The P18240 Reference Manual

Written by Bill Nace

Version 1.0: 24 July 2014

Introduction

The P18240 is a special purpose instruction-set architecture (ISA) for use in the class 18-240: Structure and Design of Digital Systems. It really has no purpose any deeper than educational. As such, key components were not included in the design, to simplify and lower the learning curve for introductory students. It is not intended to be a high-performance ISA, so don't expect superscalar features, pipelining, multi-threading, etc. — you will learn plenty about those features in your computer architecture class.

This manual collects lots of relevant information about the P18240 in one place. It is intended to supplement the lectures and lab experience; documenting the assembly language, tools, protocols; disciplines and whatever else I find useful. Especially in the early versions, this document may have typos and other bugs. Please report them on piazza.com or via email to the staff list (ece240-staff@ece.cmu.edu).

Instruction Set Architecture

An instruction-set architecture (ISA) is an interface specification — a set of guidelines — about how to program a CPU or family of CPUs. Sometimes we say something like “An ISA is a contract between the chip designer (i.e. the computer architect) and the programmers.” What is meant, is that both sides; the programmer and the chip designer; need to have a common vision about how to get the CPU to run programs properly. The chip designer has built the CPU to work in a certain way and he[1] needs to be able to tell programmers the details. But, he doesn’t just want to give them a circuit diagram or a datapath picture — most programmers aren’t computer architects, so those communication methods don’t work well. Instead, the ISA is a list of instructions and other details about how the chip works — but not too many details.

The reason the ISA isn’t a complete specification for how the chip works, is that the chip designer probably intends to build other chips in the future. And, he intends those chips to work with the same software without needing a re-write. If programmers came to depend on specifics of how the first chip works, they might write programs that would lock the chip designer into building future chips that have that same specific detail —

[1] or she. More likely, the chip designer is an entire engineering team.

and thus not give the chip designer enough freedom to build the high-performance future chips he dreams of.

Visible State in the P18240

The assembly language programmer selects instructions and orders them such that they have a particular effect on the P18240 CPU. Through this process, the programmer is manipulating the *visible state* of the CPU[2]. For the P18240, there are these pieces of visible state:

Registers: There are 8 registers, each 16-bits wide. The registers are named R0 through R7

Program Counter: a 16-bit wide register that, at the beginning of the first clock cycle of the execution of an instruction, holds the memory address of that instruction.[3] The program counter is often abbreviated as "PC".

Stack Pointer: a 16-bit wide registers that holds the address of the top element on the stack, if there is one. If the stack is empty, it holds the address plus one of where the first element will go.

Condition Code Flags: There are 4 flags (single-bit fields) that are set by the ALU operations. We usually list them in the order: ZCNV

The Zero Flag (Z): This bit is set (i.e. Z=1) if the result of the last ALU operation was zero.

The Carry Flag (C): This bit is the carry-out bit of last ALU operation.

The Negative Flag (N): This bit is the most-significant bit of the last ALU operation. Therefore, if it is set (i.e. N=1), that means the result was a negative value.

The Overflow Flag (V):[4] This bit is set if the last ALU operation resulted in overflow.

[2] The programmer is also manipulating non-visible state, but they don't need to know about those bits — this is the entire point of an ISA.

[3] The reason that this definition is so tortured is that the PC is changed at different points during the execution of different instructions. At various instances, it might hold the location of the second word in a two-word instruction, the instruction following the current one in memory or the instruction to be executed next (a branch target, for instance).

[4] Not the O flag — that looks too much like a zero (and it would be weird to say things like O=0), so overflow is specified as a V.

Memory: The P18240 includes a specification of memory, unlike many CPUs. In our case, memory is an array of 65,536 (i.e. 2^{16}) words, each of which is 16-bits wide. As such, it has a 16-bit address and a 16-bit data bus. Memory has combinational read and sequential write capabilities. The sequential write only requires one clock period.

