Project Documentation

Virtual CPU with 16kB Memory

Daniel Tudzi

Instructor: Prof. Paul Michaud

April 27, 2015

Table of Contents

List of Tables

List of Figures

1. Introduction

This documentation deals with the various ways a CPU operates. This specific documentation is for a 32bit Virtual CPU which uses 16 bit instructions. There is an available 16kB of memory allocated to the Virtual CPU. The Design of this CPU is a scaled down version of the ARM cortex M0+. The processor has the following 32-bit registers:

REGISTERS

- (16 32-Bit) general purpose registers, r0-r15

SPECIAL PURPOSE REGISTERS

- Stack Pointer (SP), r13
- Link register (LR), r14
- Program Counter, r15

The general purpose registers have no assigned usage architecturally. The 3 special purpose registers; *Stack Pointer*, uses r13, the *Link Register*, uses r14 which receives the return address from PC when a Branch or Branch with Link instruction is executed and lastly the *Program Counter* which stores the address of the next instruction to be executed, uses r15.

The CPU will simulate register to register transfers. The program has the following features:

- A simple user interface that will allow the test machine level programs written for the CPU
- Read files, including binary files to the virtual memory
- Write contents of the Virtual memory to disk
- Display the contents of the virtual memory
- Single step instructions
- Display registers

OTHER REGISTERS & FLAGS:

- CCR (Condition Code registers) - Sign, Zero and Carry Flags
- Stop Flag - Flag set by stop instruction
- IR active Flag - Selects active IR

- MBR (big endian) - (32-bits) Memory Buffer Register
- MAR - (32-bits) Memory Address Register
- IR0, IR1 - (16-bit) 2 Instruction registers

2. Final State of Program

The program is 95% complete, with the remaining 5% representing debugging of the pull instruction which has been tested to the best of my knowledge but not a 100%. The main parts of the program which deals only and basically with memory such as; the load file, dump memory, the user interface and display memory contents have been vigorously tested and proven to work.

The second half which deals with the CPU and registers have been completed and works perfectly. Visible issues are as follows:

1. Pull Instruction not setting the right PC value
2. Above problem causes a fetch violation
3. PC value picked up by MAR is way out of bounds for the allocated virtual memory
4. Crashes when run under Microsoft Visual Studio
5. Returns Segmentation Fault when run with a gcc compile.

The above said issues were experienced when I created my own bin file with push and pull register instructions.
I graded this Program as 95% complete due to the problems I encountered during the implementation of the PULL register instruction.

