registers and updates the condition code flags based upon the result. It does not affect the value of either register. A very common use of TST is to determine whether a single bit is set or clear.

Syntax

```
TST  <Rd>, <Rs>
```

where:

- <Rd> specifies the register containing the first value. It can be any of R0 to R7.
- <Rs> specifies the register containing the second value. It can be any of R0 to R7.

Operation

```
alu_out = (Rd and Rs)
N Flag = alu_out[15]
Z Flag = if (alu_out == 0) then 1 else 0
C Flag = unaffected
V flag = unaffected
```