The P18240 ISA

ADD Rd, Rs Addition

Description:	Rd ← Rd + Rs (carry in = 0)
ALU Flags:	ZCNV - Set normally
Cycles:	5
Encoding:	

15 6	5 3	2 0
0000 1110 00	Rd	Rs

ADDSP imm Add to stack pointer

Description:	SP ← SP + imm
ALU Flags:	Not changed
Cycles:	7
Encoding:	

15 6	5 3	2 0
0000 1111 00	000	000
immediate		

AND Rd, Rs Bitwise AND

Description:	Rd ← Rd • Rs
ALU Flags:	ZN — Set normally CV — Zeros
Cycles:	5
Encoding:	

15 6	5 3	2 0
0001 1010 00	Rd	Rs

ASHR Rd Arithmetic shift right

Description:	Rd ←Rd[15], Rd[15..1]
	C ← Rd[0]

ALU Flags:	ZN — Set normally V=0
Cycles:	5
Encoding:	

15 6	5 3	2 0
0010 0110 00	Rd	Rd

BRA addr Branch always

Description:	PC ← addr
ALU Flags:	Not changed
Cycles:	7
Encoding:	

15 6	5 3	2 0
0010 1000 00	000	000
address		

BRC addr Branch if carry

Description:	if (C) PC ← addr
ALU Flags:	Not changed
Cycles:	7 if taken, 6 if not taken
Encoding:	

15 6	5 3	2 0
0100 1000 00	000	000
address		

BRN addr Branch if negative

Description:	if (N) PC ← addr
ALU Flags:	Not changed
Cycles:	7 if taken, 6 if not taken
Encoding:	

15 6	5 3	2 0
0010 1100 00	000	000
address		

BRV addr Branch if overflow

Description:	if (V) PC ← addr
ALU Flags:	Not changed
Cycles:	7 if taken, 6 if not taken
Encoding:	

15 6	5 3	2 0
0010 1110 00	000	000
address		

BRZ addr Branch if zero

Description: if (Z) PC ← addr

ALU Flags: Not changed

Cycles: 7 if taken, 6 if not taken

Encoding:

15 6	5 3	2 0
0010 1010 00	000	000
address		

CMI Rs, imm Compare to immediate

Description: CC ← Rs - imm (Result is discarded)

ALU Flags: ZCNV — Set based on the subtraction of Rs and imm.

Cycles: 7

Encoding:

15 6	5 3	2 0
0100 0100 00	Rs	Rs
immediate		

CMR Rd, Rs Compare to register value

Description: CC ← Rd - Rs (Result is discarded)

ALU Flags: ZCNV — Set based on the subtraction of Rd and Rs.

Cycles: 5

Encoding:

15 6	5 3	2 0
0100 0110 00	Rd	Rs

DECR Rd Decrement

Description: Rd ← Rd - 1

ALU Flags: ZCNV — Set normally

Cycles: 5

Encoding:

15 6	5 3	2 0
0001 0110 00	Rd	Rd

INCR Rd Increment

Description: Rd ← Rd + 1

ALU Flags: ZCNV — Set normally

Cycles: 5

Encoding:

15 6	5 3	2 0
0001 0100 00	Rd	Rd

JSR addr Jump to Subroutine

Description: SP ← SP - 1
M[SP] ← PC + 2
PC ← addr

ALU Flags: Not changed

Cycles: 10

Encoding:

15 6	5 3	2 0
0011 0110 00	000	000
address		

LDA Rd, addr Load absolute

Description: Rd ← M[addr]

ALU Flags: ZN — Set normally CV — Zeros

Cycles: 9

Encoding:

15 6	5 3	2 0
0000 0100 00	Rd	Rd
address		

LDI Rd, imm Load immediate

Description: Rd ← imm

ALU Flags: ZN — Set normally CV — Zeros

Cycles: 7

Encoding:

15 6	5 3	2 0
0100 1100 00	Rd	Rd
immediate		

LDR Rd, Rs Load register indirect

Description: Rd ← M[Rs]

ALU Flags: ZN — Set normally CV — Zeros

Cycles: 7

Encoding:

15 6	5 3	2 0
0000 1000 00	Rd	Rs

LDSF Rd, imm Load from stack frame

Description: Rd ← M[SP + imm]

ALU Flags: Not changed

Cycles: 9

Encoding:

15 6	5 3	2 0
0100 0000 00	Rd	Rd
immediate		

LDSP Rs
Load stack pointer

Description: SP ← Rs
ALU Flags: Not changed
Cycles: 5
Encoding:

15 6	5 3	2 0
0011 1100 00	Rs	Rs

LSHL Rd
Logical shift left

Description: Rd ← Rd[14..0], 0
C ← Rd[15]
ALU Flags: ZN — Set normally V=0
Cycles: 5
Encoding:

