Smolproc: A simplistic yet powerful computational system

by

Martin Andronikos

INTRODUCTION

The architecture presented in this document is the fourth in a series of designs, each increasing in complexity, that I designed mainly as a hobby. I have since realized the potential of Smolproc as an educational tool in computer architecture classes due to its simplistic set of instructions. “Smolproc” is a portmanteau of Small Processor after all.

The design is nearing a year old as of the writing of this paper. Since then my terminology and groupings have changed closer to other popular instruction sets but in the interest of authenticity to my past self I will not change the names of the listed instructions in chapter 1. It will still be apparent what they do because C-style operations are specified along with them.

Also regarding this document, it is one of, if not the first large scale writing assignment that I have taken on. If the wording or perhaps the formatting is not entirely correct for the circumstances, I apologize.

THE INSTRUCTION SET

Smolproc's instruction set, while not containing just the bare essentials to be considered Turing Complete, is still quite stripped down. With just 28 total operation codes, it can be considered a Minimal Instruction Set Computer (MISC).

Section 1: The Registers

Before looking at the instructions themselves, let's have a look at what these instructions actually act on, or in other words, the pieces of data the processor operates on directly. The components that store this data are called registers.

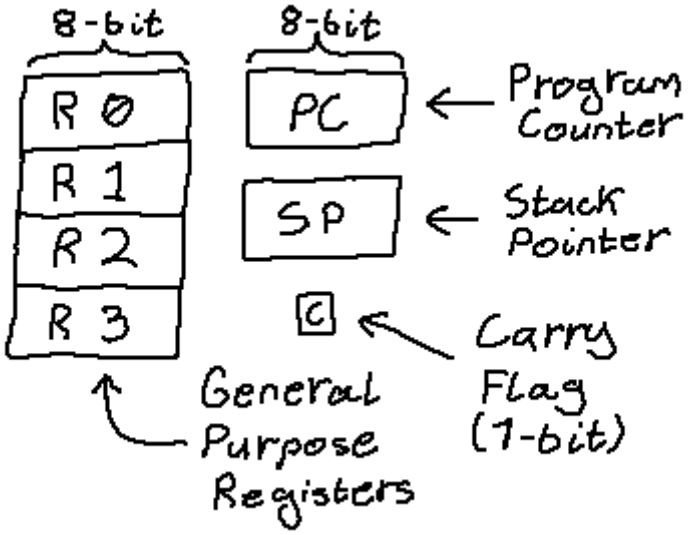


Fig 1.1: Smolproc's Registers

Figure 1.1 shows (in beautiful handwriting) all the registers and bits of memory that the processor uses for its operations. Some are of general purpose while others have a special one. We will take a look at each one separately.

The simplest registers to explain are the general purpose ones of which 4 are provided (R0 to R3). They can be used as arguments for the instructions and they hold either data or addresses. All of them contain one byte or 8 bits (the size of one memory location).

The other registers are all special purpose. Of those the one called Program Counter (PC) is the one responsible for pointing to the memory location of the currently executing instruction and is also 8 bits in size. It is updated automatically between each instruction so that the system can sequentially execute instructions placed in the memory unless the register is manually changed by a specific instruction.

There is also another special purpose register called the Stack Pointer (SP). Its job is to point to the top of the stack, which is also located in memory. It is decremented when a value is "pushed" to the stack and incremented when a value is "popped" from it. It can also be set to a value with a special instruction for example to set the initial position of the stack during initialization of the system.

The last piece of internal state the processor has is the Carry Flag (C). It is a single bit value generated from arithmetic and logic instructions to indicate a variety of things depending on the instruction. It can be used in conjunction with some specific instructions to chance the flow of program execution conditionally.

Section 2: Instruction Formats

Processors execute instructions that are in defined formats and laid out in memory in a known way. Smolproc specifically has 4 distinct formats that the instructions can have. Half take up 1 memory location (word) and the other half take 2. Care has to be taken to distinguish these two pairs from each other and resolve the true length of each instruction. That process as well as a few others happen during instruction decoding which we will discuss in more detail in chapter 3.

