WHITE PAPER

Software based Finite State Machine (FSM) with general purpose processors

White paper

Joseph Yiu January 2013

Overview

Finite state machines (FSM) are commonly used in electronic designs. FSM can be used in many applications such as digital signal processing, general data processing, control applications, communications, sensors and so on. It can be developed in many different ways, for example, using CPLDs, FPGAs, ASICs, microcontrollers, or in software in a computer. Traditionally, CPLD are commonly used for small standalone FSM designs, and larger FSMs are developed in ASICs or FPGA. Nowadays, most of the CPLD designs are replaced by microcontrollers because:

- Performance of microcontrollers are getting much better
- Cost of high performance microcontrollers are becoming much lower

Nowadays you can get a Cortex-M processor series-based microcontroller running at over 100MHz for just a few dollars, even cheaper than many CPLD products. They are easier to use, have many peripherals built in and low power.

The same changes can also be found in complex SoC designs, where some of the complex FSMs are being replaced by a processor subsystems. The usage of such software-based FSM includes power management in complex SoC, control of hardware interface or radio frequency circuitries in communication systems, boot sequence management, etc. In this paper, we will investigate the reasons behind using of Cortex-M series processors as FSM replacements, and various software / programming techniques which can help designers in different situations.

Why use software based FSM

There are many advantages for using software FSM:

1. Flexibility

Replacing a hardware FSM with a software FSM reduces the risk of the project by allowing the FSM design to be changed at anytime of the project cycle. If the FSM is designed as hardware, the behavior cannot be changed after the silicon is produced. If any bug is found, the silicon might have to be redesigned. Software based FSM allows the behavior of the FSM to be changed, and can be fine tuned even while the product is in field testing.

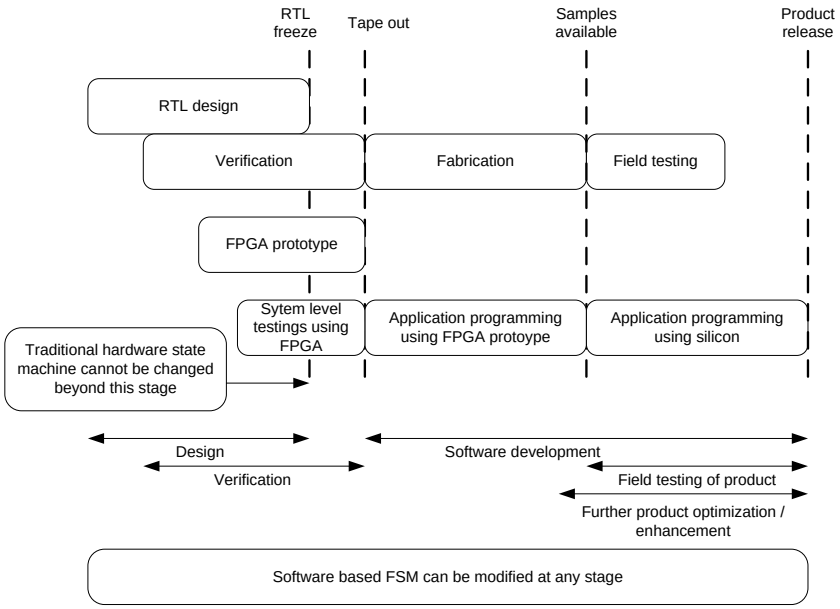

Figure 1: Software FSM allow modification of the FSM behaviors in different stages of the design cycle

In advanced SoC designs, some of the FSMs might also need to be interacting with application software running on application processors. In such applications, sometimes the details of the FSM operation cannot be finalized until a very late stage of the product development flow. As a result, it can be impractical to use hardware FSM for the design.

2. Debug methodology

Software FSM design can be debugged using debug solution for microcontroller software development. For example, you can halt, single step or set breakpoint to the processor, which you cannot do with traditional FSM designs. You can also use a debugger to examine/modify the status of the FSM, memory contents and input/output interface with debugger. Such debug operations and visibility of the system status allows easy debug of the FSM design, and provides much better chance for designer to optimize the FSM behavior.

3. Combine software control capability to FSM design

By using software FSM design, the handling of complex sequential operations become much easier. For example, looping can be handled easily, and repetitive tasks can be implemented as function/subroutine calls. If needed, assembly code can be used to optimize part of the FSM operations, and the rest of the FSM programmed in C for easier development.

You can also connect standard peripherals for microcontrollers to the processor in your FSM design, so that the design can handle additional I/O operations which can be difficult to handled using hardware FSM, or even use the FSM design as a general microcontroller system when the application does not need the FSM operation.

Potentially, multiple program images can be stored on the system and the design can selectively load a suitable program image to the memory during runtime. This allows a device to select between multiple FSM behaviors at runtime. In addition, you can even download a program into the SRAM and execute from there for maximum access speed. Running code from SRAM can also be very useful for BIST and field testing.

How about disadvantages?

So far it all sounds good, then how come this was not done in the past?

In older products, often the FSMs are much simpler. As a result the gate count of a processor based system was often much larger than that of a hardware FSM. Recent years FSMs are getting more and more complex, and the gate count getting close to or even larger than that of a processor based system. For example, in communication systems, the hardware FSM for the control logic might involve large number of counters and data registers. By moving to software FSM, the counters can be replaced with fewer generic counter peripherals and be shared under software control. At the same time, data values can be stored in SRAM instead of hardware registers. As a result, the total silicon area of a processor based system can be smaller than a hardware FSM, and yet easier to program and test.

In addition, the power consumption of the software based FSM can be reduced by utilizing low power features in the processor under software control. As a result, a software FSM can be more energy efficient than a hardware FSM.

