IP 08

Session : IP Design (2)

µBIP: A SIMPLIFIED MICROCONTROLLER ARCHITECTURE FOR EDUCATION IN EMBEDDED SYSTEMS DESIGN

Maicon Carlos Pereira, Univali
Cesar Albenes Zeferino, Univali

Itajaí, Brazil

Abstract. In this paper, it is presented a basic microcontroller, named µBIP, developed to be used in the teaching of embedded systems design. The microcontroller architecture was specified in order to ease its understanding by students and, also, the building of a physical implementation. The architectural specification was done by using ArchC, and a synthesizable IP core was implemented in VHDL and validated in FPGA.

1. INTRODUCTION

The demand for professionals able to deal with the design of digital systems for embedded systems has increased in the last years. Courses covering issues related with this topic lack of basic architectures which could easily be used to teach introductory concepts related to the design of processors and microcontrollers.

In this direction, a basic microcontroller architecture named µBIP (read *micro bip*) was proposed and implemented in order to be used in the teaching of concepts on the design of digital systems and microcontrollers. A first organization was specified and a soft core was modeled in VHDL. In the design flow, architectural specification was performed by using ArchC, an Architecture Description Language – ADL developed University of Campinas [1]. ArchC automatically generated the assembler, the linker and an ISA (Instruction-Set Architecture) Simulator. After that, µBIP was specified and a synthesizable soft core was modeled in VHDL and prototyped in FPGA for validation.

This paper presents µBIP architecture and organization and the issues related to its implementation and validation.

2. µBIP ARCHITECTURE

In µBIP microcontroller, instructions and data are 16-bit wide. There is only one instruction format (Fig. 1), which includes a 5-bit opcode and an 11-bit operand. µBIP is an accumulator-oriented architecture and ACC register works as an implicit operand for the most part of the instructions.

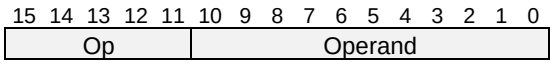

Fig. 1. µBIP instruction format.

µBIP architecture supports three addressing modes: immediate, direct and indirect. Depending on the addressing mode, the explicit operand of the instruction format has different meanings: a constant (in the immediate addressing mode); a variable (in the direct addressing mode); or a vector index (in the indirect addressing mode).

µBIP uses a Harvard architecture with separated memories for instruction and data. The addressing space is organized into two regions: a 2 Kword program space and a 2 Kword data space. This data space is divided into an 1 Kword space for data memory and an 1 Kword space for memory-mapped I/O (see Fig. 2). This region is used to address the registers of peripherals, like timers and I/O ports.

The CPU has only five registers: PC (Program Counter), ACC (Accumulator), STATUS, SP (Stack Pointer), and INDR (Index Register), used in array-based operations. STATUS registers has three flags: Z (Zero), N (Negative) and C (Carry).

Instruction-set includes only 29 instructions, which are described in Table 1.

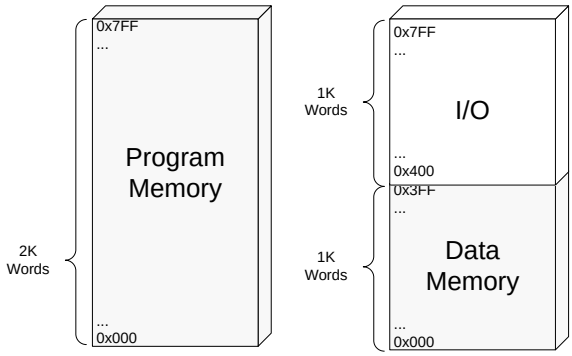

Fig. 2. Addressing spaces.

For procedure calls and interrupts, µBIP uses a hardware stack to save the current context, that is, the address of the next instruction (PC+1). This approach is based on the one used in Microchip's PIC16 architecture.

As one can see, µBIP architecture was specified to be "as simple as possible". Since it uses a single format for all the instructions of its small instruction set, students can easily learn how to program it. Also, its architecture was specified in order to make easier the design of its organization, as is shown in the following section.

Table 1. The µBIP Instruction-Set Architecture.