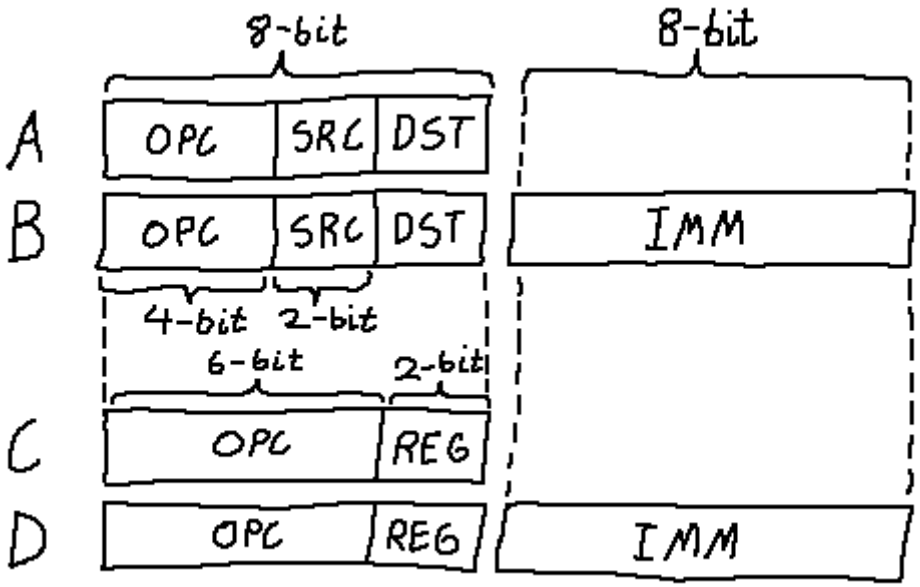


Fig. 1.2: Smolproc's Instruction Layout

As shown in Figure 1.2 (with more beautiful handwriting), the 4 formats are labeled A through D. The first pair are A and C, and the second are B and D. In each format there are 2 or 3 types of fields. OPC stands for Operation Code and is what identifies the instruction, SRC, DST and REG stand for Source, Destination, and Register respectively and are the values that select between the 4 general purpose registers to use as the instruction's arguments. Finally there's IMM which stands for Immediate (value), and it contains a direct 8-bit value to use as an argument instead of a register's value.

All of the 28 instructions are of one of these formats. The specific format they use depends on what kind of instruction it is. For example, instructions that do arithmetic between two registers use format A, while a memory access instruction would use format B or D. Formats C and D are generally used when the instruction does not need both registers as operands and instead the 2 bits for one of them is used as an extended operation code. That is the reason there is 28 instruction spaces to be had even with 4 bits of base opcodes (that's 16 spaces). Because some of these 4-bit combinations (4 in Smolproc's case) call for another 2 bits to be used, thus extending the space available for instructions.

The instruction set and the wider architecture was designed mainly with the goal to have instructions fit in 1 or 2 words of memory to ultimately allow the processor to adopt the von Neumann architecture or, in other words, have the instructions and the data share the same memory space. With just 8 to 16 bits of space to fit all of an instruction it left a lot to be desired. Only a 2 times 2-bit register address could be made to fit per instruction so as to have a reasonable space left over for opcodes. Such a configuration is atypical for the RISC family of processors that Smolproc (loosely) fits under since most other of those processors have upwards of 3 times 5-bit register addresses.

Section 3: Instruction Listing

Now that we have looked sufficiently enough into the characteristics of the instructions, where they act on, and the reasons they are laid out how they are, let's go over all the instructions one by one. First, in this section, tables with all of them will be shown. Later in section 4 each one will be examined in detail.

Opcode	Type	Mnemonic	Operation	Description
0000	A	ADD d s	d += s	Add SRC to DST
0001	A	ADC d s	d += s + C	Add SRC to DST and also + 1 if C is 1
0010	A	SUB d s	d -= s	Subtract SRC from DST
0011	A	SBC d s	d -= s - !C	Subtract SRC from DST and - 1 if C is 0
0100	A	AND d s	d &= s	Bitwise AND SRC with DST
0101	A	IOR d s	d |= s	Bitwise OR SRC with DST
0110	A	NOR d s	d = ~(d|s)	Bitwise NOR SRC with DST
0111	A	XOR d s	d ^= s	Bitwise XOR SRC with DST
1000	A	MOV d s	d = s	Set DST to SRC's value
1001	B	BNZ d s+i	d!=0 ? PC = s+i	Set PC to SRC+IMM if DST is not zero
1010	B	MST d s+i	[s+i] = d	Store DST to SRC+IMM
1011	B	MLD d s+i	d = [s+i]	Load DST from SRC+IMM
11xx	C/D	--	--	*Long Opcode Instructions*

Fig. 1.3: Primary Instruction Table

Opcode	Type	Mnemonic	Operation	Description
110000	C	RSH r	{0,r}>>=1	Shift REG right and set MSB to 0
110001	C	RSC r	{C,r}>>=1	Shift REG right and set MSB to Carry
110010	D	ADD r i	r += i	Add IMM to REG
110011	D	AND r i	r &= i	Bitwise AND REG with IMM
110100	C	PSH r	[SP--] = r	Store REG to SP then decrement SP
110101	C	POP r	r = [++SP]	Increment SP then load REG from SP
110110	C	SLD r	SP = r	Overwrite SP with REG's value
110111	D	BRC i	C=1 ? PC = i	Set PC to IMM if Carry is 1
111000	C	PST r	r = PC+1	Set REG to PC's value + 1
111001	C	PLD r	PC = r	Set PC to REG's value
111010	D	PLD i	PC = i	Set PC to IMM
111011	D	BNZ r i	r!=0 ? PC = i	Set PC to IMM if REG is not zero
111100	D	MST r i	[i] = r	Store REG to IMM
111101	D	MLD r i	r = [i]	Load REG from IMM
111110	D	RLD r i	r = i	Set REG to IMM
111111	C	HLT	--	Stop execution

Fig. 1.4: Secondary Instruction Table

Yes, if you can believe it, that's all the instructions this processor has. They all fit in one page with room to spare. You don't need hundreds of instructions to showcase some principles. That girth only comes in when you want to expand a processor's capabilities (which I am currently in the process of doing).

