μ–EP–1
a simple 32-bit architecture

B. Ulmann
ULMANN@MZDMZA.ZDV.UNI-MAINZ.DE

October 13, 1995

Abstract

This article describes a very simple but quite powerful 32-bit architecture ideally suited for experimental and educational purposes in computer architecture design. It employs a very orthogonal instruction set in conjunction with 16 general purpose registers and a simple interrupt capability. It was developed at the University of Mainz, Germany and a prototype using standard TTL-circuits is being developed – large parts of the microcode are written and simple assembler and an emulator written in FORTRAN already exists.

1 Introduction

One of the main design topics for μ–EP–1 *(microprogrammed experimental processor)* was its simple realisation using techniques like TTL-circuits etc., so it could be successfully employed for educational purposes in computer science classes. Another very important point is the very orthogonal instructionset – it may be called a RISC-architecture in many respects. All instructions are of the same structure – there are no exceptions in coding operands, address modes and so on, which results in a very simple instruction decoder design. There are no mechanisms for memory management or floating point (but it should not be impossible to build appropriate extensions).

2 General Purpose Registers

There are 16 general purpose registers in μ–EP–1 named *R0...R15*. The first thirteen of these registers are of real general purpose – they have no special meaning to the operation of the processor. *R15* is used as the program counter *PC*, *R14* resembles the status register *SR* and *R13* is used in processing interrupts. When an interrupt occurs it is loaded with the appropriate return address which can be pushed onto a stack by the interrupt routine itself.

X	C	Z	N	I	Interrupt-address	
31	30	29	28	27	26	0

Figure 1: The status register – *R14*

The four most significant bits (31...28) of *SR* are used as condition flags. They are: comparison bit *X*, carry bit *C*, zero flag *Z* and negative flag *N*. These bits are modified by ALU-operations (the carry bit handling is a little more complex – see section 3.1 of this article) and SHIFT-operations. They can be used for programflow-control etc. by utilizing the condition-code-field of the instructions. Bit 27 of the *SR* is the interrupt-enable bit. If it is set, interrupts are enabled, else disabled. Bits 26...0 of this register are used for the interrupt-address – this is the biggest exception in the whole design because the address of an interrupt routine is only 27 bits in length, so it cannot use the entire range of memory addresses. Figure 1 depicts the structure of the status register *SR*.

3 Instructions

All instructions (not using constants) are 32 bits wide and of the structure shown in figure 2. Bits 31...28

SEL	~	/	OPC	M_3	OP_3	M_2	OP_2	M_1	OP_1
31 30	29	28	27 21	20 18	17 14	13 11	10 7	6 4	3 0

Figure 2: General instruction format

Bit 2	Bit 1	Bit 0 cleared	Bit 0 set
0	0	Rxx	@Rxx
0	1	--Rxx	@--Rxx
1	0	Rxx++	@Rxx++
1	1	const	@const

Figure 3: Addressing modes of operands

are used to specify the condition under which the instruction is executed. If bit 29 (the ~–bit) is cleared the instruction will always be executed. If it is set the instruction will only be executed if the condition flag of *SR*, selected by the value stored in bits 31 and 30 of the condition field of the instruction, is set. This implies that bit 28 was cleared – if it is set it inverts the selected flag bit before using it in this process.

Each instruction can use up to three operands coded in bits 0...20. Every operand consists of a 4 bit operand-field which can select any of the 16 GPRs and a 3 bit mode-field to store the desired addressing mode. Figure 3[1] shows the possible modes. If the use of a constant is coded in the mode-field of the operand, the value of the operand-field is neglected and the constant is loaded from the next storage location following the instruction. The program counter is then incremented.

Bits 27...21 are used as some kind of opcode-field. This name however is not really appropriate for our design because this field is not fully decoded to determine the instruction. Instructions are grouped into ALU-, SHIFT- and the HALT-instructions. If bit 27 is cleared the instruction uses the ALU of μ–EP–1. If it is set, bit 26 is used to distinguish between SHIFT-instructions (bit 26 cleared) and HALT (set).

