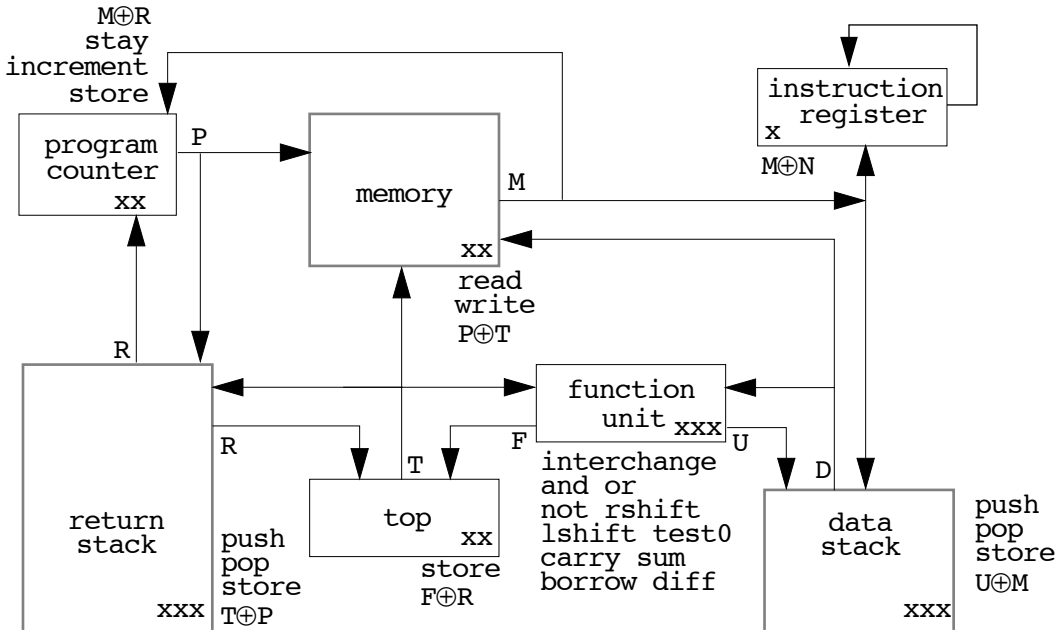
A Writable Computer

Rob Chapman

1 Abstract

Some digital design solutions can best be described in either hardware, software or both. When looking at a complete system, software means adding a microprocessor with all the support logic and memory. The computer described, is a simple stack processor. It can be built with a small number of parts that can fit inside of a single programmable logic chip (PL chip). The processor is used to execute instructions for sequential programs. The rest of the PL chip is used for custom hardware. In this heterogeneous solution, you not only write the hardware and software, you also write the computer. In the world of acronyms: CISC (complex instruction set computer), RISC (reduced instruction set computer), WISC (writable instruction set computer), MISC (minimum instruction set computer), this is a WC (writable computer).

2 Basic Sketch



All the components in the above diagram are inside a PL chip except the ones in dotted lines. The dotted components represent external RAM chips plus some logic on the PL chip. If the PL chip has room, the stacks reside on the PL chip. As well, if the program is small enough, the memory may reside on the PL chip. The operations for each of the blocks are listed beside them. The function unit can be customized by adding/changing functions to speed up certain algorithms that the processor sequences through. Trade-offs can be made for speed and size, shuffling design back and forth between hardware and software. The width of the data paths can be varied to match the environment that will be interfaced.

To make instruction coding much simpler, a stack based design was chosen. There are no register sets to choose from as in a conventional processor where there can be one, two or even three operands from a choice of sixteen or more.

2.1 Instructions

This table lists all the component instructions with a brief description. Each component can perform one instruction each cycle.

component	instruction	description
Function Unit	aor	top AND data stack are available to top; top OR data stack are available to the data stack
	ntr	the complement of top is available to top; data stack shifted right one bit is available to the data stack
	ltz	the top shifted left one bit is available to top; a flag indicating whether data stack is zero is available to the data stack
	+'	partial addition operation
	-'	partial subtraction operation
Program Counter	pstay	keep current program counter
	pstor	store return stack into program counter (return)
	pstom	store memory contents into program counter (jump, branch, jsr)
Memory	mrdp	read memory cell pointed to by the program counter
	mrdt	read memory cell pointed to by the top register
	mstop	store data to the memory cell pointed to by the program counter
	mstot	store data to the memory cell pointed to by the top register
Top	tstof	store function unit output 1 into top
	tstor	store top of the return stack into top
Data Stack	dpop	pop the top item off the data stack
	dpshu	push the function unit output 2 onto the stack
	dpshm	push the memory contents onto the stack
	dstou	store the contents of the function unit onto the stack top
	dstom	store the contents of the memory cell onto the stack top
Return Stack	rpop	pop the top item off the return stack
	rpsht	push the top register onto the stack
	rpshp	push the program counter onto the stack
	rstot	store the contents of top
	rstop	store the contents of the program counter
Instruction register	nop	execute an internal nop using data flow N