3. HOW THE PROGRAM WORKS

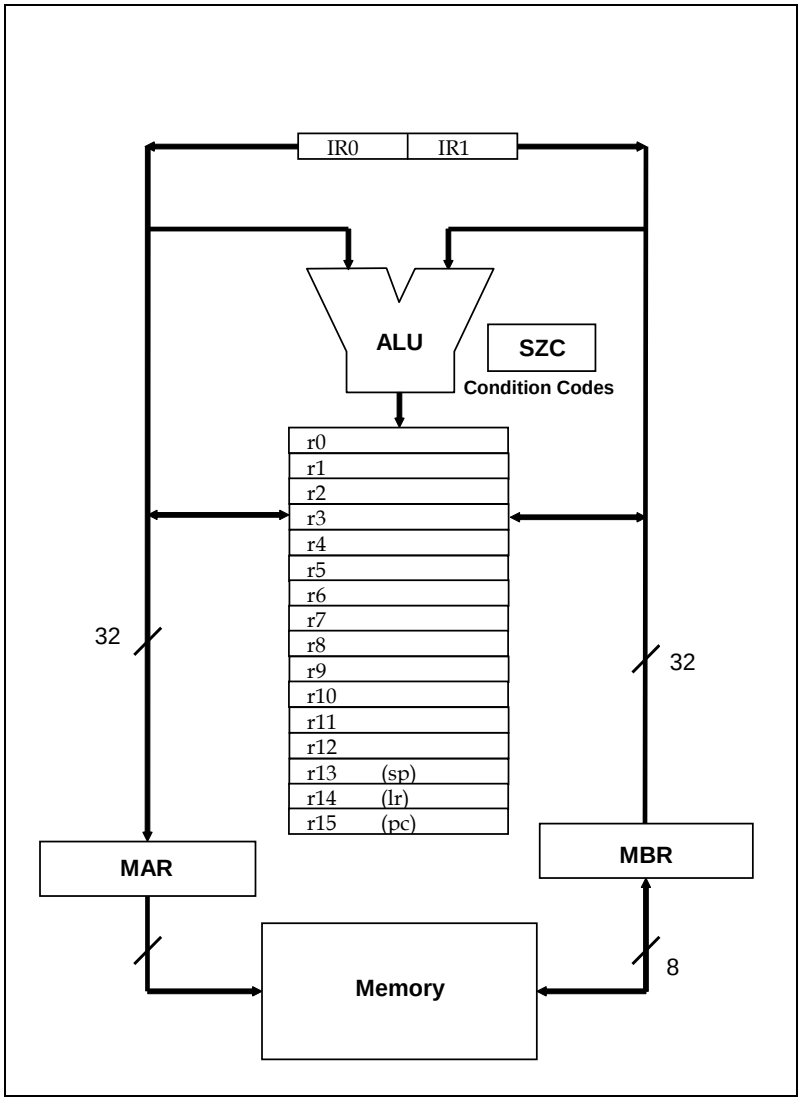

Figure 1 – Virtual CPU Flow Diagram

4. OPERATIONAL STEPS

The following steps could be done per the users order, but the load file option should be run to actually see results per how the Program works.

- At startup, the registers and flags are cleared (set to zero)
- The main menu is displayed which asks for user input (ignores case)
- User selects ‘q’ to quit and exit the program
- User selects ‘l’ to load a file from disk which could be any file including a binary file
- The contents of the file are then loaded into memory (truncated if file is larger than the given virtual memory size)
- User selects ‘d’ to view the contents of the loaded file in memory by specifying the start location to view in memory and an offset value.
- User selects ‘m’ to modify the contents of virtual memory
- User selects ‘w’ to write what has been loaded into memory to a new file on disk
- User selects ‘t’ to step through the loaded program
- Select ‘r’ to display the contents of the general purpose registers which include the Flags
- User selects ‘z’ to manually reset all the registers
- Option ‘g’ is go which runs the loaded program in virtual memory at once.

5. SAMPLE OUTPUTS

```
>> r
****** Display Registers ******!

r00:00000002    r01:ffffffff    r02:57595b5d    r03:56575859    r04:00000114
r05:00000124    r06:00000134    r07:0000000e    r08:00000000    r09:00000000
r10:00000000    r11:00000000    r12:00000000    r13:00000000    r14:00000000
r15:00000046    MAR:00000042    MBR:e0000000    IR :e0000000
ZCS:IRST:
110:1 1
IR0: e000       IR1: 0000
>>
```

Figure 2 – Register Display

```
>> d

****** Dump Memory ******

Enter value for offset in Hex: 0
Enter number of bytes to display: 30
Offset is : 0 and length is : 30
0000: c0  20  00  00  00  04  00  00  01  00  00  00  01  10  00  00
       .       .   .   .   .   .   .   .   .   .   .   .   .   .   .
0010: 01  20  00  00  00  00  00  00  00  00  00  00  00  00  00  00
       .       .   .   .   .   .   .   .   .   .   .   .   .   .   .
0020: 40  20  28  01  40  67  28  74  40  a7  28  75  40  e7  28  76
       @       (   .   @   g   (   t   @   .   (   u   @   .   (   v
>>
```

Figure 3 – Load File to Memory Option

6. TESTING

The tables below show a summary of specific tests for the Program

LOAD FILE	
Tests	**Current Outputs**
Loading Files	Loads all files correctly, Including bin and exe files
Accuracy	Truncates files that are larger than memory to avoid overruns.

