Creating Embedded Microcontrollers (Programmable State Machines)

By Ken Chapman
Senior Staff Engineer, Applications Specialist - Xilinx UK

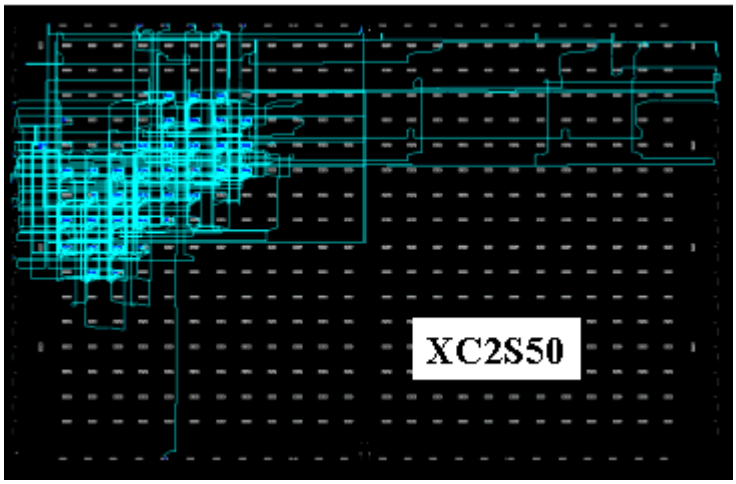
Introduction

It may seem strange that, as a Xilinx Applications Engineer, I am such an advocate of microprocessors. However, as my previous articles regarding "Performance + Time = Memory" explained, microprocessors really are a very efficient use of silicon. Although specific hardware implementations using time sharing techniques can be formulated, a microprocessor offers superior levels of flexibility via the software programming methodology, providing the "Time" factor is adequate. This is particularly useful for applications in which the processing to be implemented is "esoteric" in nature and would require very complex state machines.

With the release of the MicroBlaze™RISC processor soft core last year, and the more recent introduction of the Virtex-II Pro™ devices with their embedded PowerPC 405 hard cores, you can imagine how much potential I can see for future applications with Xilinx devices. However, I have not merely waited for these cores to become available, but have been implementing my own processor macros inside Xilinx devices since1993. From the publications on the subject, I also know that I have not been the only one exploiting Xilinx devices in this way.

My particular interest is in the creation of very small processor macros. These have much more in common with the world of microcontrollers than full-blown data processors, which are larger and more powerful. My focus is to bring the most significant advantages that a processor can offer to a design environment at minimum cost. For this reason, I have considered these small processors to be "Programmable State Machines" and refer to them as "PSM".

By exploiting the Xilinx device architecture, it has been possible to create processors such as the KCPSM, which occupies just 35 CLBs in a Spartan-II™. The plot below shows a single KCPSM in an XC2S50 Spartan-II device. You can actually fit 8 PSM processors in this device and still have 30% of the device remaining for hardware circuits. Even the smallest Spartan-II (2S40) can support 2 of these PSMs.



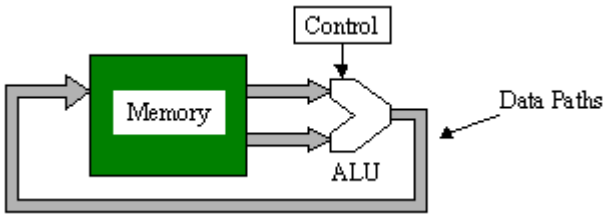
I know that many of you have enjoyed using this macro and I never cease to be amazed at the applications you find for it. For many applications, a single PSM combined with hardware circuits has been more than

adequate. For applications requiring more processing power, the ability to use multiple KCPSMs in a single Xilinx device has the added advantage of distributing the processing and keeping the interfaces-to-hardware circuits simple and independent. The introduction of MicroBlaze and PowerPC cores has only increased the usage of PSM macros. These tiny processors can handle tasks independently to the main processor and with much tighter and predictable coupling to the hardware circuits.

In this *techXclusive* mini-series, I will explain how I derived the architecture of KCPSM and how it exploits the architecture of the Xilinx FPGA devices. I hope you will be inspired to create your own application-specific PSM processors as well as find new applications for existing PSM macros.

How many bits?

How many bits should a processor have? The nice thing about an FPGA is that you can decide what is most suitable for your applications and implement the data width that you require. In general, the wider the data width, the more logic you will need to support it, and the more capable the processor will become. Conversely, the narrower the data width, the smaller the implementation, but the lower the arithmetic performance. My primary objective was a small size and low cost, so this meant a narrow data width and acceptance of the accompanying lower arithmetic performance. As the processor is surrounded by programmable hardware that offers the ultimate in high performance, the processor was intended to be a complex programmable state machine and not a DSP processor.



Even so, I found it very hard to decide on a bit width, as I did not have one specific application in mind. So when I designed my first PSM in 1993 using XBLOX™ (Anyone remember that one? It was fundamentally a schematic synthesis tool), I avoided the issue by creating a processor in which the data width was defined only at the point of synthesis. It turned out to be quite difficult to construct a processor around this decision, and this lead to many other restrictions, especially when combined with the limits of the XC4000™ devices then available. However, it proved to me that efficient PSM processors could be made, and I started to gather feedback from the most important people -- our customers. Of the people that used this first PSM, nearly all defined an 8-bit data width. It was a simple case of conformance!

Given the choice of conforming to a standard of 8, 16, or 32 bits, the future PSM processors just had to be 8 bits. Not exactly a sound engineering decision, but I learnt never to fight with nature! However, I would ask you to consider "non-standard" bus widths if it fits with your needs.

Embedded Program

A PSM, like any other processor, will execute a program. A program is formed by a set of instructions that are defined by the user and held in a memory. Each instruction is encoded into a machine code. That much is obvious, but where should that program be stored?