There are limitations of course. The reaction time of pure software based FSM is limited by the processor instruction execution speed and the interface cycle behaviors. In some cases silicon designers use a processor based system combined with some small hardware accelerator blocks in the interface to achieve rapid response time (in the range of just 1 clock cycle or couple of clock cycles).

Why Cortex-M processors are useful as FSM replacement?

Since the release of the Cortex-M3 processor, the first product in the Cortex-M family, we started to see the use of the Cortex-M series processors as FSM in complex SoC such as high-end application processors used in smart phones and tablet computers. There are many reasons for using a Cortex-M series processor for these applications:

Small area - The Cortex-M processors are designed for low power applications and typically have quite low gate count. For example, the Cortex-M0 processor and the recently announced Cortex-M0+ processors have only 12K gates in minimum configurations. The excellent energy efficiency and the small silicon area make these processors very attractive for FSM replacement uses.

Code density – The Thumb-2 technology used in the Cortex-M architecture provides best in class code density. This allows the FSM program code to be stored in smaller memory. This is important for a number of FSM replacement scenarios where the FSM systems are operating in high clock frequency (several hundred MHz) and the program images are first copied into SRAM for faster accesses.

Low latency and deterministic – All the Cortex-M processors have very good deterministic behavior in interrupt handling. The Cortex-M3 and Cortex-M4 processors have an interrupt latency of just 12 clock cycles. This interrupt latency includes time required to push a number of registers to the stack. The Cortex-M0 and the Cortex-M0+ processors take a bit longer (16 and 15 cycles respectively), but these two processors have a feature to ensure zero jitter in interrupt behavior. And in the case of handling multiple interrupt requests, the Nested Vector Interrupt Controller (NVIC)interrupt controller in the Cortex-M processors automatically handles interrupt prioritization and nesting, and inter interrupt delay of the Cortex-M3 and Cortex-M4 processors are as low as 6 cycles.

Scalability – the Cortex-M processors are designed for scalability. Using the CoreSight debug architecture, multiple processors can share a single debug connection and a single trace connection. With most of the application processors used in high-end SoC being CoreSight compliant, the debug interface of the Cortex-M processors used in the FSM replacement can be linked to the debug system of other processors in the chip. The AMBA bus architecture also allows some of the system's memories and peripherals to be shared between the Cortex-M

processors and the application processors. This allows some of the low level I/O processing tasks to be off loaded to the Cortex-M processors if needed.

High performance – the Cortex-M processor series is tailored to fit different application requirements. For simple I/O control which is common for FSM replacement situations, the Cortex-M0 and the Cortex-M0+ processors can both handle the job. For more complex data processing, the Cortex-M3 or Cortex-M4 processors might be needed.

The recently announced Cortex-M0+ processor make it an even more attractive solution for FSM replacement.

1. First of all, by compressing the processor design to a two stage pipeline, the conditional branch instruction only need two clock cycles, allowing a FSM state transition to happen quicker.

2. Second, the single cycle I/O port feature accelerates the reading of data from inputs during decision making process. It also allows faster output response time.

3. Finally, the Cortex-M0+ provide even better energy efficiency. For the same design configuration, the dynamic power of the Cortex-M0+ is 20% lower than that of the Cortex-M0 processor. And by reducing the number of pipeline stages, the branch shadow is reduced and thus further enhances the system level power consumption.

The Cortex-M0+ processor uses the same instruction set as the Cortex-M0 processor, and therefore you can reuse your software including middleware, development tools and knowledge from previous Cortex-M0 processor-based projects. In most cases you just need an update to your development suite, firmware in your debugger adaptor and then you can start your development with a Cortex-M0+ processor.

Simple FSM design

In order to illustrate various methods in designing FSM using a processor system, we specified a simple FSM design with four states as follows:

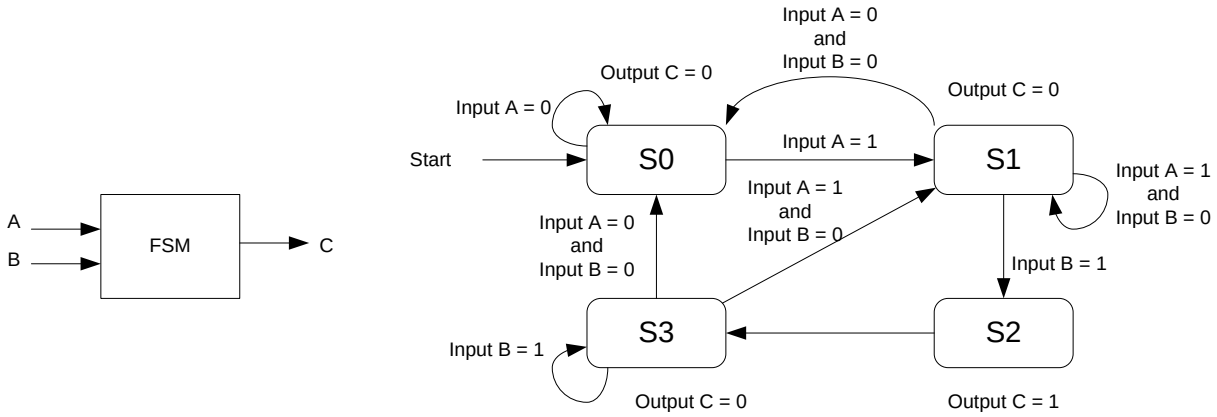

Figure 2: Simple FSM design example

For such a FSM to be implemented in a Cortex-M processor-based system, for example, using a Cortex-M0 processor, we can use the following simplified system level design.

