MPROC Manual and Architecture Description

Christian Deussen

2013

Abstract

MPROC is a 16 bit CPU built out of TTL logic chips. The 74HCTxxx logic series is used.

Contents

1 Overview

1.1 Machine word

- always one nibble of opcode bits and the second nibble includes which registers are involved
- the next byte can be an immediate value.
- data-bus is 8 bits wide
- address-bus is 15 bits wide: 0x0000 to 0x7FFF is the ram, 0x8000 to 0xFFFF is the flash!
- thus we have 32kb flash and 32kb ram. We can execute both from ram and flash
- To adress something larger than 8-bit addresses, the BankRegister(BR) is used for the bits 7..15. Affected instructions: STR and LDA
- PC is 16 bits wide: P_L:0...7; PC_H:8...15
- ALU is 8 bits wide

1.2 Stack

- 32kb Stack FIFO. The stack memory can only be addressed via PUSH and POP. No stack pointer arithmetic is possible because the stack is not connected to the address-bus. Thus we have a total of 32kb Ram +32KB Stack = 64kb RAM and 32kb ROM.
- The Stack pointer is implemented using 74xx193 binary counter chips.

1.3 I/O

- I/O can be written to with the pop/ldr and read from with the push/str instructions. There are 2 Output ports and one input port. Always 8 bits wide.

1.4 Memory

- Ram(w24129ak12) and Flash(W29EE011) is connected to the same address Bus. The memory devices have both three-states outputs. The bank register(BR) controls with the highest bit(bit14 on the address bus) whether RAM or the flash will be targeted. Writing to the flash will not work since a whole page(256bits) needs to be written at once. Maybe I can upgrade this later with a more decent flash chip, which will allow easy writing just like the ram.

1.5 JMP(C/Z)-instructions

- Since the jmp instructions do not support offset but only immediate operands, the compiler needs to make sure that if a target of a jmp is not in the current 256byte block, a FAR_JMP with both high and low address are emmitted.

- If the JMP has only one argument, it is used as an offset instead as an address. Encoded with operand 0x00: reg1, number.

2 Overview

2.1 Machine word

- always one nibble of opcode bits and the second nibble includes which registers are involved
- the next byte can be an immediate value.
- data-bus is 8 bits wide
- address-bus is 15 bits wide: 0x0000 to 0x7FFF is the ram, 0x8000 to 0xFFFF is the flash!
- thus we have 32kb flash and 32kb ram. We can execute both from ram and flash
- To adress something larger than 8-bit addresses, the BankRegister(BR) is used for the bits 7..15. Affected instructions: STR and LDA
- PC is 16 bits wide: P_L:0...7; PC_H:8...15
- ALU is 8 bits wide

2.2 Stack

- 32kb Stack FIFO. The stack memory can only be addressed via PUSH and POP. No stack pointer arithmetic is possible because the stack is not connected to the address-bus. Thus we have a total of 32kb Ram +32KB Stack = 64kb RAM and 32kb ROM.
- The Stack pointer is implemented using 74xx193 binary counter chips.

2.3 I/O

- I/O can be written to with the pop/ldr and read from with the push/str instructions. There are 2 Output ports and one input port. Always 8 bits wide.

2.4 Memory

- Ram(w24129ak12) and Flash(W29EE011) is connected to the same address Bus. The memory devices have both three-states outputs. The bank register(BR) controls with the highest bit(bit14 on the address bus) whether RAM or the flash will be targeted. Writing to the flash will not work since a whole page(256bits) needs to be written at once. Maybe I can upgrade this later with a more decent flash chip, which will allow easy writing just like the ram.

2.5 JMP(C/Z)-instructions

- Since the jmp instructions do not support offset but only immediate operands, the compiler needs to make sure that if a target of a jmp is not in the current 256byte block, a FAR_JMP with both high and low address are emmitted.

- If the JMP has only one argument, it is used as an offset instead as an address. Encoded with operand 0x00: reg1, number.

3 Instructions

3.1 Instruction Set

Low Nibble	Instruction	Note	Decode Table
0x0	ADD reg1, reg2	reg1 + reg2. Result in reg1	A
0x1	SUB reg1, reg2	reg1 - reg2. Result in reg1	C
0x2	NOR reg1, reg2	reg1 = !(reg1 OR reg2)	C
0x3	AND reg1, reg2	reg1 = reg1 AND reg2	A
0x4	MOV reg1, reg2	reg1 = reg2	A
0x5	MOVZ reg1, reg2	MOV if reg1 is zero	A
0x6	JMP reg1, reg2	set PC_L to reg1, PC_H to reg2	A
0x7	JMPZ reg1, reg2	JMP if reg1 is zero	A
0x8	JMPC reg1, reg2	JMP if carry is set	A
0x9	STR reg1	Store reg1 where BR points to	A
0xA	LDR reg1	Load into reg1 where BR points to	A
0xB	SET_BR reg1, reg2	Set BR_L to reg1, BR_H to reg2	A
0xC	Unused		
0xD	COUNT_BR UP/DOWN	increase(UP) or decrease(down) BR by one	A
0xE	PUSH reg1	Push reg1 to the stack	A
0xF	POP reg1	Pop stack item into reg1	A