Opcode	Intruction	Affected Flags	PC updating and operation	
00000	**HLT**		PC ← PC	
00001	**STO** operand		PC ← PC + 1	DataMemory[operand] ← ACC
00010	**LD** operand		PC ← PC + 1	ACC ← DataMemory[operand]
00011	**LDI** operand		PC ← PC + 1	ACC ← operand
00100	**ADD** operand	Z,N,C	PC ← PC + 1	ACC ← ACC + DataMemory[operand]
00101	**ADDI** operand	Z,N,C	PC ← PC + 1	ACC ← ACC + operand
00110	**SUB** operand	Z,N,C	PC ← PC + 1	ACC ← ACC – DataMemory[operand]
00111	**SUBI** operand	Z,N,C	PC ← PC + 1	ACC ← ACC – operand
01000	**BEQ** operand		if (STATUS.Z=1) then PC ← operand else PC ← PC + 1	
01001	**BNE** operand		if (STATUS.Z=0) then PC ← operand else PC ← PC + 1	
01010	**BGT** operand		if (STATUS.Z=0) and (STATUS.N=0) then PC ← operand else PC ← PC + 1	
01011	**BGE** operand		if (STATUS.N=0) then PC ← operand else PC ← PC + 1	
01100	**BLT** operand		if (STATUS.N=1) then PC ← operand else PC ← PC + 1	
01101	**BLE** operand		if (STATUS.Z=1) or (STATUS.N=1) then PC ← operand else PC ← PC + 1	
01110	**JMP** operand		PC ← operand	
01111	**NOT**	Z,N	PC ← PC + 1	ACC ← NOT(ACC)
10000	**AND** operand	Z,N	PC ← PC + 1	ACC ← ACC AND DataMemory[operand]
10001	**ANDI** operand	Z,N	PC ← PC + 1	ACC ← ACC AND operand
10010	**OR** operand	Z,N	PC ← PC + 1	ACC ← ACC OR DataMemory[operand]
10011	**ORI** operand	Z,N	PC ← PC + 1	ACC ← ACC OR operand
10100	**XOR** operand	Z,N	PC ← PC + 1	ACC ← ACC XOR DataMemory[operand]
10101	**XORI** operand	Z,N	PC ← PC + 1	ACC ← ACC XOR operand
10110	**SLL** operand	Z,N	PC ← PC + 1	ACC ← ACC << operand
10111	**SRL** operand	Z,N	PC ← PC + 1	ACC ← ACC >> operand
11000	**STOV** operand		PC ← PC + 1	DataMemory[operand + INDR] ← ACC
11001	**LDV** operand		PC ← PC + 1	ACC ← DataMemory[operand + INDR]
11010	**RETURN**		PC ← Top of Stack	
11011	**RETINT**		PC ← Top of Stack	
11100	**CALL** operand		PC ← operand	Top of stack ← PC+1

3. µBIP ORGANIZATION

Fig. 3 depicts the µBIP organization. It is based on a Harvard monocycle approach, like the one used in the implementation of MIPS [2]. It includes the instruction and data memories, the CPU (composed by the Control Unit and the Datapath), and the peripherals.

The Control Unit fetches an instruction from the Program Memory, decodes it and sends command signals to the Datapath, which is responsible to process data. The current version of µBIP includes the following peripherals and hardware features: (*i*) two 16-bit I/O ports with individual direction control for each pin; (*ii*) one 16-bit timer; (*iii*) an interrupt controller; and (*iv*) hardware support for procedure calls.