Instruction	logic-mode	mnemonic
0 0 0 0	!A	NOT
0 0 0 1	!(A|B)	NOR
0 0 1 0	!A &B	
0 0 1 1	0	ZERO
0 1 0 0	!(A& B)	NAND
0 1 0 1	!B	
0 1 1 0	A^ B	XOR
0 1 1 1	A& !B	
1 0 0 0	!A|B	
1 0 0 1	!(A^ B)	XNOR
1 0 1 0	B	
1 0 1 1	A& B	AND
1 1 0 0	1	ONE
1 1 0 1	A|!B	
1 1 1 0	A|B	OR
1 1 1 1	A	MOVE

Instruction	arithmetic-mode	mnemonic
0 0 0 0	A	
0 0 0 1	A|B	
0 0 1 0	A|!B	
0 0 1 1	- 1	
0 1 0 0	A+ (A & !B)	
0 1 0 1	(A|B) + (A & !B)	
0 1 1 0	A - B - 1	SUB
0 1 1 1	(A & !B) - 1	
1 0 0 0	A +(A& B)	
1 0 0 1	A + B	ADD
1 0 1 0	(A|!B)+ (A & B)	
1 0 1 1	(A & B) - 1	
1 1 0 0	A + A	2A
1 1 0 1	(A|B) + A	
1 1 1 0	(A|!B) + A	
1 1 1 1	A - 1	DEC

Figure 4: ALU-instructions and abbreviations

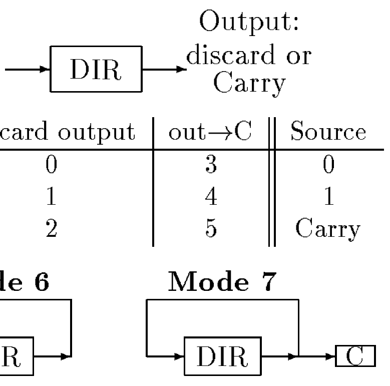
[1]The @-sign denotes that the argument is used as a pointer to the desired operand.

Bit	Val	Operation	Comparison
X	1	SUB cc	A = (B MINUS 1)
	1	XNOR	A = B
	1	XOR	A != B
C	0	SUB cc	A < B
	1	SUB cc	A >= B
	0	SUB cs	A <= B
	1	SUB cs	A > B

Figure 5: Compare instructions

3.1 ALU-instructions

The ALU of μ-EP-1 uses the well known *74 LS 181* chips together with *74 LS 182* carry-lookahead-generators. These chips implement 16 arithmetic and 16 logic instructions shown in figure 4[2]. To select one particular instruction out of these 32 possibilities, bits 25...21 of the *opcode*-field are used[3]. These bits are directly fed to the appropriate ALU-control lines. The remaining bit 26 is the carry-control-bit *CCB*. Only if it is set the actual instruction is allowed to modify the carry bit of the *SR*[4]. This is coded by *(C)* following the desired instruction, for example **ADD(C)** is allowed to alter the value of the *C*-bit while a simple **ADD** is not. The bits *N*, *Z* and *X* of the status register are affected by all ALU-instructions. The *X*-bit corresponds to the product of all 8 A=B-outputs of the 4-bit ALU-chips. It can be used to implement some compare-instructions shown in figure 5[5].



	discard output	out→C	Source
Mode	0	3	0
Mode	1	4	1
Mode	2	5	Carry

Figure 6: The 8 basic shift operations

3.2 SHIFT-instructions

The shifting unit is capable of performing 8 different shift-operations – each of these can be done in both shifting directions so there are 16 operations which can be coded in a shift-instruction using bits 24...21. Bit 21 denotes the desired direction – the remaining three bits are used to select one of the 8 basic modes shown in figure 6.

3.3 HALT

This instruction is detected by bit 27 and 26 set. It halts the entire processor and enters a microcode-loop which can be left on power-up or reset only or by flipping the start-switch on the console panel.

4 Basic programming concepts

As you will have noticed there are no special instructions for program-flow-control such as BRANCH, JUMP, TEST and other operations. They are not even necessary in this concept because all of them can be simply rebuilt by using ALU-instructions, e.g. MOVE, ADD etc. Because the execution of each instruction can be made dependend on the value of one of the 4 condition-flags of the status register *SR*, conditional jumps or branches are also implemented using ADD, MOVE etc. instructions. If you want to jump to address \$1234 you could simply write **MOVE \$1234, R15**. If this instruction is to be performed only if the carry-bit is set, you could code **~C MOVE \$1234, R15**. The sign ~ denotes that the

[2]As one can easily see there are ALU-instructions which do not need all of the three possible operands – some even produce only a result. So there has to be a mechanism to prevent unused operands from being fetched by the microcode. This is implemented using a 32 by 2 bit PROM where the 2 bits correspond to the 2 input operands of the instruction. It is addressed using the 5 ALU-control bits and returns a value of one if the operand has to be fetched and zero else. Such circuitry is not necessary for other types of instructions such as SHIFT and HALT since SHIFT always needs three operands and HALT needs none.

[3]Bit 25 distinguishes between arithmetic (cleared) and logic (set) mode.

[4]The used ALU-chip-set employs an active-low carry as both input- and output-carry. To simplify programming, the *C*-bit of the *SR* is inverted before fed into the ALU. The generated carry is also inverted before it is stored in the *SR*.