Table 6.1 – Load File Test

MEMORY DUMP	
Tests	**Current Outputs**
Dumping Memory	Dumps the contents of memory from the users given offset and the users range
Accuracy	Dumps exactly what is expected and does not display memory contents over the declared memory capacity

Table 6.2 – Memory Dump Test

MODIFY MEMORY	
Tests	**Current Outputs**
Modifying Memory	Modifies starting from the users input address or location and goes on to the next location and waits for user to modify that also and so on.
Accuracy	Modifies exactly the location specified with the right hex values. Does not exceed the given memory boundaries.

Table 6.3 – Modify Memory Test

WRITE FILE	
TESTS	**CURRENT OUTPUTS**
Write to disk	Writes the exact content of memory specified to the disk
Accuracy	Contents of memory written matches contents of created file

Table 6.4 – Write File Test

DISPLAY REGISTER	
TESTS	**CURRENT OUTPUTS**
Display register	Displays the contents of all general purpose registers. MAR, MBR and Flags with their current contents
Accuracy	Contents are correct and display as expected

Table 6.5 – Display Registers Test

GO	
TESTS	**CURRENT OUTPUTS**
Run entire Program	Runs entire program till stop flag is set. Results are accurate as expected.
Accuracy	Program stops at the stop flag being active always. Which makes the program to run in an infinite loop

Table 6.6 – GO Test

7. CONCLUSION

The successful completion of this projects iterates the various processes a CPU operates. Bit shifting and bit masking regularly implemented in this code shows into detail how instructions are generated and decoded. The Fetching of 32-bit data from memory and splitting it into 2 16-bit registers was done and experienced in this Program. The most important part for me was the decoding of the various Instructions the CPU processes. Unsigned variables were converted to sign and most arithmetic functions were implemented and also using of the ALU.

8. APPENDIX

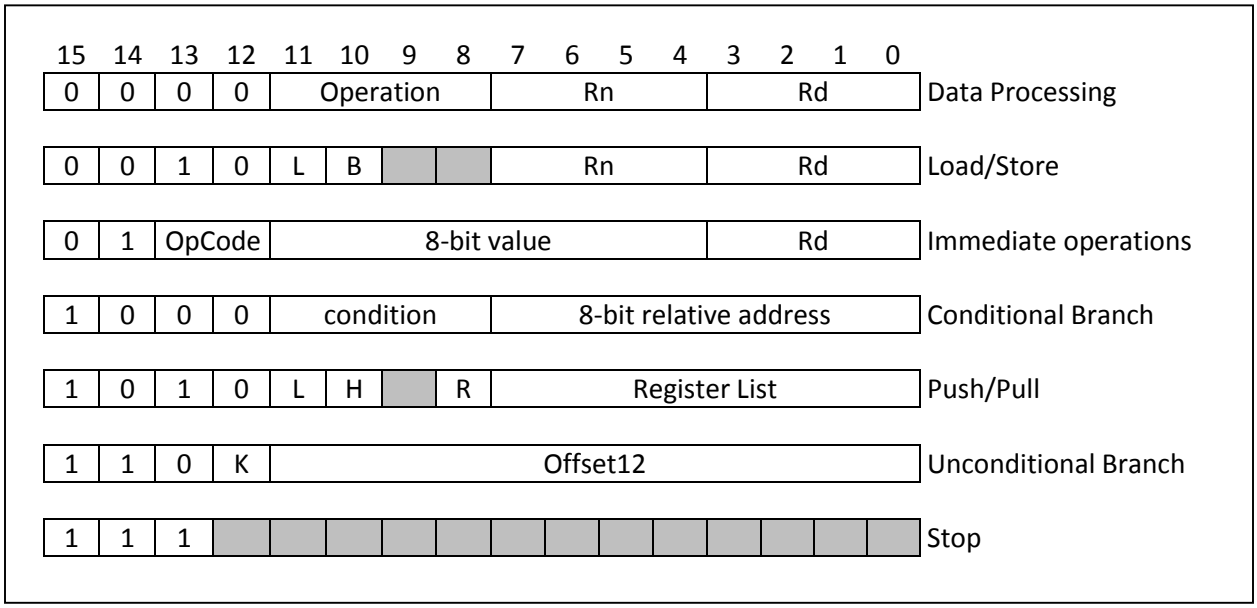

Figure 4 – Instruction Formats

Instructions (see Figure 2 for Instruction formats)