3.2 Argument Decode Table A

The second instruction Nibble defines the operand registers. Table A is used by MOV, ALU, SET_BR and ALU

Opcode Nibble 2	Involved Registers	Note
0x0	reg1, number	JMP(Z/C) uses this operand byte to encode JMP(Z/C) pc_high, number thus JMP(Z/C) reg1, number cant be used.
0x1	reg2, number	
0x2	reg3, number	
0x3	reg4, number	
0x4	reg1, reg2	
0x5	reg1, reg3	
0x6	reg1, reg4	
0x7	reg1, reg1	
0x8	reg2, reg3	
0x9	reg2, reg4	
0xA	reg2, reg1	
0xB	reg3, reg2	
0xC	reg3, reg4	
0xD	reg3, reg1	
0xE	reg4, reg2	
0xF	reg4, reg3	

3.3 Argument Input Table B

Used by LDA and POP

Opcode Nibble 2	Involved Registers
0x0	reg1
0x1	reg2
0x2	reg3
0x3	reg4
0x4	BR_LOW
0x5	BR_HIGH
0x6	IO_OUTPUT_REGISTER_1
0x7	IO_OUTPUT_REGISTER_2

3.4 Argument Output Table C

Used by STR and PUSH

Opcode Nibble 2	Involved Registers
0x0	reg1
0x1	reg2
0x2	reg3
0x3	reg4
0x4	number
0x5	BR_LOW
0x6	BR_HIGH
0x7	IO_INPUT_REGISTER_1

4 Calling Convention

- Caller saves his working registers(For example on the stack)
- Arguments are passed in Reg1, Reg2, Reg3. Additional arguments are passed via the stack(Reg4 cannot be used since it is needed for the call)
- Return Address is passed through the stack. First the Low byte is pushed, than the High byte
- Return values are in Reg1 and Reg2. Additional return values are on the stack (Reg3 and 4 cannot be used since they are needed for the return JMP instruction)

5 Hardware Implementation

The 74HCT logic family is used. Write here the propagation delay calculation, Power consumption, and performance charachteristics. PC and SP are implemented using 74xx193 binary counters. With that no 16 bit addition needs to be done.

5.1 Accessing Memory

Step	Read	Write
1	Apply address to A-bus, clear Not_OE and NOT_CS	Apply address to A-bus, Apply data to D-bus, clear Not_OE and NOT_CS
2	wait > 6ns	wait > 6ns
3	Read Data from Databus/Write read data back to registers	Set NOT_CS
4	Set NOT_CS	

5.2 Fetch and Decode

Step	Fetch and Decode
1	Connect PC to Mem Addr, IR to DBus
2	Do Mem Read and IR write Signals
3	Connect PC and 1(through multiplexer ctrl)to ALU, ADD ctrl
4	Reg write back

5.3 Execution Steps

Command	1	2	3	4	5
ALU	Write Reg A to DBUS. High to low triggered	Latch Reg A from DBUS for the ALU. Happens at the High to low transition.	Write Reg B to DBUS	At the High to low transition of the state signal the ALU output is latched	Fill Reg
MOV(Z)	Write Reg to DBUS	Fill Regs			
LDA					
STR					
JMP(C/Z)	Write Reg A to DBUS	Full PC_LOW with reg. if Reg1 is 0x00 done	Apply Reg B to DBUS. This is done using the multiplexer to use an input selector as an ouput selector	Fill PC_HIGH with Reg B	
COUNT_BR	Increment/Decrement BR				
PUSH	Decrement SP				
POP			Increment SP		
SET_BR	Write Reg to DBUS	Fill PC_LOW with reg	Apply the Table 2 multiplexer that decode table 2 is used	Fill BR with Reg 2	

6 Current Problems

6.1 ALU

ALU carry_out signal is changed by NOR and AND instructions. Need a flip flop or something similar. Do we need a flip flop for REG1_ZERO_MOVZ if reg1 is changed during a MOVZ instruction(Which could in turn have an effect on the STATE_SIGNAL of the MOV instruction)?

6.2 Startup

How to initialize the registers when power is turned on? RING_CNTR_CLR needs to go high. This clears the Ring Counter. Is this signal low active? CLR_ALL_REGS_NOT needs to go low for a short period and then for the rest of the time HIGH

6.3 Memory Access

Read: Can we apply the address, clear_not_oe and clear_not_cs in one instruction and fill the registers in the following instruction? 2 cycle memory access would be possible with this!