AVR Enhanced RISC Microcontrollers

Alf-Egil Bogen
Vegard Wollan
ATMEL Corporation
ATMEL Development Center, Trondheim, Norway

High level languages (HLLs) are rapidly becoming the standard programming methodology for embedded microcontrollers (MCUs), even for smaller 8-bit devices. The C language is probably the most widely used HLL in MCUs, but will in most applications give an increased code size compared to assembly programming. ATMEL identified the need of an architecture developed specially for the C language in order to reduce this overhead to a minimum. The result is the ATMEL AVR MCU, that in addition to the optimized code size, is a true single cycle RISC (Reduced Instruction Set Computer) machine with 32 general purpose registers (accumulators) running 4-12 times faster than currently used MCUs.

1. Introduction

The initial AVR product offering is three 8-bit base-line devices with enhanced 16-bit hardware support. Atmel's low-power non-volatile memory technology is used for program code and data. The on-chip program Flash and data EEPROM are in-system programmable. The three first AVR MCUs have 1K, 2K and 8K bytes program Flash organized as 16-bit wide instruction words.

The Atmel AVR Enhanced RISC Microcontrollers offer an architecture concept for high performance and low-power consumption simultaneously. A full range of AVR MCUs - from base-line to top end - feature a RISC architecture and instruction set optimized for efficient code density with built-in support for high-level languages.

Please refer to [1] for more details.

2. Enhanced RISC

Many existing RISC architectures require larger code size to perform a given task with the traditional CISC (Complex Instruction Set Computer) architectures. RISC MCU's are often chosen where a high speed is needed. The reduced instruction set will be fast, but reduced in complexity.

The *AVR* is designed to be a RISC MCU with a larger number of instructions to reduce the code size and to increase the speed further. Ciscy-like instructions are introduced without letting the RISC performance and low power consumption features suffer. This first major enhancement was made after thorough analysis of several architectures and large amounts of application code. Still, the regular *AVR* RISC architecture enables cost effective implementations.

The second enhancement is achieved by tuning the architecture for optimizing code generation for the C language. This was done with a large application oriented benchmark suite, where the code was pseudo-compiled for the different enhancement alternatives in the architecture. Special tuning of the different addressing modes was important, "need to have" instead of "nice to have".

Many MCU architectures have only a small number of general registers or working registers (accumulators) - typically 1-8 registers. This is a major drawback for the C compiler design, where a lot of data moving is necessary. The *AVR* has 32 general-purpose working registers, that the C compiler fully utilizes to achieve the highest code density.

3. True Single Cycle Instructions

With true single cycle instructions the internal clock is identical to the oscillator clock. There is no internal divider to produce the different clock phases!

Most of the micros in the 8 - 16-bit market are dividing the clock with a ratio of 1:4 to 1:12, which is a bottleneck for the speed. For a given task the *AVR* will run 4 to 12 times faster, or the power consumption can be reduced by a factor 4-12 with the same clock frequency. In a CMOS technology, the power consumption of digital logic is proportional to the frequency. Figure 1 shows the extreme increase of MIPS (Million Instructions Per Second) with true single cycle (1:1 ratio) compared to a clock division ratio of 1:4 and 1:12.

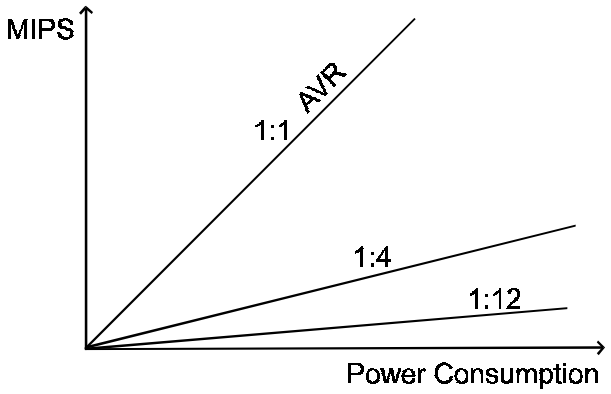

Figure 1: MIPS/Power Consumption

4. Designed for the C language

The C language is the most used HLL in the world today for MCUs. Since most MCU architectures are developed with assembly programming in mind the support for typical C instructions are poor. ATMEL's goal was to develop an architecture that was efficient both for the C language and assembly. With many C experts from C compiler suppliers in the design team, a very code efficient 8-bit micro with 16-bit support is developed.