15 6	5 3	2 0
0010 0000 00	Rd	Rd

LSHR Rd
Logical shift right

Description: Rd ← 0, Rd[15..1]
C ← Rd[0]
ALU Flags: ZN — Set normally V=0
Cycles: 5
Encoding:

15 6	5 3	2 0
0010 0100 00	Rd	Rd

MOV Rd, Rs
Move register

Description: Rd ← Rs
ALU Flags: Not changed
Cycles: 5
Encoding:

15 6	5 3	2 0
0011 1010 00	Rd	Rs

NEG Rd
Negate (2’s complement)

Description: Rd ← - Rd
ALU Flags: ZCNV — Set normally
Cycles: 6
Encoding:

15 6	5 3	2 0
0001 0010 00	Rd	Rd

NOT Rd
Bitwise NOT

Description: Rd ← NOT Rd
ALU Flags: ZN — Set normally CV — Zeros
Cycles: 5
Encoding:

15 6	5 3	2 0
0001 1000 00	Rd	Rd

OR Rd, Rs
Bitwise OR

Description: Rd ← Rd OR Rs
ALU Flags: ZN — Set normally CV — Zeros
Cycles: 5
Encoding:

15 6	5 3	2 0
0001 1100 00	Rd	Rs

POP Rd
Pop from the stack

Description: Rd ← M[SP]
SP ← SP + 1
ALU Flags: Not changed
Cycles: 7
Encoding:

15 6	5 3	2 0
0011 0100 00	Rd	Rd

PUSH Rs
Push on the stack

Description: SP ← SP - 1
M[SP] ← Rs
ALU Flags: Not changed
Cycles: 7
Encoding:

15 6	5 3	2 0
0011 0010 00	Rs	Rs

ROL Rd
Rotate Left

Description: Rd ← Rd[14..0], C
C ← Rd[15]
ALU Flags: ZN — Set normally V=0
Cycles: 5
Encoding:

15 6	5 3	2 0
0010 0010 00	Rd	Rd

RTN
Return from subroutine

Description: PC ← M[SP]
SP ← SP + 1
ALU Flags: Not changed
Cycles: 7
Encoding:

15 6	5 3	2 0
0011 1000 00	000	000

STA addr, Rs
Store absolute

Description: M[addr] ← Rs
ALU Flags: ZN — Set normally CV — Zeros
Cycles: 9
Encoding:

15 6	5 3	2 0
0000 0110 00	Rs	Rs
address		

STOP Halt the CPU

Description: PC ← PC

ALU Flags: Not changed

Cycles: 5

Encoding:

15 6	5 3	2 0
0011 0000 00	000	000

STR Rd, Rs Store register indirect

Description: M[Rd] ← Rs

ALU Flags: ZN — Set normally CV — Zeros

Cycles: 7

Encoding:

15 6	5 3	2 0
0000 1010 00	Rd	Rd

STSF Rs, imn Store on stack frame

Description: M[SP + imm] ← Rs

ALU Flags: Not changed

Cycles: 9

Encoding:

15 6	5 3	2 0
0100 0010 00	Rs	Rs
immediate		

STSP Rd Store the stack pointer

Description: Rd ← SP

ALU Flags: Not changed

Cycles: 5

Encoding:

15 6	5 3	2 0
0011 1110 00	Rd	Rd

SUB Rd, Rs Subtraction

Description: Rd ← Rd - Rs

ALU Flags: ZCNV — Set normally

Cycles: 5

Encoding:

15 6	5 3	2 0
0001 0000 00	Rd	Rs

XOR Rd, Rs Bitwise XOR

Description: Rd ← Rd ⊕ Rs

ALU Flags: ZN — Set normally CV — Zeros

Cycles: 5

Encoding:

15 6	5 3	2 0
0001 1110 00	Rd	Rs

AS240 Pseudo-Instructions

Pseudo-instructions are directives that show up in an assembly program, but are intended to communicate with the *assembler* as it prepares the machine language. Pseudo-instructions don't communicate anything to the CPU (unlike an instruction) and so are not part of the ISA. Unlike an instruction, there is no machine-language equivalent for each pseudo-instructions — they don't show up in the machine language file, though they do effect how the assembler shapes the machine language.

For our assembler, we have three pseudo-instructions.

Origin (.ORG): This pseudo-instruction tells the assembler that the line following should have a particular address. Normally the next line will have an address that is zero, one or two more than the current line (depending on if this line is blank, has a short or long instruction on it). The .ORG pseudo-instruction interrupts this incremental processing and allows you to put data or instructions at any place in the P18240 memory map.