If you use a standard processor, it may be natural to think of the program being stored in a ROM or RAM device, or even on a floppy disk. But if you were implementing a state machine as part of your Xilinx FPGA design, you would not connect any external components. It was clear to me that a PSM must be 100% embedded in the device and totally self-sufficient. In this way, you can make the decision to use one or more PSMs anywhere it makes sense to do so without concern for the design of the PCB on which the FPGA is sitting. This meant that the program had to be implemented inside the device.

Unfortunately the XC4000 family did not contain any dedicated memory, so the CLBs had to be converted to program ROM. However efficient the processor, this ROM was expensive, so programs had to be small and

very efficient. Imagine my joy at seeing block RAM appear in Virtex™! Here were 4096 bits of memory just asking to hold a PSM program.

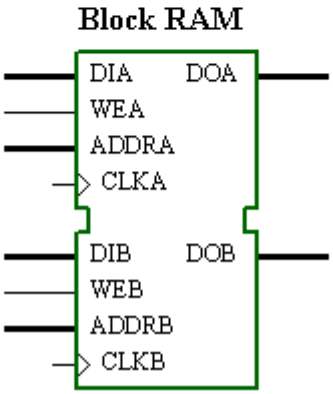
In Peter Alfke's *techXclusive* "Using Leftover Multipliers and Block RAM in Your Design", we can see that a block RAM can be used to implement a state machine. This is similar in many ways to the way a PSM works, but it does not exploit the dimension of time. The result is a relatively simple state machine that operates very quickly. With a PSM, we can achieve very complex state machines that work relatively slowly.

Block RAMs are very flexible in aspect ratio and are initialisable. It would again be very awkward to have to go through a program memory-booting phase before the state machine could become active. However, since all 4096 bits of each block RAM are defined in the configuration bit stream of the device, the PSM is able to operate from the very first clock edge.

So, the block RAM of Virtex (which then became the basis of Spartan-II devices) was to be the program ROM. This defined the second major limit for the PSM in that the size of program would be limited to that supported by one block RAM.

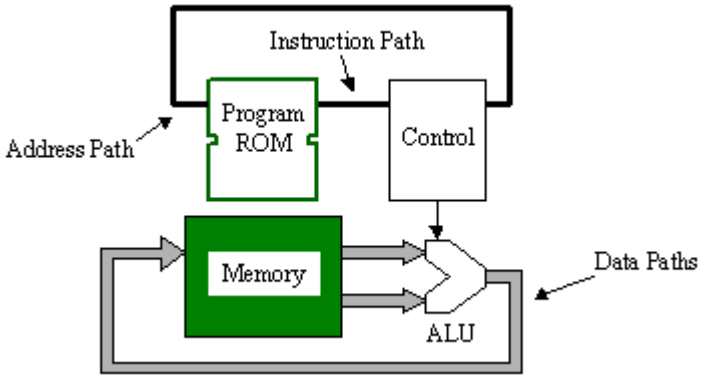
Block RAM Aspect Ratios

Locations	Instruction Width	Address Bits
4096	1	12
2048	2	11
1024	4	10
512	8	9
256	16	8



Embedded Program Advantage

Embedding the program ROM inside the FPGA has the advantage that all of the inputs and outputs of both the program ROM and the processor are "virtual pins" within the FPGA fabric. This means that there is no need to consider bus sharing to reduce the number of pins and busses. It is also obvious that a Harvard architecture is the natural selection. All the requirements for multiplexed data and instructions that would waste time, rather than exploit it, are avoided.



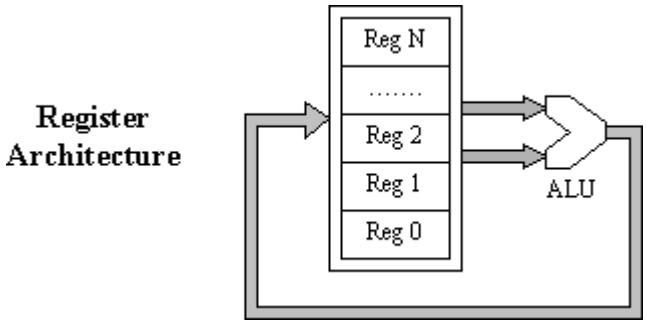
Given that there is no real restriction on the number of virtual pins the PSM can have, it is possible to explore all of the aspect ratios supported by the block RAM. At first, it may appear reasonable to adopt 8-bit instructions to be the same as the data path. This would make sense in a shared memory and bus system, but here we do not need to be constrained in that way.

By having a narrow instruction width, we have the advantage of more program memory locations, but a greater need to encode instructions. It will also be necessary to have fetch cycles to obtain operands when required. This will increase the amount of decoding and sequencing logic and lower performance due to the number of cycles per instruction. As multiple locations will be needed for many instructions, having more memory locations does not actually imply that the program can be longer.

Opting for a wider instruction path means that less decoding logic is required, and there is the potential for each instruction to be completely self-contained in terms of operation and operands obviating the need for fetch cycles. Given the desire to keep the PSM small, it was therefore decided that the 256×16 aspect would be most suitable. Admittedly, a program of only 256 instructions was going to be a constraint, but the intention was to implement a complex programmable state machine rather than full-blown data processor.

Register, Accumulator, or Stack?

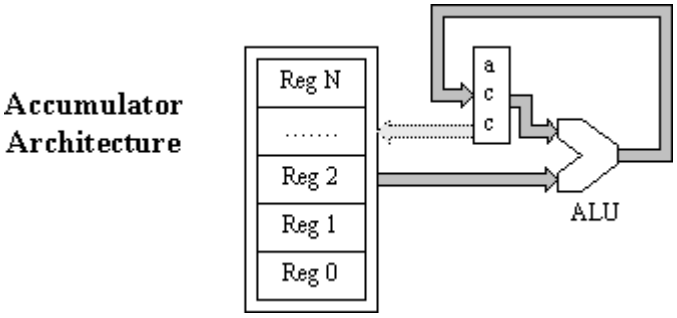
There are three fundamental architectures in which the data memory and Arithmetic Logic Unit (ALU) can be organised. Hybrids of these also exist, but we will focus on the basic forms to make a selection that is most suitable for a PSM.