2.2 Control

There is no execution unit, microcode sequencer, multi-state controller, bus controller, pipeline or cache. The output bits of the instruction register control all the components directly. The `x`'s in the component boxes, indicate how many bits of control are needed to control them. There are 16 x's on the diagram. Only 16 bits are needed from the instruction register. This is a nice match with 16 bit memory but as we'll see later, we are not restricted

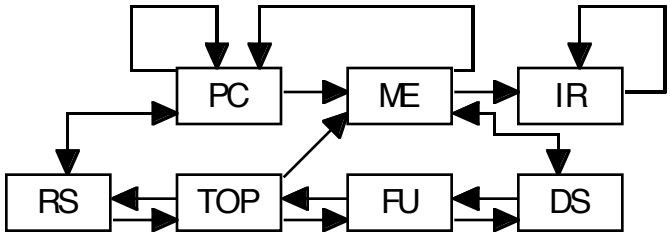
to 16 bits.

Operation is synchronous. All data transfers take place on the positive clock edge. This allows many data paths to be active at the same time. For instance, this allows the program counter to be pushed to the return stack while a new one is being loaded from memory for a jump to subroutine (if the program counter wasn't pushed, this would just be a jump).

2.3 Data

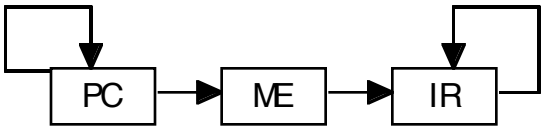
Each component has at most two data paths feeding into it, one of them will be selected as needed. The data flows are all valid on the positive edge of the system clock. Data paths out of and into a block are valid at the same time since all transfers take place on the same clock edge.

3 Data Flow Analysis



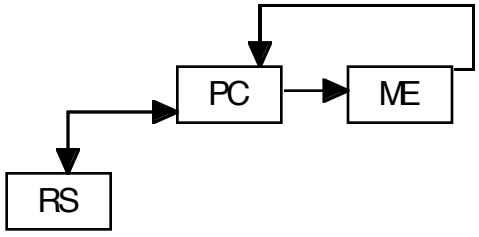
This diagram shows all the data flows for the computer. Any blocks which have two inputs will choose the input to use. The memory block has two inputs for address and one input for data. The function unit accepts two inputs and has two separate outputs.

3.1 Instruction Input



The next instruction to be loaded into the instruction register comes either from `memory` or the `instruction register`. Since `memory` is being shared with data, there are times when it cannot supply the next instruction which is then taken from the `instruction register`. For sequential operation, when the instruction is latched, the `program counter` is incremented to the next location.

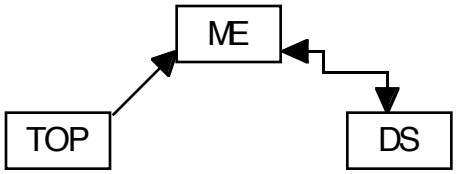
3.2 Program Branch



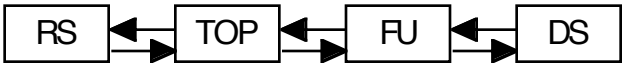
Program instructions are normally taken sequentially from `memory` as the `program counter` is incremented. However, to implement programs which involve multiple instruction sequences, there must be a way to change the program counter to a non sequential location. This is done by writing a value from either the `return stack` or `memory` to the `program counter`. The contents of the `program counter` may be saved for later on the `return stack`. If the `program counter` is written, but not pushed to the `return stack`, then this corresponds to a jump in the program execution. If the `program counter` is pushed when it is written, this corresponds to a nesting of the current program execution. A new program sequence may be run and then the previous one returned to later.

3.3 Data Transfer

When data is transferred to and from memory, the top register supplies the address to memory and the data is read from or written to the data stack.



3.4 Data Operations



Data operations are performed by feeding the contents of the top register and the data stack into the function unit. The outputs of the operations are fed back to the top register and the data stack. This allows two operations to be performed in parallel. Values may be pushed to the return stack for intermediate calculations.

4 Testing

Using a high level language to write a test program, I will describe a simple data transaction involving two input variables, an addition, and a saved result. This test program will be compiled, optimized and run to test the architecture and show how the computer is written.