When programming in C, one general rule is to use variables defined within a routine (local), instead of using global variables known in the whole program. Local variables will only allocate RAM memory when executing the specific routine while global variables will occupy RAM all the time. To handle local variables fast and code efficient, a lot of general-purpose registers are needed. The *AVR* has 32 general-purpose registers, all of them in the Arithmetic Logic Unit (ALU) path allowing true single cycle instructions. Figure 2 shows how efficient many registers can be compared to traditional CISC architectures with one accumulator.

Function: A = ((A .and. 84h) + (B.eor.C).or.80h

AVR code	CISC code
`EOR B,C`	`MOV ACC,C`
`ANDI A,#84h`	`EOR ACC,B`
`ADD A,B`	`MOV TMP,ACC`
`ORI B,#80h`	`MOV ACC,A`
	`AND ACC,#84h`
	`ADD ACC,TMP`
	`OR ACC,#80h`
	`MOV A,ACC`
`8 bytes`	`12-16 bytes`
`4 clocks`	`48-96 clocks`

Figure 2: Efficient code in the AVR

Three pairs of the 32 registers can be used as 16-bit pointers allowing indirect jumps and calls as well as many data memory accessing modes directly related to the C language. In addition to that the traditional stack for return address is available through the instruction set. Most MCU architectures have only 1-2 accumulators and 1-2 pointers.

Since pointers are very frequently used in C, the operations on the pointers are very important due to speed and code size. The *AVR* has addressing modes that directly pre-increments or post-decrements the different pointers when used to access data memory.

In addition to that, table lookup or stack operations can effectively be performed by using displacement (relative to the current pointer value) as shown in Figure 3. The displacement range is 0-64 bytes and detailed analysis shows that the range is sufficient for

most lookups in structures and tables. One very important note is that this fits into a single word instruction!

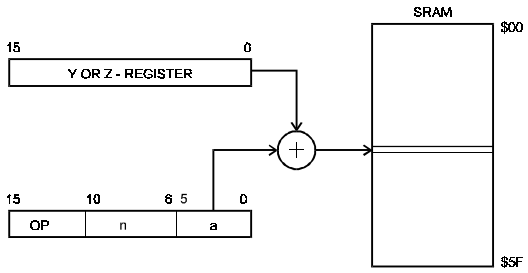

Figure 3: SRAM Direct with Displacement

To be able to change the pointers more than by using the pre-decrement or post-increment modes addition and subtraction between a 16-bit pointer and a constant are implemented in one cycle and a single word. The three pointers can also be used as eight 8-bit or three 16-bit general purpose working registers.

In general it is important to use 8-bit numbers in an 8-bit MCU since all data memories are 8 bit wide. However, when using C, larger number like integers (16 bit), long (32 bit) and float (32 bits floating point) are frequently used. Traditionally, computing on such numbers generates very large code, but since the *AVR* is designed to handle it, the *AVR* code generated is extremely small.

Example 1 shows an example where a small part of a routine written in C is compiled to see how it is translated to assembly code.

Example 1:

```
void routine(void)
     {
     long n1, n2;
     int n3;
     ...
     if (n1 != n2) n3 +=5;
     ...
     }
```

n1 and n2 are both 32-bits numbers that require 4 bytes each. n3 is an integer that requires two bytes. All three variables are defined as local variables within the routine.

This program example will in *AVR* assembly code be extremely compact. n1 = R3-R0, n2 = R7-R4 and n3 = R17:R16.:

```
...
CP   R0,R4      ; n1-n2 (byte 0)
CPC  R1,R5      ; n1-n2-C (byte 1)
CPC  R2,R6      ; n1-n2-C (byte 2)
CPC  R3,R7      ; n1-n2-C (byte 3)
BREQ EQUAL      ; Branch if equal
SUBI R16,LOW(0xfffb) ;n3+5 low byte
SBCI R17,HIGH(0xfffb);n3+C high byte
EQUAL:
...
```

In most MCU's the comparison from example 1 needs many more instructions since they miss Compare with Carry (CPC) and zero flag propagation. Zero flag propagation has a similar function as Carry propagation, to adjust the higher bytes in a number. The Zero propagation is implemented on compare and subtract instructions. A Compare instruction is generally a subtraction without storing of the result. The CPC is added to support comparisons of larger numbers than 8 bit.

Example 1 does also show how a Subtract Immediate (SUBI) and a Subtract Immediate with Carry (SBCI) can be used instead of addition (ADDI and ADCI) to compute n3 + 5 by subtracting the 2's complement of 5. Since ADDI and SUBI are complementary instructions only one pair is implemented to leave decoding space for other important instructions.

All the discussed features are implemented as a result of code benchmark and input from C compiler experts. The result is a very fast microcontroller with RISC performance and CISC code density.

[1] "*AVR* Enhanced RISC microcontroller", data book, May 1996. Atmel Corporation