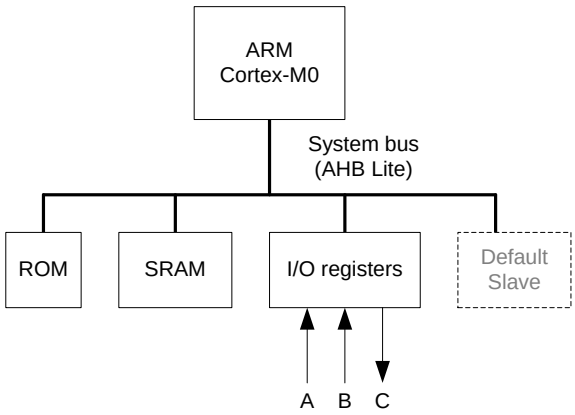

Figure 3: Example software FSM system design with the ARM Cortex-M0 processor

The optional default slave is added so that if the system has somehow failed and the processor accessed an invalid address, the bus system will respond with an error and this triggers a fault exception in the processor. The fault handler can then carry out corrective actions or reset the system.

The same FSM can also be implemented using a Cortex-M3 or even a Cortex-M4 processor. The Cortex-M3 and Cortex-M4 processors have multiple AHB Lite bus interfaces:

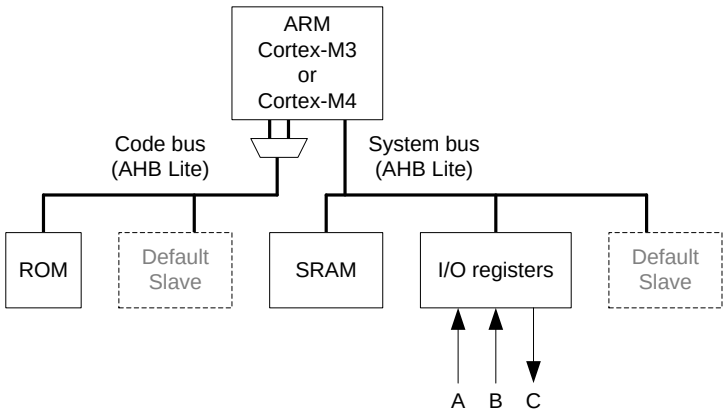

Figure 4: Example software FSM system design with the ARM Cortex-M3 or Cortex-M4 processor

If the same FSM design is to be implemented with the Cortex-M0+ processor, we can make use of the single cycle I/O port feature to accelerate the access to the inputs and outputs.

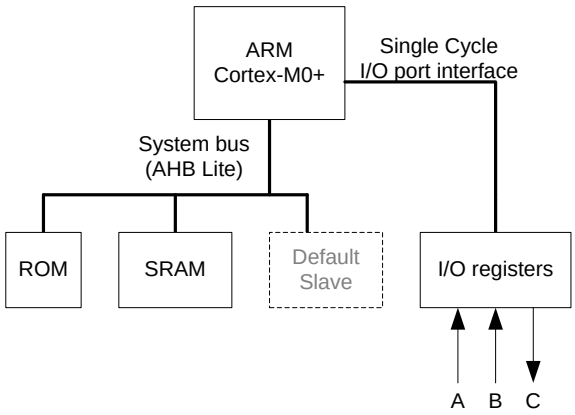

Figure 5: Example software FSM system design with the ARM Cortex-M0+ processor

Many software FSMs are implemented inside complex SoC and linked to the rest of the system in multiple ways. For example, the system bus of the FSM subsystem might be linked to the system bus of the application processor, and the debug system might be joined together using CoreSight debug architecture. In this way, the whole system can be debugged using just one debug connection.