Section 4: The Instructions In-Depth

Following is a list of the 28 instructions, their full name, their bit format and how they act.

ADD d s : Register Addition

Format: 0000 SS DD

Description: DST gets the value of DST + SRC

Carry Flag: Set if the result can't fit in 8 bits

ADC d s : Register Addition with carry

Format: 0001 SS DD

Description: DST gets the value of DST + SRC + C

Carry Flag: Set if the result can't fit in 8 bits

SUB d s : Register Subtraction

Format: 0010 SS DD

Description: DST gets the value of DST - SRC

Carry Flag: Cleared if the result can't fit in 8 bits

SBC d s : Register Subtraction with Carry

Format: 0011 SS DD

Description: DST gets the value of DST - SRC - !C

Carry Flag: Cleared if the result can't fit in 8 bits

AND d s : Register Bitwise AND

Format: 0100 SS DD

Description: DST gets the value of DST & SRC

Carry Flag: Retains its previous value

IOR d s : Register Bitwise OR

Format: 0101 SS DD

Description: DST gets the value of DST | SRC

Carry Flag: Retains its previous value

NOR d s : Register Bitwise NOR

Format: 0110 SS DD

Description: DST gets the value of ~(DST | SRC)

Carry Flag: Retains its previous value

XOR d s : Register Bitwise XOR

Format: 0111 SS DD

Description: DST gets the value of DST ^ SRC

Carry Flag: Retains its previous value

MOV d s : Register Move

Format: 1000 SS DD

Description: DST gets the value of SRC

Carry Flag: Retains its previous value

BNZ d s+i : Base-Offset Branch If Not Zero

Format: 1001 SS DD IIIIIIII

Description: PC gets the value of SRC+IMM if DST is not 0 (delayed by two fetch cycles)

Carry Flag: Retains its previous value

MST d s+i : Base-Offset Memory Store

Format: 1010 SS DD IIIIIIII

Description: Store DST to memory location SRC+IMM

Carry Flag: Retains its previous value

MLD d s+i : Base-Offset Memory Load

Format: 1011 SS DD IIIIIIII

Description: Load DST from memory location SRC+IMM

Carry Flag: Retains its previous value

RSH r : Register Right-Shift

Format: 110000 RR

Description: REG gets shifted right one bit and its most significant bit becomes zero

Carry Flag: Gets the bit that was shifted out of REG

RSC r : Register Right-Shift with Carry

Format: 110001 RR

Description: REG gets shifted right one bit and its most significant bit gets the value of the carry flag

Carry Flag: Gets the bit that was shifted out of REG

ADD r i : Immediate Addition

Format: 110010 RR IIIIIIII

Description: REG gets the value of REG + IMM

Carry Flag: Set if the result can't fit in 8 bits

AND r i : Immediate Bitwise AND

Format: 110011 RR IIIIIIII

Description: REG gets the value of REG & IMM

Carry Flag: Retains its previous value

PSH r : Register Push

Format: 110100 RR

Description: Store REG to memory location SP then subtract one from the value of SP

Carry Flag: Retains its previous value

POP r : Register Pop

Format: 110101 RR

Description: Add one to the value of SP then load REG from memory location SP

Carry Flag: Retains its previous value

SLD r : Stack Pointer Load

Format: 110110 RR

Description: SP gets the value of REG

Carry Flag: Retains its previous value

BRC i : Immediate Branch Carry

Format: 110111 00 IIIIIIII

Description: PC gets the value of IMM if Carry is set (delayed by two fetch cycles)

Carry Flag: Retains its previous value

PST r: Program Counter Store

Format: 111000 RR

Description: REG gets the value of PC+1

Carry Flag: Retains its previous value

PLD r : Program Counter Load

Format: 111001 RR

Description: PC gets the value of REG

Carry Flag: Retains its previous value

PLD i : Program Counter Load Immediate

Format: 111010 00 IIIIIIII

Description: PC gets the value IMM

Carry Flag: Retains its previous value

BNZ r i : Immediate Branch If Not Zero

Format: 111011 RR IIIIIIII

Description: PC gets the value of IMM if REG is not 0 (delayed by two fetch cycles)

Carry Flag: Retains its previous value

MST r i : Immediate Memory Store

Format: 111100 RR IIIIIIII

Description: Store REG to memory location IMM

Carry Flag: Retains its previous value

MLD r i : Immediate Memory Load

Format: 111101 RR IIIIIIII

Description: Load REG from memory location IMM

Carry Flag: Retains its previous value

RLD r i : Immediate Register Load

Format: 111110 RR IIIIIIII

Description: REG gets the value IMM

Carry Flag: Retains its previous value

HLT : Halt

Format: 111111 00

Description: Stops program execution

Carry Flag: Does it even matter at this point?