Data Word (.DW): This pseudo-instruction tells the assembler that you want a particular value to be placed in memory at the address of this line. The value can be specified as a hexadecimal value or a label.

Equals (.EQU): The .EQU pseudo-instruction assigns a constant value to a particular label. The label on this line does not equate to an address, unlike all other labels. Rather, it is assigned to whatever hexadecimal value is specified as an opcode. <TODO: Check if you can use another label as the value for a .EQU>

Assembly Program Format

An assembly program is written as a text file, in ASCII format[5], much like other programming languages. The file is named with a .asm extension.[6] Assembly programs are line-oriented; which means that the line stands alone. All information about a single instruction is contained in a single line. There is no way to have an instruction span multiple lines. On the other hand, not all lines have an instruction on them — they may be blank, contain a comment or a pseudo-operation.

On any line of the assembly file, up to four fields will occur, in the following order.[7] Each is separated by some form of whitespace (spaces, tabs, etc):

---

[5] Our assembler might be able to handle UTF-8 to allow for non-ASCII labels and comments, but I've never tested it.

[6] This isn't a hard-and-fast rule. as240 expects it, but you could work around it. Note that naming your assembly language files with a .c or .java extension is asking for trouble.

[7] Note that blank lines are legal, as they have zero of the fields.

- Label
- Instruction
- Opcodes
- Comment

Label: The label specifies a human-usable name for the *address* of the instruction on that line. Labels also can be used to name a constant specified in a .EQU pseudo-operation.

The label is an alphanumeric value starting at the beginning of the line. There should be no whitespace before the label. Labels are case-sensitive. Labels may be up to 6 characters in length. Each label must be unique — there can be no other label in the file with the same value.

You may use any combination of letters, digits and the underscore ('_') character, for a label, though I strongly recommend that it be meaningful. Also, labels that look like register names (R4) or instruction mnemonics (LDA) are frowned upon. (TODO: Check this paragraph is actually true).

Instruction: This is the mnemonic for the assembly language instruction. It must match one of the given instructions in the P18240 ISA. It is case-insensitive.

The instruction field can also contain one of the pseudo-operations: .EQU, .DW and .ORG. Pseudo-operations are also case-insensitive.

Opcode(s): Depending upon the instruction, there should be zero, one or two opcodes. If there are two opcodes, they must be separated by a comma and may have whitespace before and/or after the comma.

The nature of the opcode is determined by the instruction. Opcodes can be register names, immediate values or memory addresses.

Register names start with the 'R' character followed by a single digit, zero through seven. Register names are case insensitive.

Immediate values and memory addresses can be numerical values or label names. Numerical values are hexadecimal numbers of no more than 16-bits (i.e. 0 through FFFF). All hexadecimal values must start with the dollar sign ('$'). Leading zeros are not necessary. Hexadecimal numbers are case-insensitive.

Label names must match a value in the label field somewhere in your assembly code. Note that label names are case sensitive, so their use in the opcode field must match case.

Comment: A comment is specified by using the semicolon (';'). All text following the semicolon is ignored.[8]

The AS240 Assembler

AS240 is an assembler for the P18240 ISA. It is found in /afs/ece/class/ece240/bin and may be executed from there. You are welcome to make a copy so that you can execute it from your own machine or in your own AFS space. Note that it is a perl program, and will require a perl5 installation to execute.

Note that the directory /afs/ece/class/ece240/bin is not writable. That means you will have to be working in another directory somewhere (your own AFS space, for instance) and either specify the full path to as240 or include /afs/ece/class/ece240/bin in your path.

Normally, you would invoke the assembler in your working directory (at the very least, in a directory in which you have write permissions) and name the assembly file you wish to assemble. Like so:

```
> as240 lab4.asm
```

The assembler would read your assembly language file and produce a **lab4.list** file and a **memory.hex** file. The former shows your code and how it was converted by the assembler — all of the memory values for all of the instructions are shown, as well as label values, data words, etc. The **memory.hex** file is just a memory image, showing addresses and the values that should be stored in each. Note that the **.list** file is named with the same base filename as your **.asm** file. But **memory.hex** is always named such, regardless, in order to make the synthesis of P18240 easier.