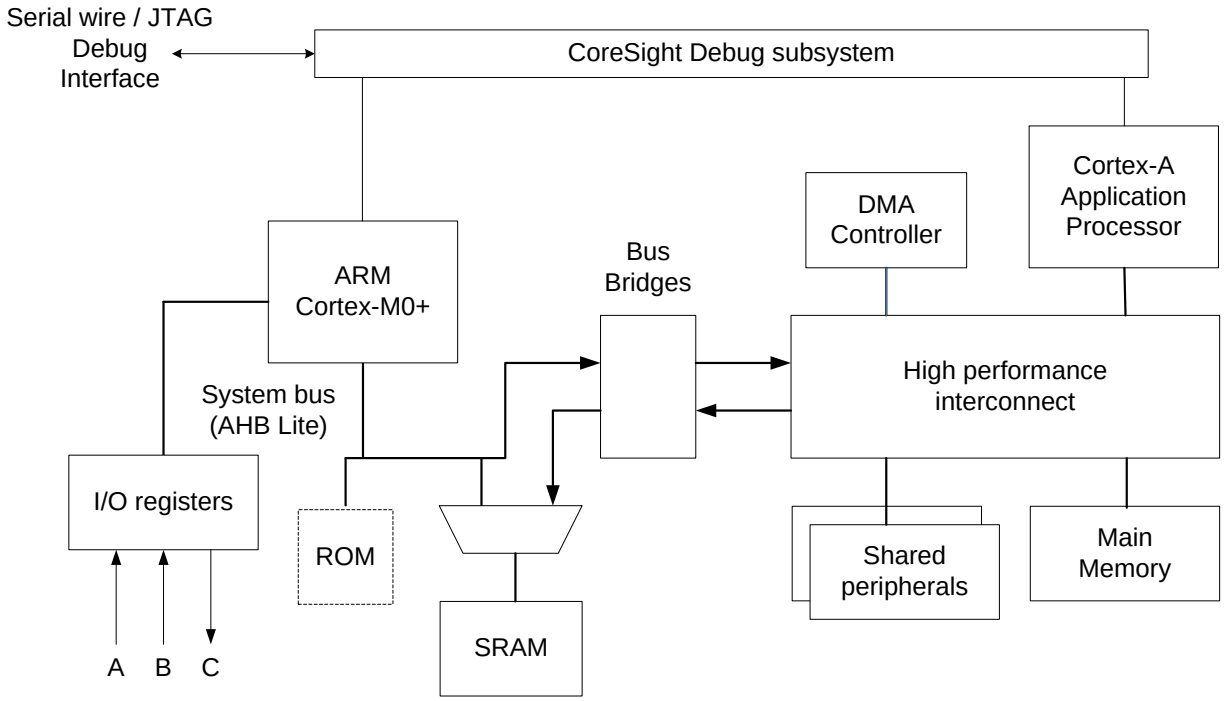

Figure 6: Cortex-M as a FSM subsystem in a complex SoC

In such a system, the Cortex-M processor can also access to shared peripherals. Potentially some of the I/O control tasks can be off loaded to the Cortex-M processor so that the I/O processing latency can be reduced. Also, if the main application processor is in sleep mode, there is no need to wake up the application processor when a peripheral need interrupt service. The interrupt service could be handled by the Cortex-M processor to save power.

Another potential benefit of such arrangement is to use the application processor or the DMA controller to load the FSM program image into the SRAM of the FSM subsystem before starting the Cortex-M0 processor. In this way, you can omit the ROM in the design to save silicon area and cost.

There are many different ways to arrange the programmer's model in the I/O registers. One common dilemma is whether we should have a register for each input, or have one register for multiple input bits:

- A single register with multiple input bit status can make it faster for state flow decision with switch/case statement.
- Multiple registers with one input status in each register make it faster for compare to zero and branch.

To solve the problem, we can implement both options in an I/O register block:

Address offset	Register	Type	Descriptions
0x0	INPUTS	Read only	Bit 2 – Input A. Bit 3 – Input B. Other bits are read as zero.
0x4	A	Read only	Bit 0 – Input A. Other bits are read as zero.
0x8	B	Read only	Bit 0 – Input B. Other bits are read as zero.
0xC	C	Read/Write	Bit 0 – Output C. Other bits are read as zero, write ignored.

If we have multiple output signals, we can use the same approach to allow outputs to the controller separately or together by using multiple address space.

You might notice that we keep the bit 0 and bit 1 of the INPUTS registers as zero. We will explain this later.

Assume the base address of the I/O register block is at 0x40000000, we can declare the registers in the following data structure.

Data structure for I/O register definition

```
// Define IO registers as data structure
typedef struct
{
  volatile uint32_t INPUTS;  // Inputs {B, A, 0, 0}
  volatile uint32_t A;       // Input A
  volatile uint32_t B;       // Input B
  volatile uint32_t C;       // Output
} IOREG_Type;

#define IOREG ((IOREG_Type *) 0x40000000 )
```

Using data structure instead of individual pointers is more efficient as the program only need one address constant, and the access for each register can be handled with load/store instructions with immediate offset.

There are many different ways to design this state machine in C programming language. A typical approach is to use switch/case statements:

Example of FSM implementation using switch/case statement

```
int main(void)
{
  // Define states
  enum FSMSTATE {S0, S1, S2, S3} curr_state;

  curr_state = S0; // Start
  IOREG->C   = 0;
  while(1){
    switch (curr_state){
        case (S1) :
            IOREG->C   = 0;
            if      (IOREG->B)    curr_state = S2;
            else if (IOREG->A==0) curr_state = S0;
          break;
        case (S2) :
            IOREG->C   = 1;
            curr_state = S3;
          break;
        case (S3) :
            IOREG->C   = 0;
            if (IOREG->B==0) {
              if (IOREG->A) curr_state = S1;
              else          curr_state = S0;
              }
          break;
        default   : // case (S0)
            IOREG->C   = 0;
            if (IOREG->A) curr_state = S1;
            else          curr_state = S0;
          break;
            }
      } // end while
}
```

This example code results in a code size of just 68 bytes in the “main()” (excluding vector table and C startup code).

If you want to reduce the code size further, you can change the code to use “goto” with labels. This is generally not preferred for general programming but is fine for FSM implementations.

Example of FSM implementation using “goto” and labels

```
int main(void)
{
  IOREG->C   = 0;
S0:
  IOREG->C   = 0;
  if (IOREG->A==0) goto S0;
  // else          goto S1; // This line not needed as S1 is the next line
S1:
  IOREG->C   = 0;
```

```
  if (IOREG->B) goto S2;
  else if (IOREG->A==0) goto S0;
  else goto S1;
S2:
  IOREG->C   = 1;
  // goto S3;       // This line not needed as S3 is the next line
S3:
  IOREG->C   = 0;
  if (IOREG->B)      goto S3;
  else if (IOREG->A) goto S1;
  else               goto S0;
}
```

This implementation reduces the code size to 48 bytes. In addition, the FSM code can be more efficient because it does not need to jump back to the "switch" statement in each state transition.

Note that in some modern C compilers, there is no need to re-code the FSM code to use "goto". A number of C compilers (e.g. including latest version of ARM C Compiler and IAR Embedded Workbench for ARM) already able to transform the FSM code in the first example into structure like the one in the second example.