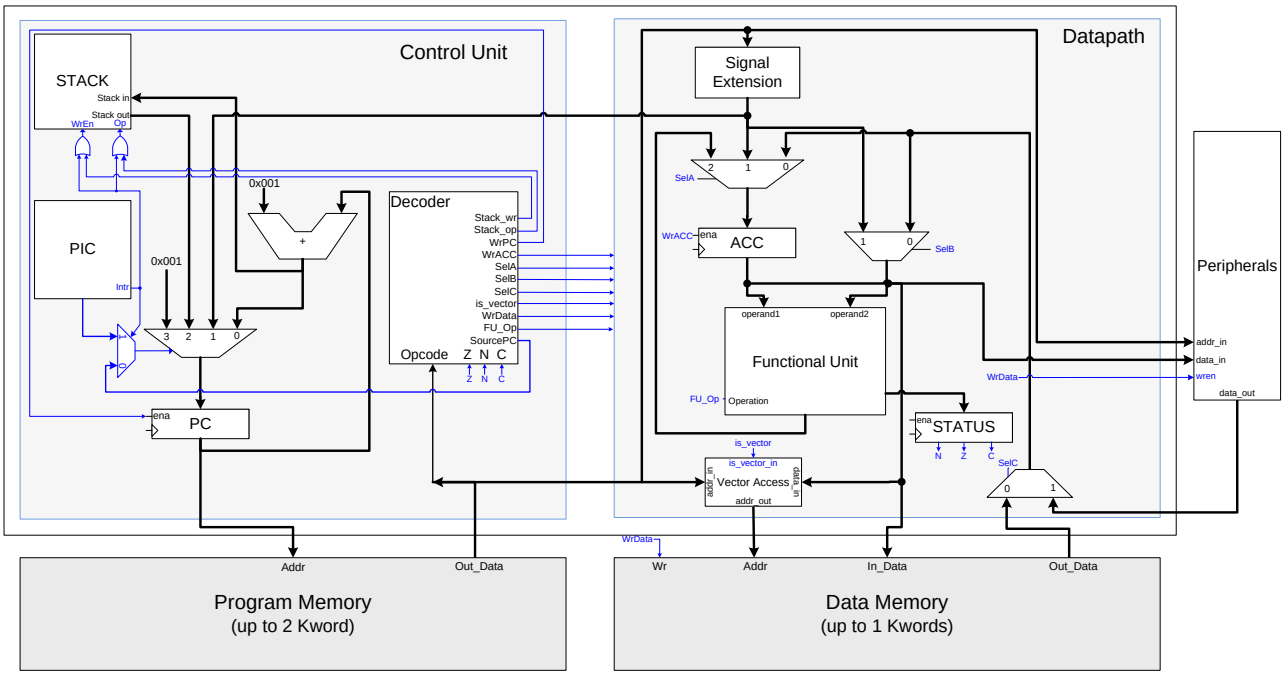

Fig. 3. µBIP organization.

3. IMPLEMENTATION AND RESULTS

In the design of µBIP, ArchC [1] was used in the architectural specification phase. Firstly an ArchC model of µBIP was described and ArchC automatically generated an assembler, a linker and an ISA simulator. By using this tools, a set of testbench codes were written to make the unitary test of each one of the instructions in the ISA. Furthermore, they were written test applications based on the benchmarks of Dalton Project [3] which covered 80% of the µBIP ISA.

The ArchC description of µBIP is composed by the following files: (*i*) *ubip.ac*; (*ii*) *ubip_isa.ac*; (*iii*) *ubip_isa.cpp*; and (*iv*) *ubip_address.h*. The first file describes the architecture resources: memories, registers, word size and endianness (see Fig. 4).

```
AC_ARCH(ubip){
  ac_mem memRAM:4K; //4Kbyte/2K-Words
  ac_mem memROM:4K; //4Kbyte/2K-Words
  ac_reg PC, ACC, STATUS;
  ac_wordsize 16;
  ARCH_CTOR(ubip) {
    ac_isa("ubip_isa.ac");
    set_endian("big");
  };
};
```

Fig. 4. *ubip.ac*: the architecture resources.

The second file (*ubip_isa.ac*), partially shown in Fig. 5, is where the instruction format and its fields (*op* and *operand*) are specified. Following, the instructions set is declared and it is defined a map (*ac_asm_map*) describing the allowed values for the instruction operands. Two functions are used to define the syntax (*set_asm*) and the opcode (*set_decoder*) of a given instruction. Another function (*pseudo_instr*) allows defining pseudo-instruction which, at the assembling process, is replaced by a sequence of native instructions.

The third file (*ubip_isa.cpp*) describes the behavior of instructions and peripherals. Fig. 6 and 7 present part of this file, where they are shown the behavior of ADD instruction (*behaviour*(*add*)) and part of the common behavior for all the instructions (*behavior*(*instruction*)). Two functions (*DataMemoryRead* and *DataMemoryWrite*) were implemented to deal with word addressing, since ArchC addressing is byte-oriented. The same problem had to be solved in the functions related with PC register (*inc_PC*, *set_PC* and *get_PC*). A number of constants used in *ubip_isa.cpp* are defined in the last of the four files which compose the ArchC µBIP model.

```
AC_ISA(ubip){
  ac_format INS_FORMAT  = "%op:5 %operand:11";

  ac_instr<INS_FORMAT> add, addi, sub, subi;

  ac_asm_map ubipSFR {
    "$"[0..2047]   = [0..2047];
    "$port0_dir"   = 1024;
    "$port0_data"  = 1025;
    "$tmr0_config" = 1040;
    ...
  }
  ISA_CTOR(ubip){
    //--- ASSEMBLY ---
    add.set_asm("add %ubipSFR", operand);
    add.set_asm("add %exp", operand);
    add.set_decoder(op=0x04);
    ...
    //Pseudo-Instructions
    pseudo_instr("psto %ubipSFR, %imm"){
      "ldi %1";
      "sto %0";
    }
    ...
  };
};
```

Fig. 5. *ubip_isa.ac*: the Instruction-Set Architecture.

```
void ac_behavior( add ){
  ac_word operand1 = ACC.read();                          //Fetching operand 1
  ac_word operand2 = DataMemoryRead(memRAM,operand);      //Fetching operand 1
  ac_word result   = operand1 + operand2;                 //Executing the operation
  ACC.write(result);                                      //Writtng result into ACC
  set_status(STATUS, Z, check_zero(result));              //Updating STATUS flags
  set_status(STATUS, N, check_negative(result));
  set_status(STATUS, C, check_carry_out(operand1, operand2));
  inc_PC(ac_pc);                                          //Updating PC
}
```