The **register architecture** has a finite number of registers that the program may select in any order. This tends to make manual programming (i.e., writing at assembler level) a straightforward task. It is desirable to have a large number of registers to perform complex tasks and hold multiple variables locally. Obviously, the more registers there are, the larger the implementation of a processor due to the number of storage elements and the multiplexer logic that is required to select the operands applied to the ALU.

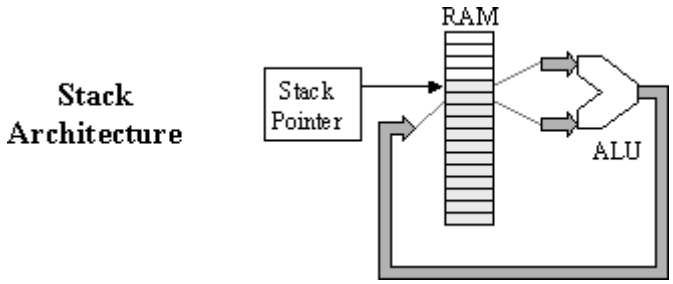
Equally significant is the impact a register-based architecture has on instruction size. Some bits must be used to specify each register used in an instruction. For example, a 3-bit binary code is required to select

one of 8 registers. Therefore, 6-bits would be required to specify the two operand registers. A further 3-bits may be required to specify a destination register for the result.



The **accumulator structure** is almost certainly associated with registers or memory to hold the various variables. The advantage of the accumulator structure is that one of the operands and the destination for the result is implied and does not need to be specified in the instruction encoding. Hence, the instruction encoding only needs to reserve bits to identify the remaining operand, and less logic is required to select the register or alternative source.

The disadvantage of this structure is that a program will tend to expend instructions and time simply moving an initial value into the accumulator or storing the accumulated value. Whilst long sequences involving the current accumulated value will be efficient, more esoteric programs, such as those implementing complex state machines, will become tedious to write and be slow to execute as the accumulator is continuously initialised and stored.



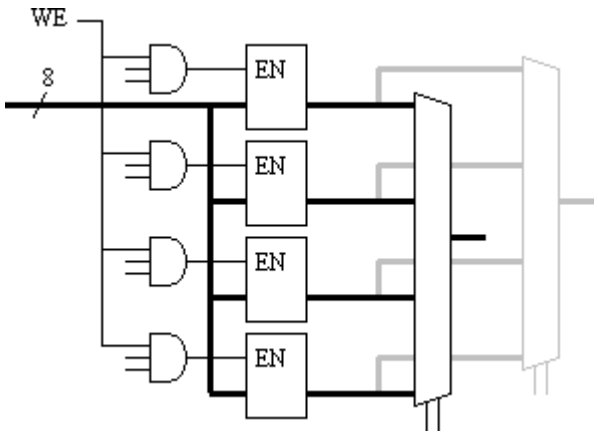
The **stack architecture** is probably the most efficient in terms of silicon resources, as it links directly to memory that is forming the stack and requires no data selection logic other than the stack pointer. The instruction encoding is also very efficient because the location of both operands and the result are implied as being the top of the stack.

However, even more instructions (stack PUSH and stack POP) are expended to ensure that the correct data is located at the top of the stack. Correct sequencing of the ALU operations will result in excellent code density and execution speed, but this "reverse polish" style does not come naturally to most of us and greatly impacts the desire to utilize a PSM that provides an easy methodology for implementing complex state machines.

Selecting the Register Structure

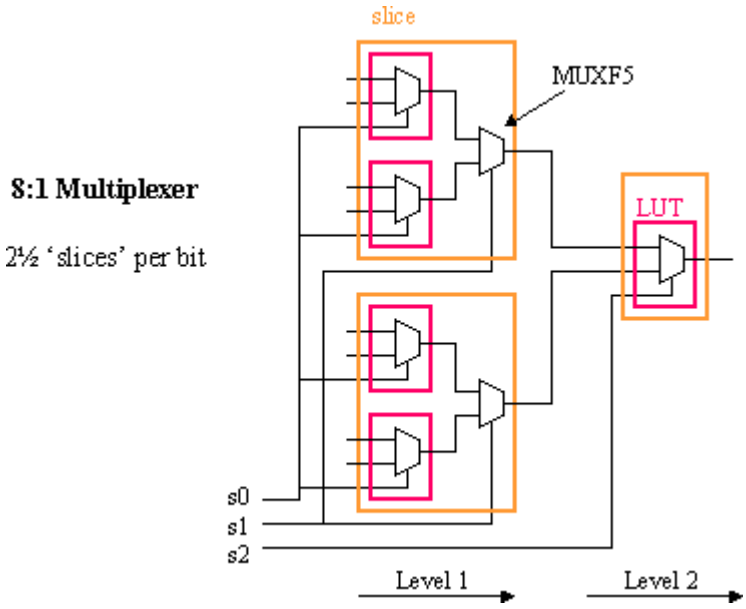
{- The stack architecture is not the best choice, as the methodology is too cumbersome.
- The accumulator structure seems reasonable, but expending too many instructions to move data is a concern given limited program memory space.
- The register architecture appears to be the best for PSM applications, but it could be expensive.

The cost of implementing a register bank can be high, as the following diagram illustrates. Only four registers are being implemented; there are 32 flip-flops for 8-bit data, which would occupy 16 "slices" of a Spartan-IITM or VirtexTM device.



We must now add the operand selecting multiplexers and clock-enable gates to the registers. A 4-to-1 multiplexer would require a complete slice per bit by combining the two LUTs and the dedicated MUXF5. Hence the 8-bit multiplexer requires 8 slices.