It is important to point out a subtle detail that a specific group of instructions has. The branches all take effect after two fetch cycles. This behavior was partly left in to make the implementation of the processor as a pipeline easier and partly because it allows the programmer to emulate a subroutine call instruction by placing a PST instruction right after a PLD or a branch. It is also important to note that one of the two fetch cycles will always be taken up by the second immediate word, thus leaving only one word to be used in the delay slot.

THE ASSEMBLY CODE

In the name of not overcomplicating things, the assembly for Smolproc was not designed from scratch but instead a tool by the name of CustomASM over at GitHub was used. All that needed to be done was tell the tool what instruction mnemonics exist and to what machine code instructions they map to. Everything else is handled by the tool. That mostly includes labels and data definitions. We will go over what these two are and how they are used in due time. Finally, the code can also contain comments which are indicated with a semicolon (;). Everything after the semicolon is ignored.

Section 1: Mnemonics and Registers

Each assembly instruction is made out of a mnemonic which identifies which of the 28 machine code instructions will be generated and additionally up to 3 operand fields, comma delimited. They can either contain a register, or a number in the form of a literal or a label.

The mnemonics chosen are the same as the names which were used before in section 1.3 and 1.4 to identify the machine code instructions, but instead in lowercase. Additionally, the general purpose registers can be identified either with the format “rX”, where X is replaced with the number of the register, 0-3, or instead with the format “L”, where L is replaced with a capital letter between A and D.

As an example we will assemble the lines “mst r2, r3 64” and “bnz r0, 13”. The first line specifies an MST instruction with 3 operands, so opcode 1010 will be used. The source for the data to be stored is register r2, and the address to store to is register r3 + the value 64 in decimal. Our final instruction is 10101110 01000000. The second line specifies a BNZ instruction with only 2 operands, so opcode 111011 will be used. The register whose value is checked is r0, and the address to jump to if the condition is met is the value 13 in decimal. Our final instruction is 11101100 00001101.

Through the above examples it is visible that the order of the operands is important. Specifically, when an instruction uses two registers, the most significant one is placed first and when an instruction uses an immediate value it is placed last.

Section 2: Labels and Literals

When an instruction requires an immediate number, there’s commonly two uses for it. It will either get used as an operand for an operation (in which case it is preferable for it to be shown as a literal) or as a pointer into memory to either read/write at that location or jump to it and execute code from there on out (in which case it would be best to use a label reference).

Labels are preferred over a literal to indicate addresses because it makes it very easy to restructure the program as it grows. If numbers were used instead, every time instructions were moved around or added, the addresses of jumps would have to change to point to their new appropriate bounds. Instead of doing that, we can define a label somewhere and then reference it by using its name in places of numbers. During assembly, the value of the label is replaced with the address of the instruction or piece of data directly after it in the code file.

Label syntax is common between all the assembly languages defined with CustomASM. They always get defined by writing a valid identifier for the label followed by a colon (:). Then every time the address of the label is needed, only the identifier is written. This will become more clear later when some program examples are shown.

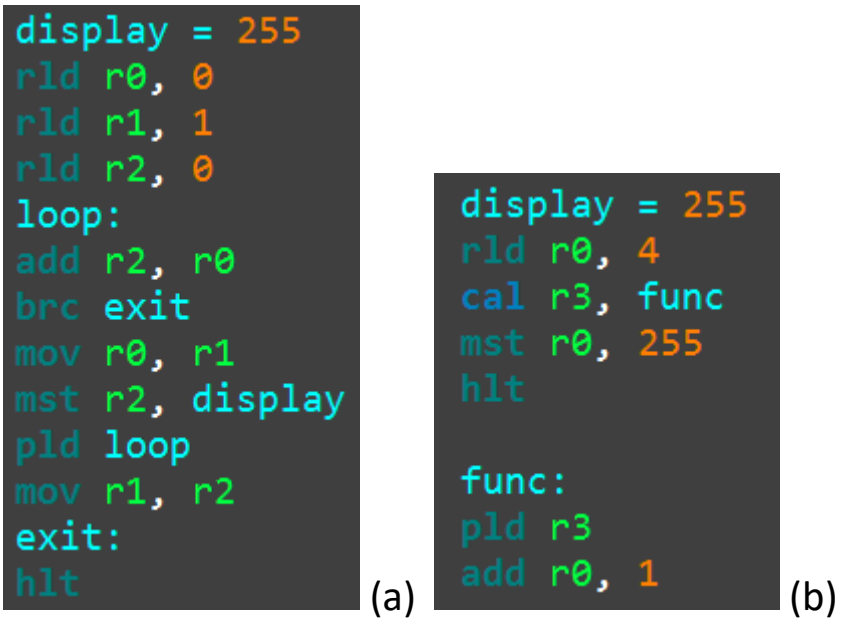
Section 3: Data Definitions and Constants

Apart from being able to assemble instructions into machine code, CustomASM also allows the placement of user provided data directly in with the assembled code. The data can be in numeric or even string form. To provide this data we write “#d” followed by the data value, or if there are many they are comma delimited. The size of the data in bits can also be specified as a number directly after, writing “#d8” to indicate a single byte of data.