For larger system implementations, you might want to use an embedded OS. By doing this, you can run several tasks in parallel (the responsiveness depends on the OS context switching time). Modern embedded OS provides various mechanisms to allow different tasks to communicate with each other. There are plenty of ways to implement FSM in such systems. For example, you can have multiple FSM running in parallel by having one task for each FSM, and use OS event communication to communicate between the multiple FSMs.

You can also use high level language such as UML (Unified Modeling Language) to develop your FSM design. Tools are available for these design flows and you can produce your FSM design with GUI within the IDE of development suite[1].

Reducing the power

One of the key advantages of using software FSM is that you can utilize the low power features of the processor to reduce the power consumption of the FSM. For example, you can put the processor into sleep modes when no processing is required, and use interrupts or event inputs to wake up the processor if any of the input change state.

While it is straight forward to use interrupts as method to handle hardware events, the interrupt overhead require might be undesirable in some situations. In this case, we can use the Wait-For-Event (WFE) sleep mode and wait up the processor with a hardware event, but not triggering an interrupt handling sequence.

For example, we can use the following logic to generate event to a Cortex-M processor to wake up from WFE (Wait-For-Event):

[1] An example is IAR VisualSTATE (http://www.iar.com/en/Products/IAR-visualSTATE/)

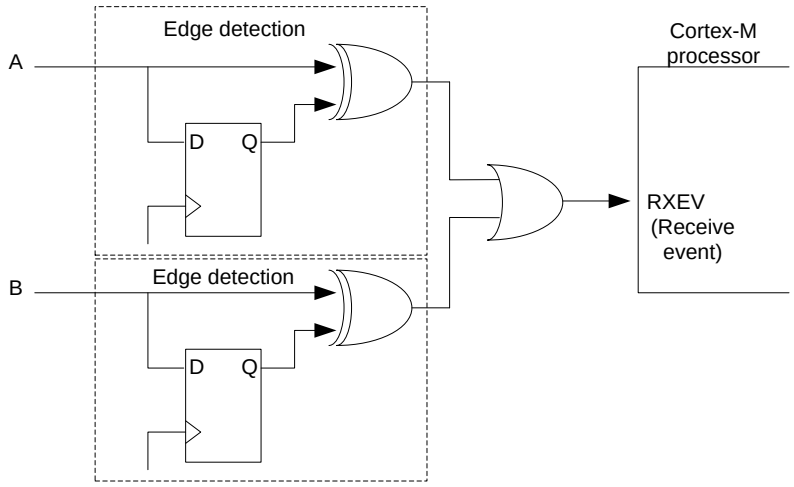

Figure 7: Using RXEV input to wake up processor from sleep when a input change

And WFE instructions can be inserted to the FSM program code to allow the processor to enter sleep mode when the FSM does not need any state transition:

Example of FSM implementation using "goto" and labels with sleep mode

```
int main(void)
{
  IOREG->C   = 0;
S0:
  IOREG->C   = 0;
  __WFE();
  if (IOREG->A==0) goto S0;
  // else          goto S1; // This line not needed as S1 is the next line
S1:
  IOREG->C   = 0;
  __WFE();
  if (IOREG->B) goto S2;
  else if (IOREG->A==0) goto S0;
  else goto S1;
S2:
  IOREG->C   = 1;
  // goto S3;       // This line not needed as S3 is the next line
S3:
  IOREG->C   = 0;
  __WFE();
  if (IOREG->B)      goto S3;
  else if (IOREG->A) goto S1;
  else               goto S0;
}
```

When the processor is in sleep mode, most of the clock signal paths can be gated off to save power. When an input changed state, a pulse is generated to RXEV input and wakes up the processor and resume program execution. This method does not require the use of interrupt mechanism, and therefore avoids interrupt latency.

Alternatively we can use the Send-Event-on-Pend (SEVONPEND) feature to achieve the same operation. When the SEVONPEND feature is enabled, the processor can wake up from WFE sleep if a new interrupt pending takes place. The interrupt does not need to be enabled, and this allows the processor to wake up from sleep without taking any interrupts.

Another trick to reduce the power is to copy the program code from ROM to RAM during startup, and execute the program from RAM and turn off the ROM completely. This also allows the system to run at a high frequency if needed because in typical systems the access speed of SRAM is faster than non-volatile memories (e.g. flash memories).

Decisions, decisions

In order to make the FSM responsive, the choice of the processor is very important. For example, if we need to manage the state S0 to S1 transition, we can use the following assembly instruction sequence:

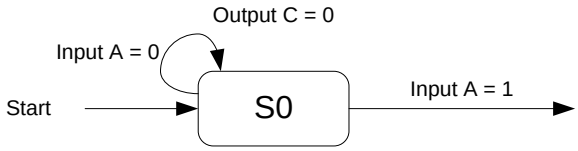

Figure 8: State S0 operation

```
; Assembly code fragment in ARM assembly syntax
; Cycle count based on Cortex-M0+ processor
;                              Size   Cycle   Comments
    LDR    R4, =0x40000000 ;  2      2       Base address of I/O block
    MOVS   R1, #0          ;  2      1
    STR    R1, [R4, #0xC]  ;  2      1       Output C = 0
S0:
    LDR    R0, [R4, #0x4]  ;  2      1       Read A
    CMP    R0, #0          ;  2      1
    BEQ    S0              ;  2    1 or 2    Loop if A = 0
```

In the Cortex-M0+ processor, the conditional branch only takes 2 cycles if the branch is taken, and 1 cycle if the branch is not taken. As a result, the loop only takes 4 clock cycles to execute.