To fetch both operands at the same time, a second multiplexer is required. This 16-slices of logic is free if the combinatorial logic is mapped into the same slices as the flip-flops. However, placing combinatorial logic between the registers and the ALU would compromise the performance of the processor. The clock-enable gates only require 2 slices, but again would add combinatorial delay.



Increasing the number of registers does not seem to be a good idea! The table below illustrates the number of slices required by a Spartan-II to implement a selection of register bank sizes. Whilst it is obvious that more flip-flops are required, the multiplexer logic dominates the size. The multiplexers for 8 and 16 registers also incur another level of logic delay as indicated by the 8:1 multiplexer above. Virtex-II has dedicated MUXF6 and MUXF7 components that help reduce the size of larger multiplexers and significantly increase performance.

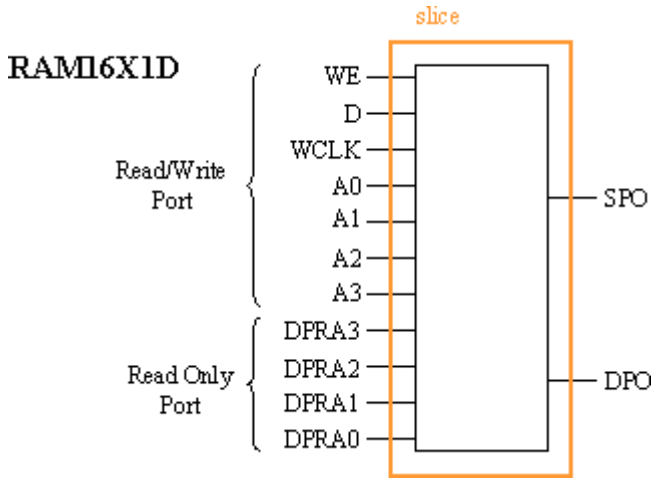
8-bit Register Bank Resources (all figures in 'slices')

№. Registers	Flip Flops	1-bit Mux	Dual register select Mux's	Enable gates	Packed Size
4	16	1	16	2	18
8	32	2½	40	4	44
16	64	5	80	8	88

So, it appears that the desire for more registers that will make a PSM easy to use must be balanced with the expense of implementation in both size and performance. Fortunately, this is where one of the most powerful features of Xilinx devices comes into play...

Distributed RAM

The SRAM configuration cells normally used to set the "gate" functionality implemented by a LUT are also available within the design directly as RAM. Hence each LUT offers 16 bits of RAM, and a "slice" can provide 32 bits of RAM. Of particular interest to PSM development is the ability to trade 16 bits of RAM per slice in order to achieve a type of dual-port RAM that is ideally suited for implementing a register bank.

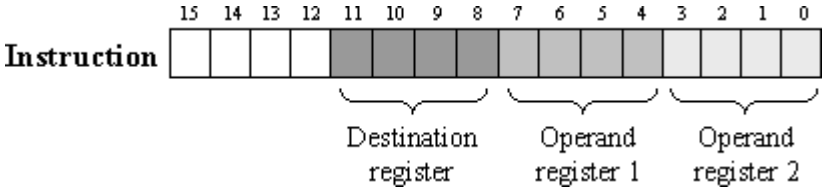


The 16 RAM cells provide all the functionality of a 1-bit, 16-register bank, complete with selective write enable. The second port enables a second operand to be accessed in the same way that a second multiplexer was used with discrete registers. With this great feature, an 8-bit, 16-register bank can be implemented in just 8 "slices" (compared to 88 "slices") and significantly increases performance.

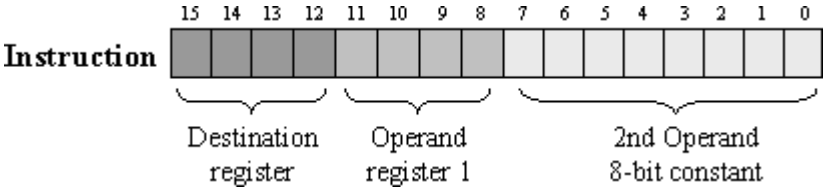
Given the potential offered by these 8 "slices", it is easy to see why a register-based architecture should be considered and why it makes sense to include 16 general-purpose registers in the Virtex-E and Spartan-II PSM implementation.

Instruction Considerations

In Part 1, I described how the 256×16 aspect ratio appeared to be advantageous for storing the program. Now that we have selected a register-based architecture and seen the potential for 16 registers, we can consider the initial format of the instruction encoding.

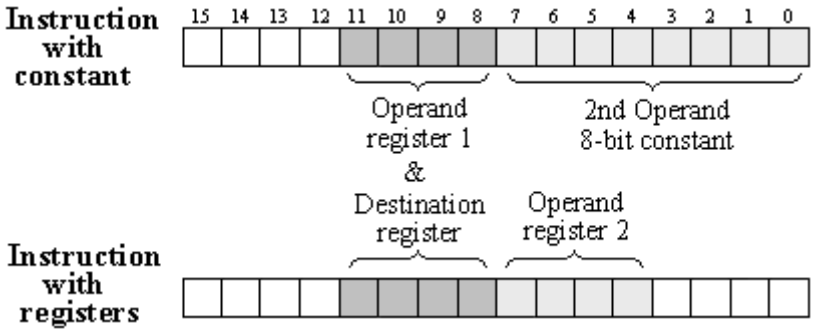


It is also necessary to consider operations involving a constant value. It is desirable to keep all instructions self-contained, so an 8-bit constant must be specified within the instruction encoding together with the register operands.



Clearly, the instruction now has to specify so much operand information that there is simply no space to define the operation. Although exploiting both ports of the block memory could form a wider instruction format, the even wider aspect ratio would further reduce the program length (128×[16+16] aspect ratio). Alternatively, we could consider using a fetch cycle to obtain constant values, but this again would reduce program length and lead to variable instruction size and duration.