An additional feature of CustomASM is constants. They act exactly like labels except that the user specifies their value explicitly instead of basing its value from the address of an instruction or piece of data. They are defined by writing a suitable identifier and an equals sign (=) followed by the desired value. They can be useful in order to, for example, give a name to an address where an important variable is stored.

All these features provided are very useful and make it easier for the programmer to keep track of the data their program uses and also to maintain the structure of their program as it grows. Following are a few example programs to put the aforementioned constructs to practical use.

Section 4: Code Examples



```
display = 255
rld r0, 0
rld r1, 1
rld r2, 0
loop:
add r2, r0
brc exit
mov r0, r1
mst r2, display
pld loop
mov r1, r2
exit:
hlt
```

(a)

```
display = 255
rld r0, 4
cal r3, func
mst r0, 255
hlt

func:
pld r3
add r0, 1
```

(b)

Fig. 2.1: Two programs written in Smolproc's assembly language

Figure 4.1a shows a program that generates the Fibonacci series of numbers (starting from 0 and ending with 233) and outputs them to the location of memory called display (where a display I/O device is connected and displaying the last number written to it).

Figure 4.1b shows a program that demonstrates the calling of a subroutine. That is partly surprising because, at first thought, the architecture of Smolproc doesn't include a "cal" instruction. In actuality "cal" is what's typically referred to as a macro-instruction. During assembly, a "cal x, y" instruction gets converted into two instructions, first a "pld y", then a "pst x". When this series of instructions is executed, the processor jumps to the address y and stores the address of the instruction right after the "pst x" into register x. Register x now contains the return address that the subroutine then jumps to when it's finished.

IMPLEMENTING SMOLPROC

Having gone over the architecture of Smolproc (what formats the instructions can have, what instructions there are and on what registers they can act on) and how code is written for it (how the code is structured, what mnemonics are available and how macro-instructions help overcome the limited set of native instructions), it is time to explore a manifestation of the architecture, both in a logic simulator and in real life.

Section 1: A Simulated Pipeline

The term “pipeline” was mentioned before, but never really explained. Pipelining is a processor design layout that, by the means of splitting the processing of an instruction into multiple clocked stages, increases the throughput of instructions. That is achieved because, since each stage is only a subset of the total required processing of an instruction it takes less time to do its job and thus the speed of the clock can be increased. Each instruction still takes the same amount of time to finish, but during that time other instructions are also in the process of going through the stages. In effect, the speed gets increased by a factor of how many stages the pipeline has.

For this reference simulated pipeline implementation the program Digital was used. It allows for circuits to be grouped together and placed inside a container called a sub-circuit. Digital also provides a lot of higher level components like whole memory units which can also conveniently be told to initialize their content from a file (that CustomASM can generate).

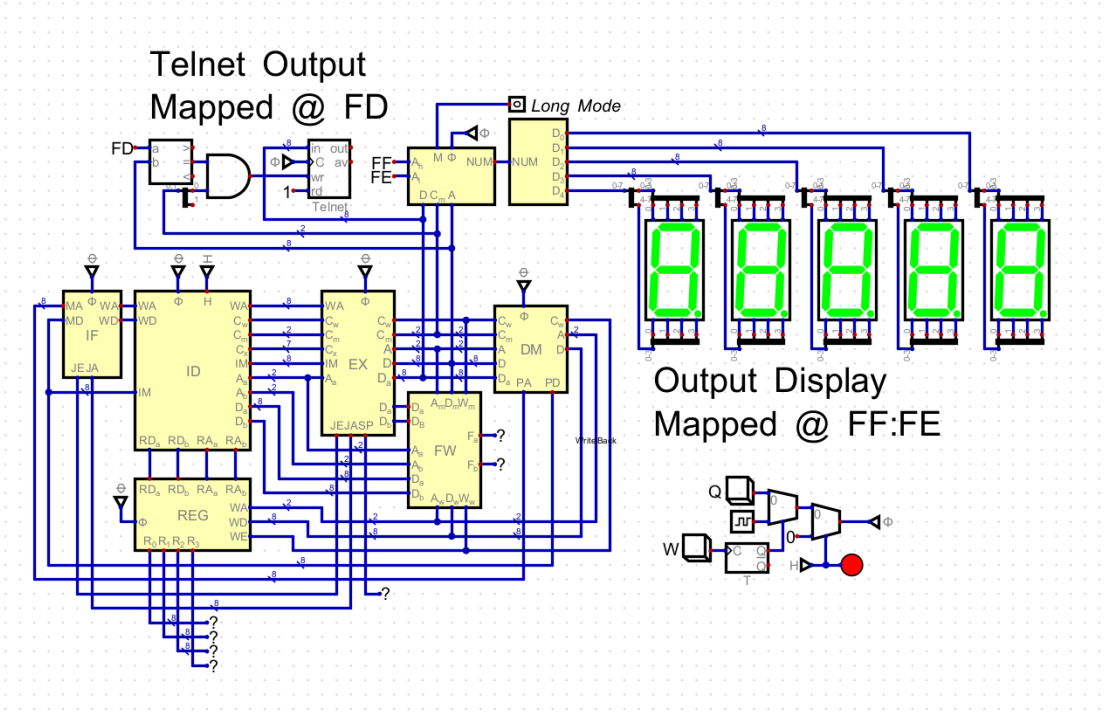


Fig. 3.1: The complete processor along with some peripherals

Figure 3.1 shows the top level circuit that contains the processor and two peripheral devices; a telnet output and a 16 bit decimal numbers display. Both of them are set to listen on the memory interface for when the processor writes to their designated address. When that happens they collect the given data and handle it. In the telnet's case it sends the byte out as an ASCII character and in the case of the numbers display it converts it to decimal and shows the number in the 7-segment displays.