Data Processing Instructions

- The Operation field specifies the logical or arithmetic operation. (See Table 1)
- The Rd register is used as one of the operands and as the destination of the result.
- The Rn register is used as the second operand.

Load/Store Instructions

- Rn - value is used as the memory address in the transfer
- Rd - Source/Destination Register
- L - Load/Store bit: 0 = Store to memory , 1 = Load from Memory
- B - Byte/Word bit: 0 = transfer Word, 1= transfer byte
- Bytes go into bottom byte of 32-bit registers. The unused bits are filled with zero.

Immediate Instructions

- Only four operations are possible with immediate values: move, compare, add and subtract. See Table 2 for the coding of the operations.
- The immediate value is an 8-bit value (zero extended).

Operation	Code	Description	Flags NCZ
AND	0000	Rd :=Rd AND Rn	n-z
EOR	0001	Rd := Rd EOR Rn	n-z
SUB	0010	Rd := Rd - Rn	ncz
SXB	0011	Rd := (signed)Rn_{byte}	n-z
ADD	0100	Rd := Rd + Rn	ncz
ADC	0101	Rd := Rd + Rn + C	ncz
LSR	0110	Rd := Rd >> Rn	ncz
LSL	0111	Rd := Rd << Rn	ncz
TST	1000	Rd AND Rn	n-z
TEQ	1001	RD EOR Rn	n-z
CMP	1010	Rd - Rn	ncz
ROR	1011	Rd := Rd rotated right by Rn	ncz
ORR	1100	Rd := Rd OR Rn	n-z
MOV	1101	Rd := Rn	n-z
BIC	1110	Rd := Rd AND NOT Rn (bit clear)	n-z
MVN	1111	Rd := NOT Rn	n-z

Table 8.1 – Opcode for data processing

OpCode	Code	Description	Flags NCZ
MOV	0 0	Rd := immediate value	n-z
CMP	0 1	Rd-immediate value	ncz
ADD	1 0	Rd := Rd + immediate value	ncz
SUB	1 1	Rd := Rd - immediate value	ncz

Table 8.2 – Opcodes for Immediate instructions

Conditional Branch Instructions

- The offset is an 8-bit relative address. The 8-bit signed value is added to the Program counter.
- The conditions are defined in Table 3.

Push/Pull Instructions

- L - Load/Store bit: 0 = PSH, 1 = PUL
- R - 0 = no extra pulls or pushes, LR, 1 = pull PC/push LR
- H - High/Low bit: 0 = Low registers(0-7), 1= High Registers (8-15)
- The register list is a 8 bit field with each bit corresponding to a register
- The registers are pushed in order from highest to lowest and pulled from lowest to highest.
- SP (R13) register is automatically used
- SP is pre-decremented for each byte pushed. Because of the big-endian architecture the first byte pushed will be the LSB of the register.
- SP is post-incremented for each byte pulled. Because of the big-endian architecture the first byte pulled will be the MSB of the register.

Unconditional Branch Instruction

- The offset is a 12-bit absolute memory location
- K - Link bit: 0 = Branch (BRA), 1 = Branch with link (BRL)

Stop Instruction

- Sets an internal stop flag which stops further instructions from being fetched.
- Used to return control to user interface when a program is run.

Suffix	Code (bits)	Code (hex)	Flags	Meaning
EQ	0000	0	Z set	Equal
NE	0001	1	Z clear	Not Equal
CS	0010	2	C set	unsigned higher or same
CC	0011	3	C clear	unsigned lower
MI	0100	4	N set	negative
PL	0101	5	N clear	positive
HI	1000	8	C set and Z clear	unsigned higher
LS	1001	9	C clear or Z set	unsigned lower or same
AL	1110	E	Ignored	Always

Table 8.3 – Condition Codes

Shift and Rotate Instructions

The operation of the shift and rotate instructions are illustrated below.