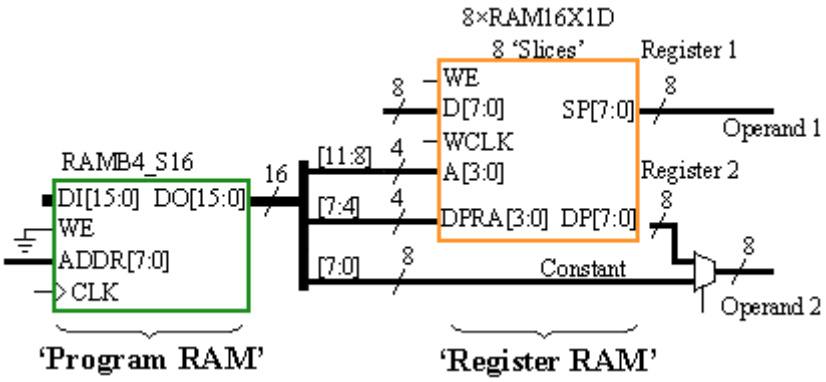
Given the desire to keep all instructions self-contained and maintain the natural 256×16 program ROM aspect ratio, a compromise must be made. In this case, the destination register was inferred by making it the same as the first operand register. This releases 4 valuable bits of instruction for encoding the operation and reveals the primary encoding for all ALU-related instructions.



It is clear that the programming may not be quite so flexible, and that additional instructions will be required when the first operand register contents must be preserved. However, program coding can often be organised such that this is not such a restriction, and can even be advantageous in the same way as an accumulator. The ability to specify a constant as the second operand (at no cost to program size or performance) will also be valuable when defining the instruction set. (I will study this later in the series.)

Memory Controlling Memory

The structure of a PSM is already emerging. Through this very simple and progressive analysis, we can see how important memory is to a processor and why the Virtex and Spartan-II devices are well-suited to this application.

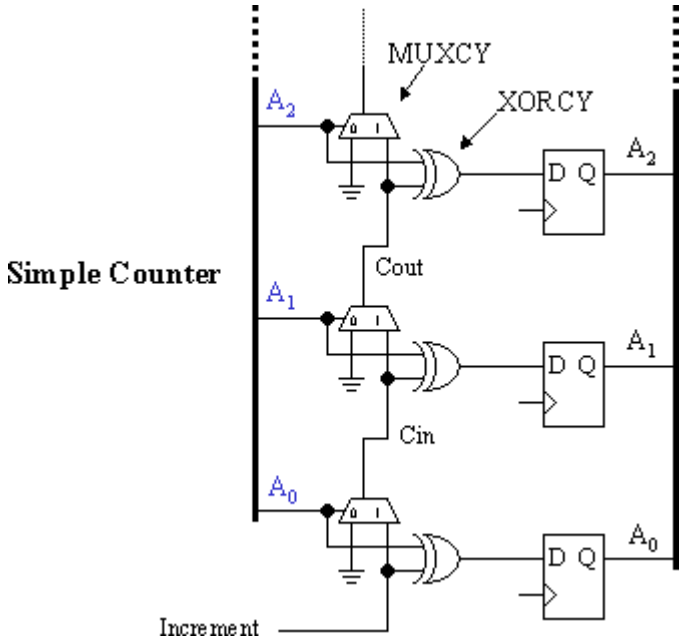


The program memory acts as the controller and is formed using block RAM. The variable data will be held in registers, and these are made cost-effective via the highly efficient distributed dual-port RAM. Without this distributed RAM option, the 44 CLBs (88 "slices") required to implement the register bank alone would already exceed the size of the complete PSM implementation!

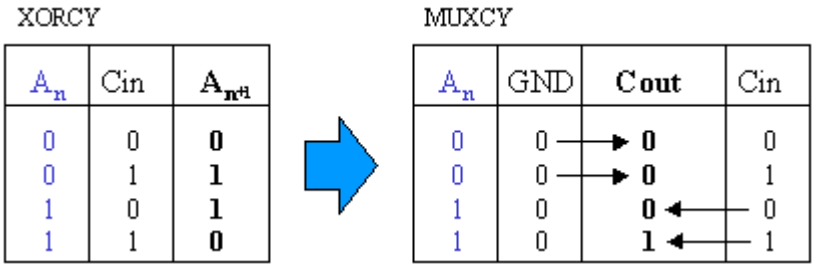
Programs are sequential most of the time!

A traditional processor executes instructions one at a time from sequential memory locations of the program memory. Execution normally starts at memory location zero. This the reason for the requirement of a program counter (PC), which increments through the program memory locations.

The simple "increment" operation of a program counter is very efficient and easy to implement in the Xilinx Virtex™ and Spartan™-II devices. Dedicated carry logic components and flip-flops within each logic "slice" are all that is required to implement a simple incrementing counter.



The carry logic is able to implement an increment function without using the look up table (LUT). The operation of each bit is defined by the following truth tables. Observe how the MUXCY propagates the carry signal.

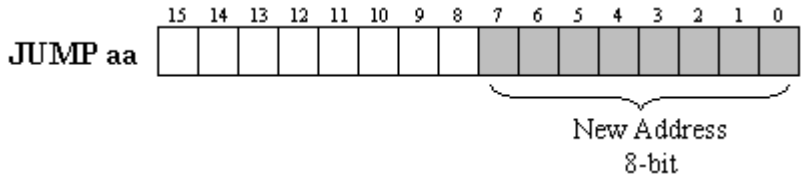


Because each logic "slice" contains the carry logic components and flip-flops for 2-bits, the required 8-bit program counter to access 256 memory locations can be implemented in just four "slices". However, the program counter must do more than increment.