The branch sequence gets longer as the decision gets more complex. For example, in state S1:

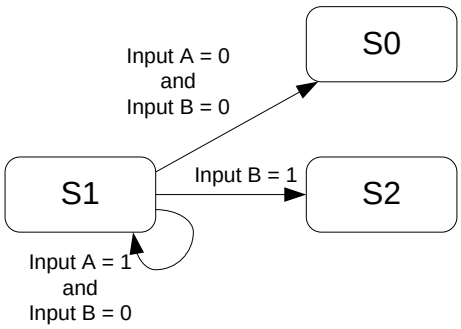

Figure 9: State S1 operation

We can either use two levels of conditional branch:

```
; Assembly code fragment in ARM assembly syntax
; Cycle count based on Cortex-M0+ processor
;                              Size   Cycle   Comments
;   ...                                       Previous code for state S0
S1:
    LDR    R0, [R4, #0x8]   ;  2      1      Read B
    CMP    R0, #1           ;  2      1
    BEQ    S2               ;  2    1 or 2   Goto S2 if B = 1
    LDR    R0, [R4, #0x4]   ;  2      1      Read A
    CMP    R0, #0           ;  2      1
    BEQ    S0               ;  2    1 or 2   Goto S0 if A = 0
    B      S1               ;  2      2
```

The number of clock cycles required for the loop (if A = 1, B = 0) become 8 clock cycles. With this arrangement, the more complex the decision is, the longer it will take for the loop to execute. Alternatively, we can use table branch to handle multiple branch target:

```
; Assembly code fragment in ARM assembly syntax
; Cycle count based on Cortex-M0+ processor
;                              Size   Cycle   Comments
;   ...                                       Previous code for state S0
S1:
    LDR    R1, =S1_branch_table      2      Note: execute once only
S1_loop:
    LDR    R0, [R4, #0x0]   ;  2      1      Read {B, A, 0, 0} via I/O port
    LDR    R2, [R1, R0]     ;  2      2      Read branch target address
    BX     R2               ;  2      2      Branch to branch target
    ALIGN  4                ;0 or 2          Make branch table aligned
S1_branch_table:
    DCD    S0               ;  4      -      Branch to S0 if B=0, A=0
    DCD    S1_loop          ;  4      -      Branch to S1 if B=0, A=1
    DCD    S2               ;  4      -      Branch to S2 if B=1, A=0
    DCD    S2               ;  4      -      Branch to S2 if B=1, A=1
```

In the earlier part of article, the INPUTS register is designed to have the two LSB set to zero. This helps the table branch operation because the branch target address values are place in word aligned addresses. In this way, we can reduce the loop to just 5 clock cycles.

If you are using the ARM Cortex-M3 or Cortex-M4 processors, there also specific instructions for table branches (TBH and TBB instructions).

How about more complex decisions?

For example, if we have 4 inputs, and we need to decide if a branch should be taken based on these inputs:

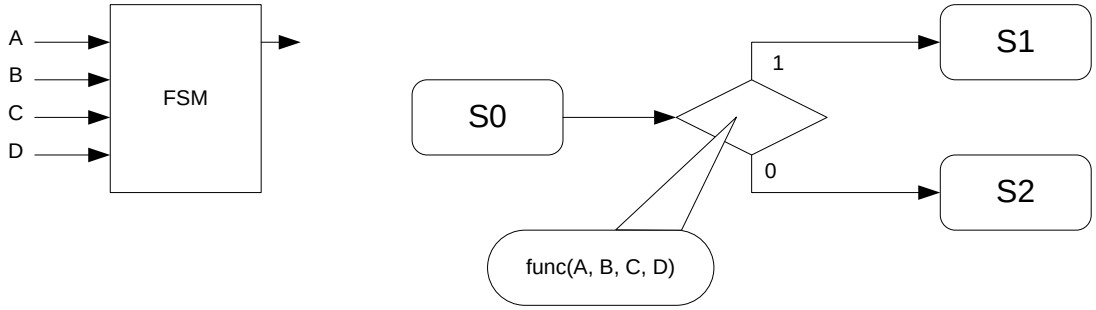

Figure 10: State transition controlled by a function of multiple inputs

If we use table branch to handle this, the branch table size can be quite big and need more program memory. So a different method is needed: first, we can calculate a constant value based on the Boolean function of the inputs A, B, C and D, and shift the bit required into the carry flag for the conditional branch:

```
; Assembly code fragment in ARM assembly syntax
; Cycle count based on Cortex-M0+ processor
;                              Size   Cycle   Comments
S0:
    LDR    R0, [R4, #N]     ;  2      1      Read {D, C, B, A} via I/O port
    LDR    R1, =BIT_PATTERN;         2      Bit map of Boolean function
    ADDS   R0, R0, #1       ;  2      1      Minimum shift is 1 bit
    LSRS   R1, R1, R0       ;  2      1      Shift bit to C flag
    BCS    S1               ;  2   1 or 2    Branch if C flag is set
    B      S2               ;  2      2
```

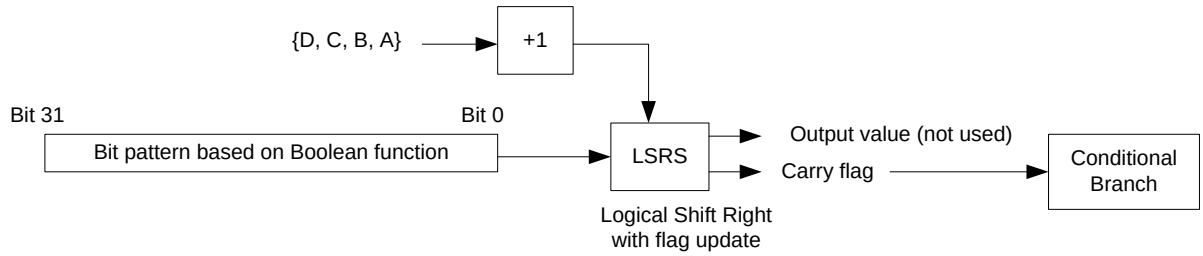

Figure 11: Conditional branch controlled by a function of multiple inputs

This method allows a complex branch decision with up to 5 inputs to be handled easily in a few instructions. If more inputs are needed, the Boolean function can be represented as an array of bytes, and place in a look up table in the program memory. It would also be useful to arrange the inputs into multiple input status registers to simplify the program code. For example, {A, B, C} is grouped to one register and {D, E, F} are grouped to another register in the following example:

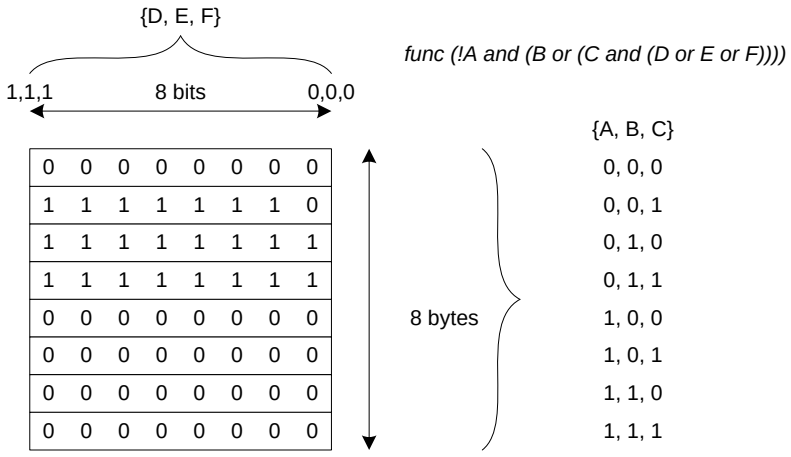

Figure 12: Creating a bit map of a Boolean function

```
                              ; tests for (!A and (B or (C and (D or E or F)))
  ADR  R4, bit_table
  LDRB R1,[R0,#N1]            ; Read {A, B, C} at offset N1 of I/O block
  LDRB R3,[R4,R1]             ; Read bit map table
  LDRB R1,[R0,#N2]            ; Read {D, E, F} at offset N2 of I/O block
  ADDS R1, R1, #1             ; Need to shift at least 1 bit
  LSRS R3, R3, R1             ; Shift required bit to Carry flag
  BCS  S1                     ; branch to state S1 if Carry flag is set
  B    S2
  ALIGN
bit_table
  DCB  0x00, 0xFE, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00
```

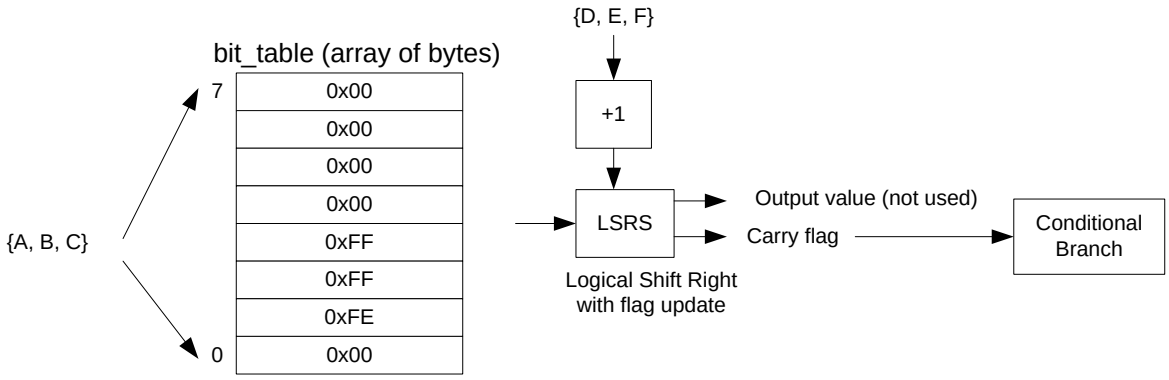

Figure 13: Conditional branch controlled by a function of six inputs

An example usage – Power Management Controller

Several ARM customers are using Cortex-M processors in their SoC products for power management control. In these products, the power management module of the OS kernel has a number of predefined power profiles, and different profiles are selected based on the work load of the processor.

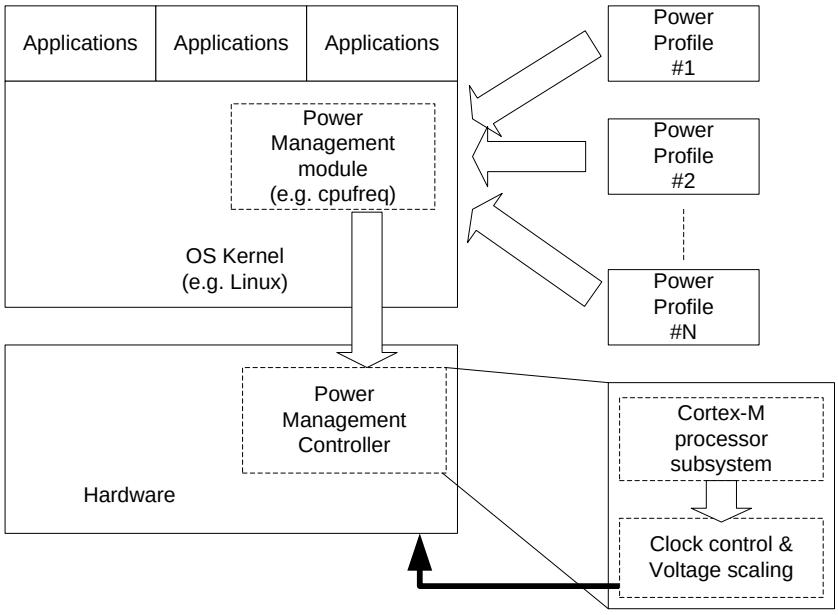

Figure 14 : Hardware acceleration FSM can be added to I/O interface if needed

The information about the required power profile is then being passed on to the Power Management Controller, which is a processor subsystem based on a Cortex-M processor. The Cortex-M processor then handle the actual voltage scaling and clock frequency control operation, which are low level I/O operations that require low latency and deterministic behavior.