Figure 5 – Shift and Rotate Instructions

Opcode	Instruction	Coding (Hex)	Description	Flags	Example
ADC	Add with Carry	05nd	Rd := Rd + Rn + C	ncz	ADC r1,r2
ADD	Add Add immediate	04nd 6iid	Rd := Rd + Rn Rd := Rd + immediate	ncz	ADD r1,r2 ADD r1,#3A
AND	And	00nd	Rd := Rd AND Rn	n-z	AND r3,r12
BIC	Bit clear	0End	Rd := Rd AND NOT Rn	n-z	BIC r1,r2
BRA	Branch	Cooo	PC := offset	---	BRA next

BRL	Branch with link	Dooo	LR := PC, PC:=offset	---	BRL subroutine
BXX	Conditional branch	8xoo	PC := PC+offset if true	---	BNE again
CMP	Compare Compare immediate	0And 5iid	Rd – Rn Rd – immediate	ncz	CMP r1,r2 CMP r1,#3A
EOR	Exclusive or	01nd	Rd := Rd EOR Rn	n-z	EOR r1,r2
LDB	Load register byte	2Cnd	Rd := $[Rn]_{byte}$	---	LDB r1,[r7]
LDR	Load register	28nd	Rd := [Rn]	---	LDR r2,[r7]
LSL	Logical shift left	07nd	Rd := Rd << Rn	ncz	LSL r1,r2
LSR	Logical shift right	06nd	Rd := Rd >> Rn	ncz	LSR r1,r2
MOV	Move Move immediate	0Dnd 4iid	Rd := Rn Rd := immediate	n-z	MOV r1,r2 MOV r1,#3A
MVN	Move Not	0Fnd	Rd := NOT Rn	n-z	MVN r1,r2
ORR	Or	0Cnd	Rd := Rd OR Rn	n-z	ORR r1,r2
PSH	Push registers	A0rr	[--SP] := registers	---	PSH {r3,r4,r6,r7}
PSHH	Push registers high	A4rr	[--SP] := registers	---	PSHH {r8,r9 ,r11}
PSHR	Push registers and LR	A1rr	[--SP] := registers	---	PSHR {r1,r2,r3}
PUL	Pull registers	A8rr	registers := [SP++]	---	PUL {r3,r4,r6,r7}
PULR	Pull registers and PC	A9rr	registers := [SP++]	---	PSHR {r1,r2,r3}
PULH	Pull registers high	ACrr	registers := [SP++]	---	PULH {r8,r9,r11}
ROR	Rotate right	0Bnd	Rd := Rd ROR by Rn	ncz	ADC r1,r2
STB	Store register byte	24nd	[Rn] := Rd_{byte}	---	STB r1,[r7]
STP	Stop	E000	Set internal Stop flag	---	STP
STR	Store register	20nd	[Rn] := Rd	---	STR r2,[r7]
SUB	Subtract Subtract immediate	02nd 7iid	Rd := Rd - Rn	ncz	SUB r1,r2 SUB r1,#3A
SXB	Sign extend byte	03nd	Rd := (signed)Rn_{byte}	n-z	SXB r7,r6
TEQ	Test equivalence	09nd	Rd EOR Rn	n-z	ADC r1,r2
TST	Test bits	08nd	Rd AND Rn	n-z	ADC r1,r2

Notes

`ooo` - offset – 12-bit absolute memory address
`oo` - 8-bit relative address
`rr` - low register list (0-7)
`hh` - high register list (8-15)
`n` - Rn register number
`d` - Rd register number (destination)
`ii` - 8 bit immediate value
`x` - Condition Code (see Table 3)

Table 8.4 – Instruction Set Coding Sheet

9.CODE

NEXT PAGE