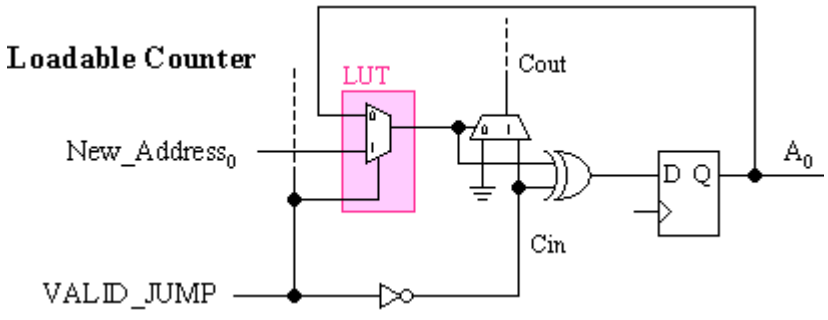
Jump with control

As a minimum, the PSM processor should have a way to repeat sections of code by enforcing the program counter to non-sequential address locations. The ability to jump (or branch) to any specified program memory location would allow for greater programming flexibility. To make a processor practical, the jump would be performed only under specified conditions (conditional jump) so that different sections of code are executed under different circumstances.

In Part 1, we decided that the 16-bit aspect ratio of the block RAM was more suitable for a PSM as it would enable the operands to be included within each instruction. With a JUMP instruction, the operand is the desired program counter value after execution; therefore, the JUMP instruction must specify an 8-bit address for the PSM program counter.



Now, the program counter must be loadable. When the instruction coding indicates that a JUMP must occur, the normal counter feedback must be replaced with the new value and the increment prevented. The LUT associated with each bit of the counter is now ideal for implementing the multiplexer to perform this selection.

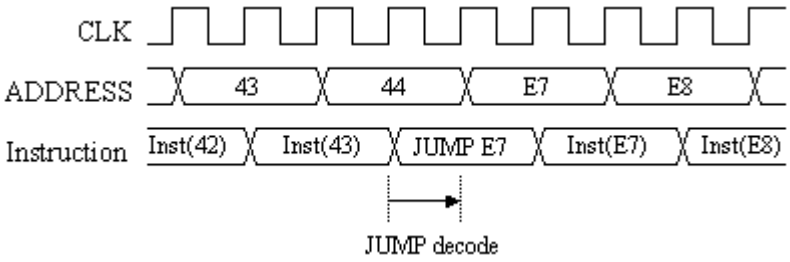


The VALID_JUMP signal will be active-High when a JUMP instruction is detected and when any conditions that are imposed on the execution are met. This will be a logical decode based on the remaining 8 bits of the instruction word and flags from the ALU. The inversion of the VALID_JUMP signal is used to drive the input to the carry chain to prevent the new address value from being incremented as it is loaded. This inverter is absorbed into the dedicated carry logic such that it requires no additional resources and does not impact performance.

Two Clock Cycles per Instruction

The JUMP instruction indicates that a decision will take place, and control over this decision implies the need for timing on the PSM. On the next clock edge, the program counter will either increment or load. The JUMP instruction therefore requires one clock cycle to determine which address of program memory will be accessed next.

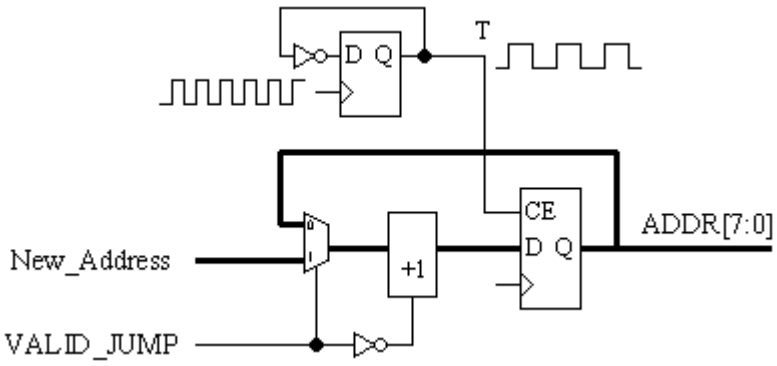
The program is stored in block RAM. It now becomes crucial to realise that the block RAM is a synchronous memory which must be clocked to both write and read data at a given address. Hence, if the program counter takes one clock cycle to determine the address, and the block RAM takes another clock cycle to access the instruction located at that address, the fundamental operation of a PSM takes two clock cycles per instruction.



In the above timing diagram you can see that most instructions result in the simple increment operation of the program counter. When the JUMP instruction located at address 44 is read from the block ROM, the program counter is loaded with the new address value supplied (E7) and the address jumps on the next clock edge.

The decoding of a JUMP instruction must take place in one clock cycle. All other instructions can be completed in two clock cycles; we will exploit this in the ALU and I/O instructions later. Although it is possible to execute all non-JUMP instructions at one-clock intervals, the handling of a JUMP instruction becomes more complex and detracts from the concept of the PSM being small and simple. The constant two clock cycles for every instruction also makes execution time easy to predict when writing a program. Processor terminology refers to the two cycles per instruction as T-states.

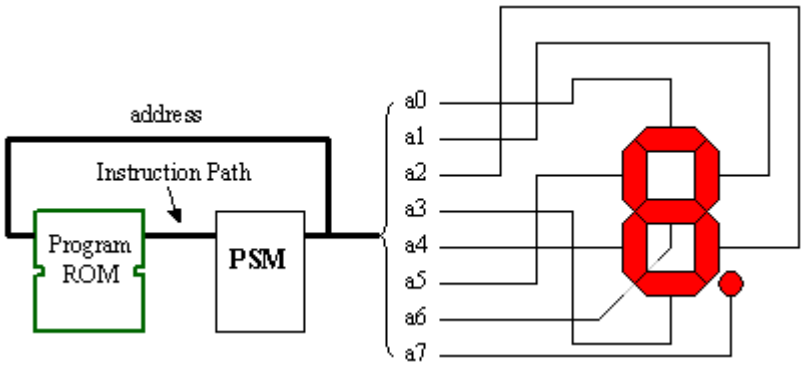
To make the program counter change only every two clock cycles, the dedicated clock enable on each of the counter flip-flops can be controlled using a simple toggle flip-flop.