Except from the peripheral devices, the processor's pipeline stages are also visible. They are the sub-circuits labeled "IF" (Instruction Fetch), "ID" (Instruction Decode), "EX" (Execute), "DM" (Data Memory) and "REG" (Register File). Thus the pipeline contains 5 stages and so each instruction takes 5 clock cycles to complete. There is also another sub-circuit labeled "FW" (Forward) which handles interactions between instructions in the pipeline.

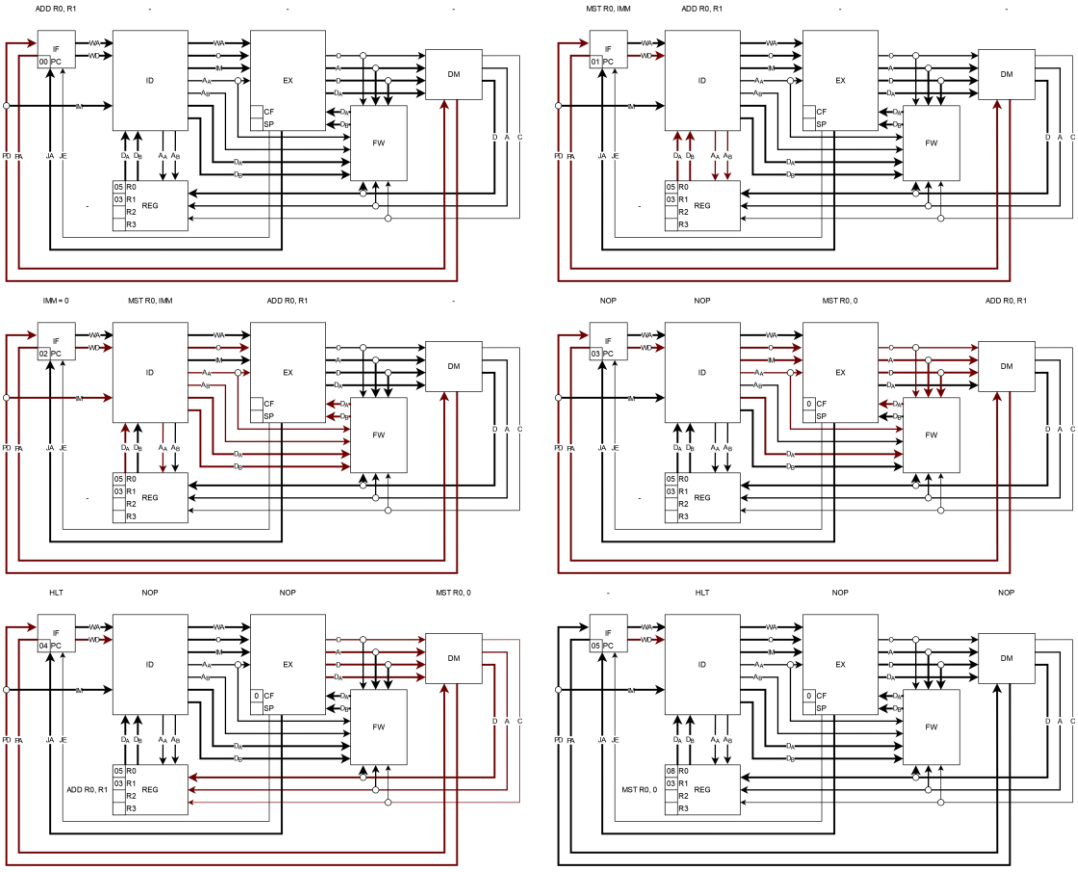


Fig. 3.2: Smolproc Executing a small program; visually shown

Figure 3.2 shows an illustrated version of the processor fetching and executing 5 words of program. Specifically, it's executing ADD r0, r1; MST r0, 0; NOP; HLT. 5 words contain only 4 instructions because the MST, being a D type instruction, requires a second word for its immediate value. During its decoding the value is grabbed directly from the fetch stage and combined with the first word of the instruction. Then on the next cycle the immediate that was fetched gets converted to a No-Operation (NOP). It is also another macro-instruction, not part of the set of instructions. NOP actually deconstructs to a MOV r0, r0 instruction. A NOP after the MST is also used to give time for the ADD instruction to write its results back to the register file before the HLT takes effect (at the decode stage).