4.1 Test Program

```
3 VARIABLE X X ! ( the variable X contains a value 3 )
6 VARIABLE Y Y ! ( the variable Y contains a value 6 )
0 VARIABLE Z Z ! ( the variable Z is for holding the result )

: TEST  ( -- )  X @  Y @  +  Z ! ;
```

The high level algorithm of statistical addition from previous work[1]:

```
: + ( a \ b -- c ) BEGIN 2DUP AND 2* >R XOR R> DUP 0= UNTIL DROP ;
```

Here's a translation to stack processor instructions:

- for +

```
BEGIN
2DUP AND 2* >R XOR R> DUP 0= UNTIL [ +'  pstay  tstof  dstou ]
DROP ; [ tstof  dpop  rpop  pstor  nop ]
```

- for TEST

```
X [ dpshm  mrdp  nop ] [ address of X ] [ tstof  dstou ]
@ [ mrdt  tstof  dstom ] [ tstof  dstou ]
Y [ dpshm  mrdp  nop ] [ address of Y ] [ tstof  dstou ]
@ [ mrdt  tstof  dstom ] [ tstof  dstou ]
+ [ rpshp  pstom  nop ] [ address of + ]
Z [ dpshm  mrdp  nop ] [ address of Z ] [ tstof  dstou ]
! [ mstot  dpop ] [ tstof  dpop ]
; [ pstay ]
```

All the instructions within the brackets are done at the same time and take one cycle. All the instructions on a line are required to implement the high level code. The instructions for + are a jump to its address of where it exists in memory so that its instructions are run as a subroutine. But since the length of the definition for + is only two instructions, it is at no cost to in-line the function and it'll save two execution cycles as well as the two memory cells that the + routine would have occupied.

4.2 Optimizing the Test

The instructions are now optimized for the architecture taking advantages of parallel data flow and redundant instructions. This table shows the phrases they implement.

phrase	fu	pc	me	tp	ds	rs	ir
X @		pstom	mrdp			rpshp	
		pstor	mrdp	tstof	dstom	rstot	nop
	address of X						
Y @		pstom	mrdp			rpshp	
		pstor	mrdp	tstof	dstom	rstot	nop
	address of Y						
+ Z !	+'	pstay/pstom	/mrdp	tstof/tstor	dstou/dpshm	/rpop	
		pstay	mstop	tstor	dstou	rpop	nop
	address of Z						
;		pstay					

The double instructions in the first line of + Z ! separated by a slash are conditional on the value in top. If top is not zero, then the first instruction will be executed, otherwise if top is zero, then the second instruction will be executed. If there is no first instruction, then it is left blank.

Initially a minimum of 18 cycles were needed to run the test program. Now that we have taken advantage of the architecture's parallel data flows and programmable instruction module, we only need a minimum of 10 cycles. Also, there are really only two different opcodes: one for fetching the contents of a variable; and one for adding the two values, storing them into a third variable and then stopping the processor.

4.3 Program Integration

The original high level program is translated down into two parts described in VHDL. The first part is the instruction set and sequences. These become part of the instruction.vhdl file. The second part of the program are the opcodes and data space which become part of the memory.vhdl file. The entire program, data, instruction set and stack processor components are described in VHDL files.

memory address	content or opcode	interpretation
0	2	address @
1	6	address of X
2	2	address @
3	7	address of Y
4	3	+ address !
5	8	address of Z
6	3	X
7	6	Y
8	-	Z

This is the new assembler.

This table lists the contents and interpretation of the memory contents implemented in the VHDL file. The program, or opcodes, are hi-lighted.

4.4 Synthesis

For this program, only 4 bits are needed for encoding the binary value of the numbers so this becomes the natural size for data flows and memory. It also speeds up synthesis time and make schematics readable. The 4 bits can be used to encode up to 16 different 16 bit

control vectors for the processor.

Synthesis was successful into FPGA technology and the design was simulated. At that level, the technology used was mostly CLBs. To proceed deeper into actually compiling the FPGA map for the chip, tools external to Synopsis were required from Xilinx. The final translation ended up with some errors in generated file names and it was decided to pursue the matter at a later date as the simulation tools at hand were quite adequate.

5 Comments

The implications of these results are encouraging. One can imagine a program described in some high level way being compiled down into a hardware description, a processor with an instruction set and a program to run. We no longer need to write for a specific processor and instruction set. This solves the same problem that virtual machines (VMs) like Java VM address but at a much deeper level. Indeed, VMs for other microprocessors could be built.

The final test program contained three nops. They occur when the transition is from an instruction register sequence out to program memory to get a new opcode. The memory wasn't available for an instruction fetch on the previous cycle because it was being used for a data access, so one cycle must go to a nop whose only purpose is to bring in a new opcode to begin a new sequence. If on the other hand, the entire program lived within the instruction register, then nops could be eliminated. There would only be two different opcodes needed so they could be encoded in one bit. Memory would only need three locations for the variables so it would become quite small. This approach would only work for small programs but it separates data from program and does two things:

- provides a Harvard-like architecture where data and program live in separate spaces
- eliminates the instruction register from the critical path of PC->ME->IR

Indeed, in the case where a large program was required to run very fast that dealt with a lot of data, this same approach could be used. If the program in the instruction register became large enough, then one would consider turning the decode logic into a ROM, or a regular two logic array. I'm not sure if one could get away from program counters completely and it might make sense to add one to the instruction register for even larger programs.

6 References

1. "A Stack Processor: Project Report for EE635", Rob Chapman
2. "A Stack Processor: Synthesis" project report for EE602, Rob Chapman
3. The Designer's Guide to VHDL, Peter J. Ashenden