The toggle flip-flop is essentially the only logic forming the "state machine" of the PSM. The counter logic exploits the capability of the four "slices" very well.

Our First Program -- Without an ALU!

With only a JUMP instruction, we can already see that a program can be written for the programmable state machine. In this simple example, the address signals are used to drive a 7-segment display directly. The program is written to ensure that the address sequence provides a decimal counter on the display. The display would change every two clock cycles.



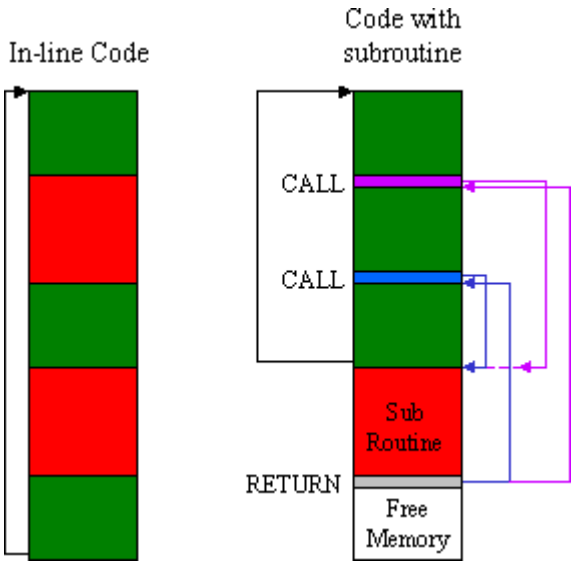

```
Addr Instruction    Comment

 03    JUMP 5B       ; display '1' and go to '2'
 07    JUMP 7F       ; display '7' and go to '8'
 3F    JUMP 03       ; display '0' and go to '1'
 4F    JUMP 66       ; display '3' and go to '4'
 5B    JUMP 4F       ; display '2' and go to '3'
 66    JUMP 6D       ; display '4' and go to '5'
 6D    JUMP 7D       ; display '5' and go to '6'
 6F    JUMP 3F       ; display '9' and go to '0'
 7D    JUMP 07       ; display '6' and go to '7'
 7F    JUMP 6F       ; display '8' and go to '9'
```

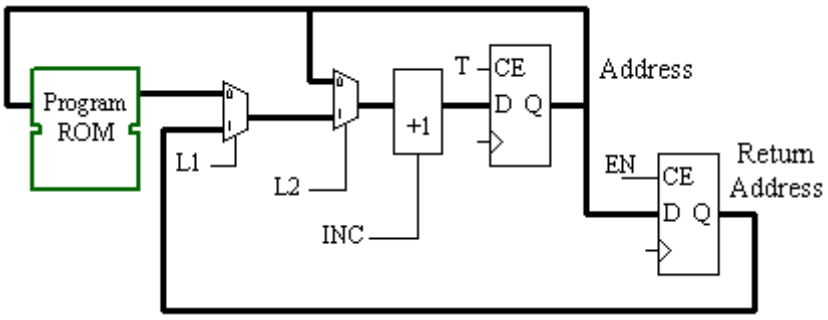
Whilst this simple example may be an interesting concept, it is unlikely to be a practical way to work with a PSM. However, the advantage of a fully embedded processor is that all signals are available to interface with the programmable logic of an FPGA. By organising programs to have routines located at particular addresses, it is possible to decode these addresses and trigger events external to the processor (but still internal to the FPGA) without actually performing I/O operations.

Subroutine for Real Programs

The PSM is intended to be a practical way to exploit time and share logic. A subroutine extends this concept by enabling sections of common code stored in memory to be shared by different parts of the program. Code does not need to be "in line," and this makes it potentially more compact and easier to write. Given the restricted program space of a PSM, a feature that enables compact code is highly desirable. Anything that makes writing a program easier is always desirable!



The CALL instruction is similar to a JUMP instruction in that the operand will again provide a new address to be loaded into the program counter provided conditions have been met. The RETURN instruction also causes the program counter to load a new address at the end of the subroutine. However, unlike the JUMP or CALL instructions, there is no operand to specify the new address; this must be derived by the PSM itself. To achieve this address specification, the CALL instruction must also store the current program counter value.



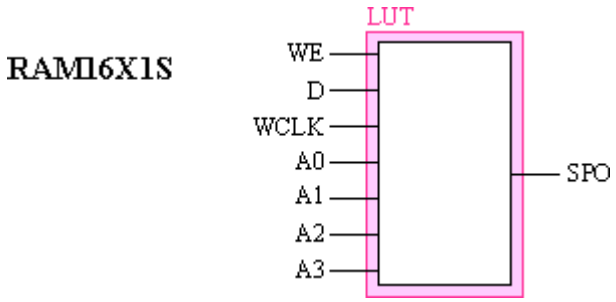
A register connected to the program counter is able to capture the address value when a CALL instruction is encountered. When the RETURN instruction is detected (at the end of the subroutine), the value from this register can be loaded back into the program counter. The program counter now has two sources of new address information, and a second multiplexer is required to make this selection, requiring four additional "slices". Closer inspection reveals that during a RETURN instruction, the return address value must be incremented as it is being loaded so that the instruction following the original CALL is executed.

Instruction	EN	L1	L2	INC
Normal	0	×	0	1
JUMP	0	0	1	0
CALL	1	0	1	0
RETURN	0	1	1	1

Careful allocation of instruction op-codes means that the decoding logic to control these signals can be very simple, and that sometimes an instruction bit can drive a control directly.