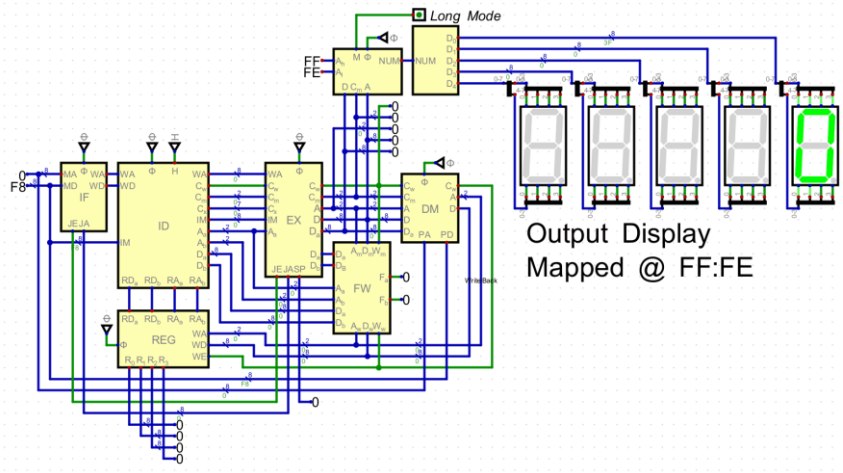


Fig. 3.3: The processor executing the program in Figure 2.1a (double-click)

Here's a more complicated example. Figure 3.3 shows the processor in the simulation executing the program shown back in chapter 2, in Figure 2.1a, calculating and displaying the Fibonacci series of numbers on the display that is connected to the memory interface. It takes about 140 clock cycles to finish, which happens when it tries to add 144 to 233 and overflows.

To view the example, you will have to double-click the image which will then open the embedded PowerPoint presentation and play the animation. There was no better way to showcase this because Word doesn't animate GIFs, nor would it be practical to put 140 images here instead.

Section 2: Making It Tangible

Our last step with this architecture is to construct a working example in real life. This was the result of months of work back when the project was initially started in early 2020. I cannot understate how important this milestone is for me. That, after all the long nights of online research, experimentation and pondering, I get to hold one of my own creations in my hand.

So then, how this was achieved was through the use of a Field-Programmable Gate Array (FPGA). An FPGA is an integrated circuit containing a matrix of programmable logic blocks interconnected by a "fabric" of wires (hence the Gate Array). Furthermore, those logic blocks can be reconfigured after the chip has left the factory, or in other words, it can be programmed by the end user to implement any combination of logic components (hence the Field-Programmable). In this case a development board from Terasic, called the DE0-Nano, was used. It contains an FPGA made by Altera/Intel, so to upload a configuration to it the software Quartus was used.

In addition to needing this special hardware, the use of a specialized programming language is also required. Specifically, of the Hardware Description Language (HDL) kind. In this case the language SystemVerilog was chosen and then began the process of writing appropriate code to generate hardware similar to the one used to build the simulated implementation. Note that Digital can generate Verilog but the quality of the generated code was not the best and as a result a hand-written approach was taken (still using the simulated implementation as a reference).

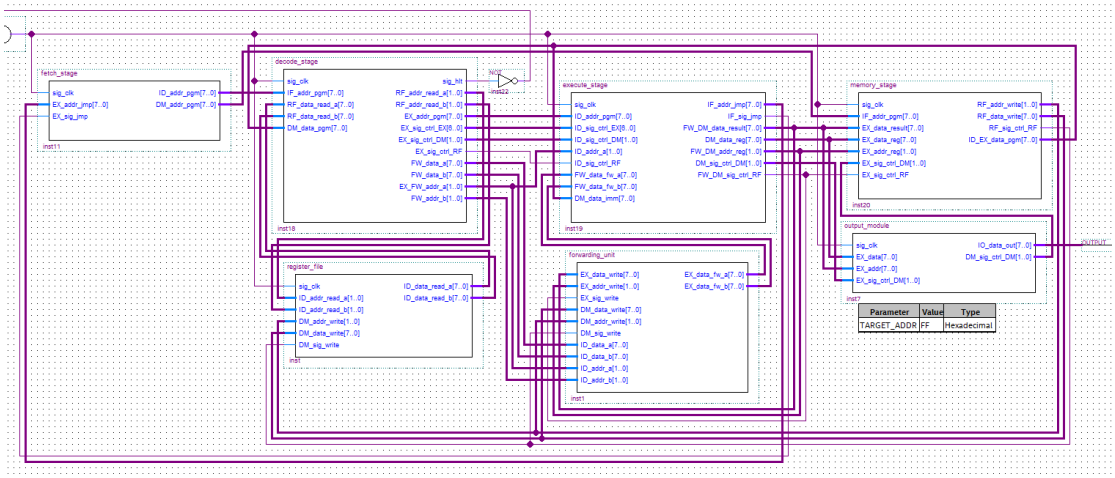
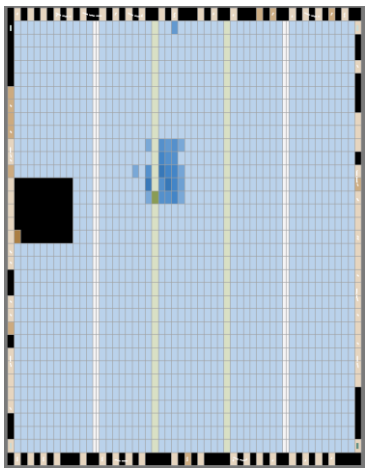