The following command line options are available[9]:

-h, --help	Provide short help text and usage information
-m <filename>	Use the specified filename for the memory.hex file
-l <filename>	Use the specified filename for the .list file
-s [<filename>]	Output the symbol list as <basename>.sym or <filename> if specified
-	Send .list output to stdout (for piping into sim240) rather than a file.
-version	Print the version of as240 and quit.

[8] Ignored by the assembler, that is. No comment is ever ignored by the TAs. Make them good.

[9] This is a bald-faced lie. The current version (1.5.2) doesn't support any such switches, but I keep hoping that we will upgrade it to do so.

Subroutine Calling Convention

As discussed in class, the JSR and RTN instructions are provided as part of the P18240 ISA in order to support the use of subroutines in assembly code. That same lecture stressed the importance of having a standardized calling convention (or calling discipline). While you could probably hack together other methods for calling subroutines, all code submitted for grade in 18-240 must use the class calling discipline.

The 18-240 subroutine discipline requires the following from the caller, whose main responsibility is to guarantee the stack state across the subroutine call:

Protect R7, should it contain a value you care about. Protect it on the stack (most likely) or in memory somewhere.

Pass parameters by pushing them on the stack. They should be pushed in left-to-right order, if given a subroutine signature.

Call the subroutine with JSR. Obviously, the subroutine will return and execute the next instruction, which will --

Remove parameters from the stack, either by popping them into unused registers or using ADDSP.

Obtain the return value (if any) from R7.

The subroutine is responsible for the following actions, in order to maintain the stack state within the call:

Create space on the stack for any local variables that will be used. Create this space with ADDSP (by adding a negative number). I suppose you could also push dummy values onto the stack, but that doesn't seem very efficient.

Protect values that are in registers used in the subroutine. Protect them by pushing them on the stack.

Retrieve parameters from the stack, using LDSF. Note that the LDSF offsets will change if you decide you need more local variable space, so plan well.

After the subroutine does it's function, it shall prepare for the return by:

Put the return value (if any) in R7.

Pop protected registers from the stack. Make sure to do this in the reverse order that they were placed on the stack.

Remove local variable space by ADDSPing the SP back into it's original spot (pointing at the return address).

Return with RTN.

The Sim240 Simulator

The P18240 simulator is a great resource to debug programs and understand the instruction and micro-instruction operations. The simulator, **sim240**, is found in /afs/ece/class/ece240/bin. It also is a perl program that can be copied to another machine for your own use, much like as240.

Invoke the simulator with the .list file generated by as240, like so:

```
> sim240 lab4.list
```

There are other command line options available as well:

-version	Print the version of sim240 and quit.
-run	Run only. Don't accept user input, but rather start the program, output all simulation steps until a STOP is reached (or some large number of clock cycles has passed). This is a very useful option for grading scripts.
-memory	Randomize all memory values before loading the .list memory image.
-seed=i	Specify the randomization seed for the -memory option.

If no list file is specified, sim240 will read the list file from stdin. Thus, it is easy to pipe your as240 output to sim240 without generating **.list** or **.hex** files.

Sim240 Commands

Unlike the assembler, the simulator is an interactive program that allows the user to specify what should happen throughout its use. When the simulator is invoked, a prompt will be printed on the terminal and the simulator will await user input.[10] At any prompt, the user can enter any of the following commands:

quit, q	Quit the simulator.
exit	See quit
help, h	Print a help screen
reset	Reset the processor state to where it was when the .list file was loaded. Memory and registers (including PC, SP) return to initial values. All breakpoints are cleared.[11]

[10] Unless it is operating in "Run only" mode, of course.

[11] TODO: Is this true?

run, r [n]	Simulate the next n instructions. If n is not specified, simulate until a STOP instruction is encountered. If a breakpoint is encountered, it will halt the run.
step, s	Simulate a single instruction. Technically, simulate until the next FETCH state is encountered, so If you have unstepped several states into an instruction it will only complete simulation of the current instruction.
ustep, u	Simulate a single micro-instruction, that is, a single state or single clock.
break [addr / label]	Set a breakpoint. If the address specified (or the address of the label specified) is encountered during a run, the simulator will stop at that point. This is incredibly useful for getting into the program to the point where you think there might be trouble (rather than stepping 87 times) or checking to see if some code is ever executed. For instance, you might place a breakpoint in a subroutine to see if your code ever actually calls the subroutine.
lsbrk	List all the breakpoints that you have previously set.
clear [addr / label]	Clear a breakpoint that you have previously set for this address.
save [file]	Save the current state of the simulation to a file. Useful if you are in the middle of debugging something when snack night starts. Your location, memory contents and breakpoints are all stored away.
load [file]	Load a simulation file, previously created with the save command. And, you're back!