[5]cc denotes carry cleared, sc carry set.

condition-field of the instruction is to be used in the instruction. If the jump should only take place if the *C*-bit is cleared it must be written as **~/C MOVE $1234, R15**. Jumps relative to the actual value of the program counter can be implemented using the ADD-instruction and so on...

4.1 Programming example

The following little program is used to add two vectors giving a third one. The start-addresses of the two input vectors are stored in *R0* and *R1*, the start-address for the resultant vector is supplied in *R2*. The end-address of the first vector + 1 can be found in *R3*.

```
L1:     ADD @R0++, @R1++, @R2++
        XNOR R0, R3 ;SET X IF R0==R3
     ~/X MOVE #L1, R15 ; DO ... LOOP
        HALT
```

4.2 Stacks

Stacks can be implemented in an easy way because of the autoincrement and -decrement feature of μ–EP–1. There is no designated stackpointer – each register can be used for this purpose as the following instructions may show: In the following *R12* shall be used that way. To push a value onto this stack you could write **MOVE <source>, @--R12**. The counterpart to this instruction would be a pop which can be written as **MOVE @R12++, <destination>**. The lack of a designated stackpointer is the reason for using R13 as an interrupt-return-address-storage. The interrupt routine itself is responsible for pushing this address onto some stack-structure.

5 Interrupts

Interrupts are implemented in a very simple yet crude way in μ–EP–1. There are two kinds of interrupts: simple interrupts, which result in a jump to the location coded in the lower 27 bits of the status register. And vector interrupts where the desired jump-address is put onto the data-bus by the external device when the interrupt is accepted by the processor.

All interrupts are maskable and there is no priority-system for then. This has to be accomplished in software. Interrupts are disabled if the *I*-bit of the *SR* is cleared. If an interrupt occurs and the *I*-bit is set, the following steps take place:

First the *I*-bit of the *SR* is cleared so that other interrupts can not take place at this time. After this the interrupt is granted by the processor using a special control line on the bus. Then the actual value of *R15*, the program counter, is copied to *R13* to be used as a return address at the completion of the interrupt-routine. In the last step the *PC* is loaded with the address of the desired routine. This is either obtained from the lower 27 bits of the *SR* (if it is a non-vector-interrupt), or read from the data-lines of the bus, where it was placed by the interrupting device after receiving the grant-signal (vector-interrupt).

Then it is a matter of the routine itself to store the contents of registers, which will be modified during the run of this program part, and the content of *R13*. If this is done the *I*-flag can be set so that other interrupts can occur if desired. Finally the interrupt-routine restores all previously saved register-contents and then performs a jump to the appropriate return-address.

6 Realisation

The actual μ–EP–1 will be built around a single 32-bit wide internal data-bus which simplifies circuit design and construction. Figure 7 shows a block diagram of the actual implementation[6]. The console-panel enables the user to examine selected memory-addresses or registers and to deposit values there. The *misc*-box in figure 7 denotes some miscellaneous hardware such as a scratch register used for panel-operation, an up-down-counter for the autoincrement/-decrement feature, the clock-generator and some glue-logic for the microprogrammed control unit not shown in the picture.

The microprogrammed control unit employs a simple horizontal microword which directly controls the various system components. It has 64 condition-inputs (there is room left for future extensions) which can be used to control microprogramflow in a man-

[6]In this figure *RFILE* denotes the registerfile, containing the 16 GPRs, *IREG* is the instructionregister with decoding logic, *PIF* and *PNL* are panel-interface and the console-panel and *PBI* depicts the peripheralbus-interface.

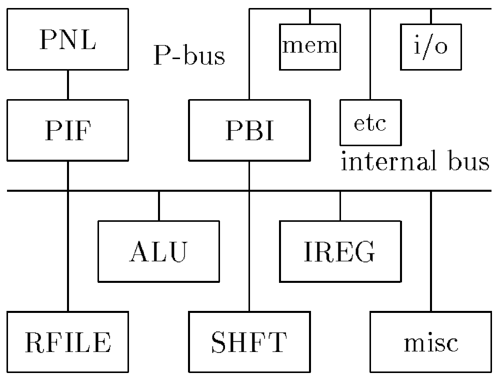


Figure 7: Implementation

ner quite similar to the μ–EP–1 assembler. Each microinstruction contains a condition field to select one of these 64 input lines. The first line is tied to logic 1 so that unconditional jumps are possible to allow simple sequential program flow. There are two address fields in each microword – the first contains the address of the next microinstruction, which has to be executed if the condition was false, the second the address for a true condition line. The address-multiplexer is directly controlled by the selected condition-line, its output is wired to the input of the microaddress-register.

As simulation runs have shown, it may be concluded that μ–EP–1 is a quite powerful architecture and easy to implement, so it seems to be really well suited for experimental and educational purposes in the field of computer architecture.