Fig. 3.4: Smolproc in Quartus

By the above figure it is visible how similar in structure the FPGA and the simulated pipelines have in common. Nothing new was introduced while transitioning to this environment, although a few rearrangements were made to get Quartus to use a dedicated memory block instead of trying to build it from logic blocks. It is also visible that this is not code but rather a diagram style design system reminiscent of the simulator. While true, Quartus does provide a “Block Design File” format, it was used only for the top level entity. All of the pipeline stages and other various components are implemented using SystemVerilog.



Family	Cyclone IV E
Device	EP4CE22F17C6
Timing Models	Final
Total logic elements	297 / 22,320 (1 %)
Total registers	146
Total pins	9 / 154 (6 %)
Total virtual pins	0
Total memory bits	2,048 / 608,256 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
Total PLLs	1 / 4 (25 %)

Fig. 3.5: Smolproc’s Footprint

The above figure shows the, frankly tiny, resource usage of the implementation. The image on the right is the results after compilation and the image on the left is an illustration that Quartus provides called the “Chip Planner”. The darker colored blocks are all that the processor occupies. Timing analysis also reported a theoretical maximum clock speed of excess of 120 MHz, which is quite impressive.

```
module register_file(
    input sig_clk,

    input [1:0] ID_addr_read_a, ID_addr_read_b,
    output [7:0] ID_data_read_a, ID_data_read_b,

    input [1:0] DM_addr_write,
    input [7:0] DM_data_write,
    input DM_sig_write
);

reg [7:0] regfile [3:0];
wire sig_fw_a = &( DM_addr_write ~^ ID_addr_read_a ) &
DM_sig_write;
wire sig_fw_b = &( DM_addr_write ~^ ID_addr_read_b ) &
DM_sig_write;

always_ff @( posedge sig_clk ) begin
    if(DM_sig_write) regfile[DM_addr_write] <= DM_data_write;
end

always_comb begin
    if(sig_fw_a) ID_data_read_a = DM_data_write;
    else ID_data_read_a = regfile[ID_addr_read_a];

    if(sig_fw_b) ID_data_read_b = DM_data_write;
    else ID_data_read_b = regfile[ID_addr_read_b];
end

endmodule
```

Fig. 3.6: A snippet of SystemVerilog from Smolproc

Here is also a code sample from the project. It specifically is the code that implements the register file. At the top are defined the inputs and outputs, and later they are interconnected with various logic elements.

So now, finally, the only thing that is left to do is upload the compiled configuration into the FPGA and watch it blink some lights. A clock speed of 10 Hz was chosen to make the following demonstration advance at an acceptable speed.

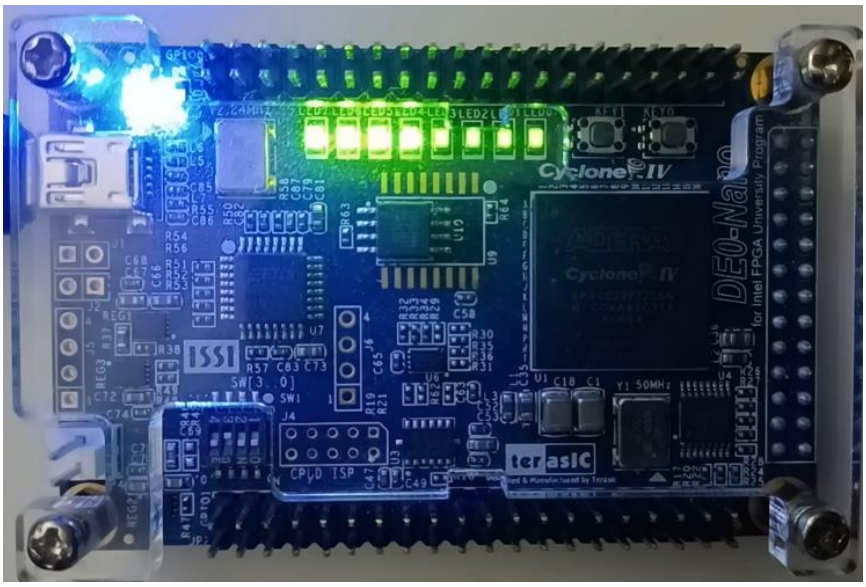


Fig. 3.7: The FPGA ready for configuration (video)

And everything goes as planned: The FPGA stops its test pattern when the button to start the programming on Quartus gets pressed. From there on out, configuration takes a few seconds. Then the LEDs turn off fully and the Fibonacci numbers begin to be displayed in binary. 00000000, 00000001, 00000010, 00000011, 00000101, 00000100, 00001101, 00010101, 00100010, 00110111, 01011001, 10010000, 11101001, are displayed in that sequence and then the program halts. The processor is seemingly working.