Querying P18240 state

You may ask about the contents of memory or a register by using the question mark command. Simply specify a register (including micro architectural registers like MAR and IR) and then follow it with a question mark. Like so:

```
R3?                 MAR?                SP?                 IR?
```

If you happen to want to see the entire register file (and really, who doesn't), then you can use the wildcard asterisk like so:

```
> r*?
R0: 0087        R1: 1492
R2: 0410        R3: 1234
R4: BEEF        R5: 0000
R6: 8787        R7: A492
```

You can also view all micro architectural registers (like the MAR and IR) with *? though the output isn't quite as pretty.

```
> *?
0208  FETCH 283E 0000 FFFE 0000 283D 0000 0087 1492 0410 1234 BEEF 0000 8787 A492
```

Similarly, you can ask about a memory location. Use the m-array notation with an address between square brackets. You may use m or mem (or their capitalized versions). You need not specify leading zeros. You must use an address, not a label. By using a colon to delimit start and end addresses, you can ask about a range of values.

```
m[0000]?            M[1234]?            mem[19:87]?
```

for each memory address, you will be told what value it holds as well as what that decodes to, if it were an instruction. For example:

```
> m[6:9]?
mem[0006]: 3600 JSR 0 0
mem[0007]: 2813 BRA 2 3
mem[0008]: 0f00 ADDSP 0 0
mem[0009]: 0001 FETCH 0 1
```

Note that this example is asking about two instructions, **JSR** and **ADDSP**. Both are two-word instructions, but the printing routine doesn't try to disassemble the values stored in memory. So, the **$2813**, which is actually the address of the subroutine, is printed as a **BRA** instruction, because that's what the first word of a **BRA** instruction looks like. The **$0001** that is the immediate value for the **ADDSP** looks sort of like the microcode for a **FETCH** state, so that's what gets printed. Just be a bit careful about interpreting what you see.

Setting P18240 state

You can use similar means to change the state of a register or memory. In this case, instead of ending with a question mark, use an equals sign.

```
R2=D2               MAR=0100            M[290]=7734
```

At the present time, you cannot set a range of memory values with one command. So sorry.

P18240 Synthesis

The P18240 has been described in synthesizable SystemVerilog, capable of configuring the Altera FPGA boards. It has an FSM which controls the datapath. The datapath contains a variety of components, including an arithmetic logic unit, a register file, and multiplexers. The source can be found in the following files, which are located at /afs/ece/class/ece240/P18240[12]:

File	Contents
alu.sv	Contains the arithmetic logic unit for the P18240 processor.
constants.sv	Definitions of readable names for bit patterns like ALU operations, microcode operations, control points, etc.
controlpath.sv	The FSM that drives the datapath
datapath.sv	A collection module where all the components in the datapath are instantiated.
library.sv	A collection of parameterized components.
p18240.sv	The top-level module, which instantiates the datapath, controlpath and I/O modules. It also has an I/O module for simulation.
regfile.sv	The register file, with 8 registers, each 16-bit in size.

These files will allow synthesis, with a few caveats:

On line 33 of p18240.sv, is a comment, which will have to be uncommented before synthesizing. Make sure to re-comment it for any simulation tasks.

These files have no method for getting input to your program, nor receiving output from it. Lab4 generally requires creating some memory-mapped I/O ports and connecting them to LEDs and switches.

Machine code needs to be included in the synthesis run. The as240 assembler outputs a file named memory.hex. This file is a memory image of your program. If it is placed in a project directory, Quartus II will find it during synthesis and use it to configure memory.

[12] TODO: Make this true

Program memory starts at address $0000 and ends at $00FF. Locations $0100 through $01FF are for data (variables). The stack exists from $FF00 through $FFFF.

The memory modules are written "inside" the datapath module. This is a bit weird, but is historically how the P18240 was created. Even weirder is that the memory system is a module found in the library.sv file. My apologies.

Future Improvements

ADC: Add with carry instruction (probably subtract also)

Additional Branch instructions (BRnotN, BR>=)

Why doesn’t STSF change the flags? All the other Stores do. Ditto LDSF.

Clean up memory in the SystemVerilog files.