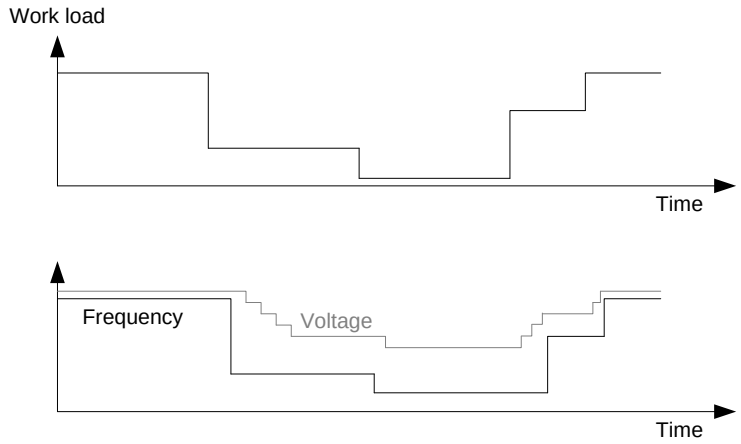

Figure 15 : Dynamic Voltage and Frequency Scaling (DVFS) in System-on-Chip (SoC)

Design considerations

I/O response speed

Some applications require a single-cycle pin response for certain events. While the latencies of the Cortex-M processors are quite low, it cannot handle some of these time critical requirement. It is possible to add additional hardware acceleration logic in the I/O register interface block to handle these events while using the Cortex-M processors for more complex data processing and FSM state controls. While this requires some small hardware FSMs, the complexity required for the hardware FSMs is much simpler than a complete hardware FSM design. But of course this means the flexibility of the FSM operation could be reduced.

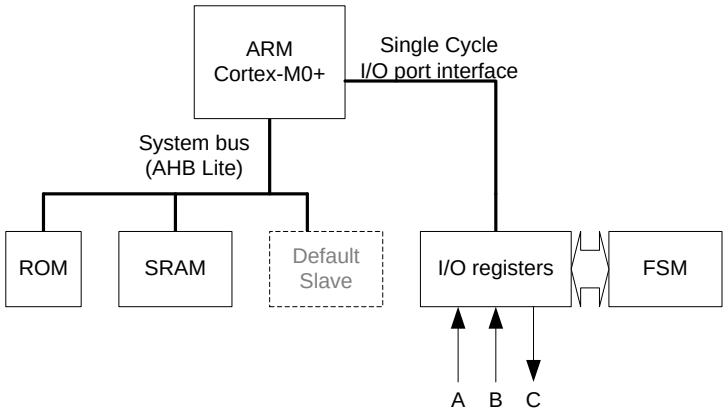

Figure 16 : Hardware acceleration FSM can be added to I/O interface if needed

FSM speed

We have already seen Cortex-M processor-based microcontroller products on the market running at over 200MHz. With modern cutting edge process technologies the operation speed of the processor can reach 700MHz or more.

Memory speed
The Cortex-M processors can operate at high speed (e.g. several hundred MHz). However, most on chip flash memories can only operate at up to 30 MHz or 50MHz. If the program is executed from flash the speed of the state machine would be limited. However, one of the advantages of using modern processors like the Cortex-M processor is that you can copy a program to SRAM and execute it from SRAM without any wait state on the processor bus. By doing so, you can even power off the flash memory to further reduce the power consumption.

Program location
Many FSM replacement scenarios using Cortex-M processors are found in complex SoC. These designs often contain other processors and have other program storage in the system. In order to reduce the number of program storage memories in the system, it is possible to hold the processor of the software FSM in reset state during startup, and copy the program memory for the software FSM from the firmware storage location to the SRAM of the software FSM, and then start the software FSM operations.

In order to make it even easier to manage such program storage arrangement, the Cortex-M0+ processor has a new feature that delays the starting of a processor after reset with an input pin.

Mixed language development
Almost all development suites for the Cortex-M processor support mixed C and assembly language development. You can implement part of the state machine in assembly for higher speed, and some of the sequential process in C functions and call these functions from assembly code.

Conclusions

In this paper we have seen that it is possible to replace a hardware state machine with a low gate ARM Cortex-M processor-based system, and benefit from the flexibility of a software based approach whilst maintaining timing determinism and low latency. This approach make handling of complex sequential processes becomes easier.

You can also connect standard peripherals designed for microcontrollers to add connectivity to your FSM. When the FSM is not used, you can use the system just like a normal microcontroller.

You can also take advantage of various low power of the processor to reduce power consumption, and utilize the debug capability of the processor to help you to development and optimize the FSM design.

Various methods can be used to handle FSM state transitions. Together with the low latency nature of the Cortex-M processors, the responsiveness of the FSM can be comparable to hardware FSM, with many more additional capability and flexibility.