Fig. 6. *ubip_isa.cpp*: the behavior of ADD instruction.

```
// For all the instruction do:
void ac_behavior( instruction ){
   //Check for interrupt requests on PORT0 and on Timer0
   bool bIntr = false;
   bIntr = bIntr | CheckInterruptPA0(memRAM, ac_pc, tempMem[PORT0_DATA],
                              DataMemoryRead(memRAM,PORT0_DATA));
   bIntr = bIntr | (trm00_inc(memRAM, ac_pc, ac_instr_counter)); //Update Timer0
   if (bIntr) ac_annul();
   (...)
}
```

Fig. 7. *ubip_isa.cpp*: the common behavior for all the instructions.

µBIP organization was described in VHDL and synthesized in Altera's EPF10K70RC240-4 FPGA. Due to the monocycle organization and the ability of the VHDL compiler in optimizing the circuit for the instruction subset used by the program, silicon costs varies for each application. Table 2 summarizes the ISA coverage of some of the Dalton Project applications, the silicon costs and the maximum operating frequency for the target device.

Fig. 8 illustrates a simulation of the execution of cast application, which takes a 16-bit word and separates it into two 8-bit words. Firstly, the two I/O ports are configured as output ports (1 and 2). After, a 16 bit word, 0x1234, is built in ACC register (3). In (4), the less significant byte (0x34) is written into *port0_data*. Finally, ACC register is shifted right (5), and the most significant byte is written into *port1_data* (6).

Table 2. Test coverage and silicon cost for Dalton Project applications

Application	Program size (instructions)	ISA test coverage	#Logic Cells	*fclk* (MHz)
cast	9	20%	492	7.44
fib	50	48%	992	6.95
gcd	22	34%	805	6.75
sort	97	44%	1050	6.58
sqroot	59	44%	996	6.51

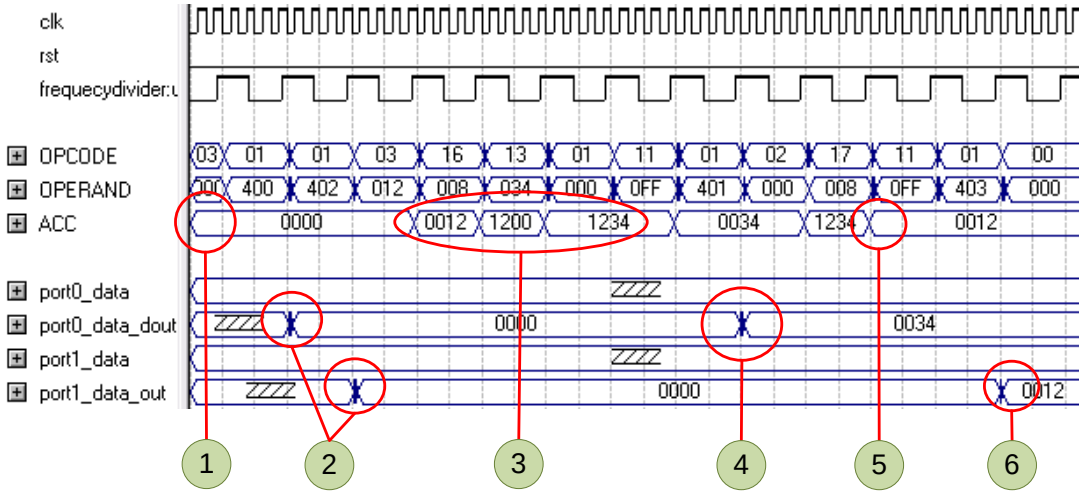

Fig. 8. Simulation of *cast* application.

The validation on FPGA was done by running the same applications on Altera UP-2 developing board. The performed experiments allowed the verification of the correctness of the developed core.

4. CONCLUSIONS

This work described the design of a basic microcontroller intended to be used in the teaching of introductory course on digital systems design. They were briefly described its architecture and organization, and design methodology used to specify, implement and validate it.

Ongoing and future works includes the implementation of new peripherals for interconnection with standard serial buses and the building of a visual IDE for the development tools.

REFERENCES

[1] The ArchC Team. The ArchC architecture description language v2.0. Campinas: The ArchC Team, 2007.

[2] Patterson, D. A., Hennessy, J. L. Computer Organization and Design. San Francisco: Morgan Kaufmann, 1998.

[3] The Dalton Project Team. The UCR Dalton Project. University of California – Riverside, 2001.