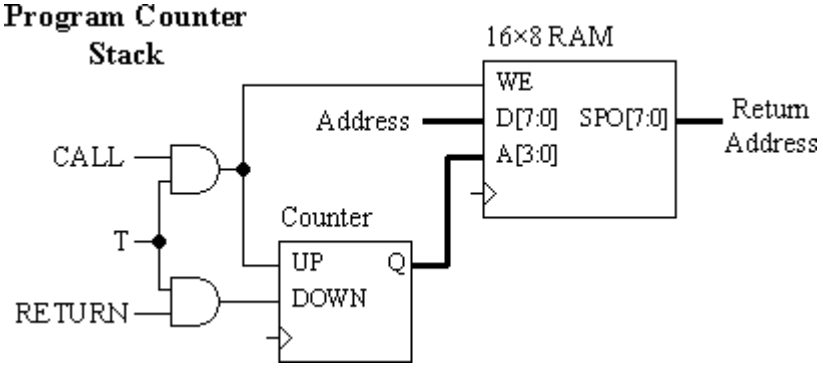
Don't just use it, exploit it!

The register used to preserve the address value is formed of eight flip-flops and therefore occupies another four "slices". In Part 2 we saw how distributed dual-port RAM could very efficiently provide a register bank, and now we can exploit distributed RAM in single-port mode to provide an address stack.



With a program address stack, nested subroutines can be executed, making programs even smaller and easier to write. Because the program address stack is implemented independently of the ALU, registers, and I/O, no special instructions or programming styles are required to initialise or control the stack operation..

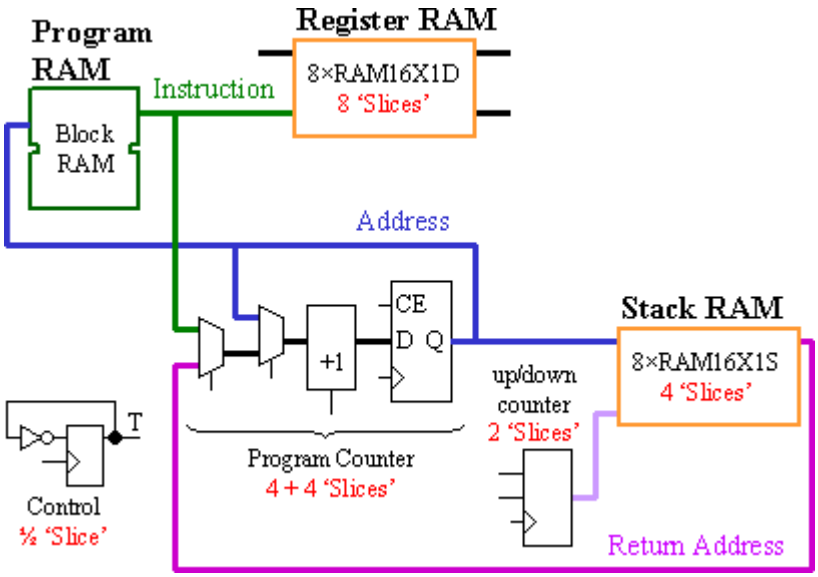
To form a stack, a 4-bit up/down counter requiring two "slices" is used to address the distributed RAM. When a CALL instruction is encountered, the current program address is pushed onto the stack by writing the memory and incrementing the stack counter. RETURN instructions will decrement the stack counter and read the program address from the distributed RAM.



Due to the two cycle per instruction format, the CALL and RETURN instruction decoding must be qualified by the T-state control such that the stack is only "pushed" or "popped" once. Although there are two clock cycles per instruction, only one cycle is available to decode a JUMP, CALL, or RETURN instruction and load the program counter. The operation of the stack should therefore be arranged such that the "top of stack" return address is presented at the output of the stack at the start of each instruction execution in case it is a RETURN operation.

RAM, RAM, and more RAM!

We can now see that the PSM employs RAM three times, each with independent address and data paths. Although the simple instruction decoding logic is not shown, the fundamental program flow control has been achieved using just 14½ "slices".



Distributed RAM has been used twice to provide a total of 256 bits of memory in 12 "slices". Using flip-flops and logic to replace these memory structures would require 152 "slices", which is more than double the size of the complete KCPSM (35 CLBs in Spartan-II provide 70 "slices"). Using block RAM to implement these small memories would be wasteful and would impact the flexibility of a PSM to be used multiple times in devices, especially small ones with limited block RAM.

Programmable Logic = Flexible Instruction Set

Another reason for leaving the definition of the ALU instructions until the end of your design process is because this is an area of great deliberation and debate. Can you imagine the long discussions and arguments occuring at companies that design and manufacture microprocessor chips? There must be enough instructions so that the chips are easy to use, but every instruction added makes the processor more expensive to manufacture. Sometimes special instructions may be included to make the processor more suitable for certain applications (such as a hardware multiplier for improved DSP performance), but such specific tuning may then preclude that processor from being adopted for other applications. Of course, any special instructions are only valuable if they are actually used, which may not always be the case if the compiler or software engineer does not invoke them when the opportunity arises.

The advantage of creating a processor inside an FPGA is that the ALU instruction set does not have to be fixed until you decide it is right for you. You may actually define a different instruction set to suit each particular application. In practice, you will probably find that defining one personalised instruction set is adequate unless you implement a very diverse range of products. Even if you discover that you made a poor choice of instruction set well into your design phase, you still have the ability to change it. However, be careful not to make changes too often as this will impact the development time of your product and detract from the advantages the PSM concept offers in the first place.

Selecting Operations

When I defined the instruction set for my KCPSM (PicoBlaze™) macros, I wanted to provide a reasonably general-purpose machine that could be applied to a wide variety of applications. My focus was also to keep the size small, so I avoided including any logically expensive operations that would be infrequently used. It was always clear to me that whatever I chose could be changed in future, so I did not loose sleep over my decisions! Provided that you have a reasonable selection of basic instructions, it should be possible to write programs (or sub-routines) to perform more complex operations. For example, repeated shift and addition operations may be used used to realise a multiplication.

Electing to include a particular operation in the ALU will probably make the PSM larger. However, it will mean that your programs should become easier to write and will execute more quickly. Generally, the

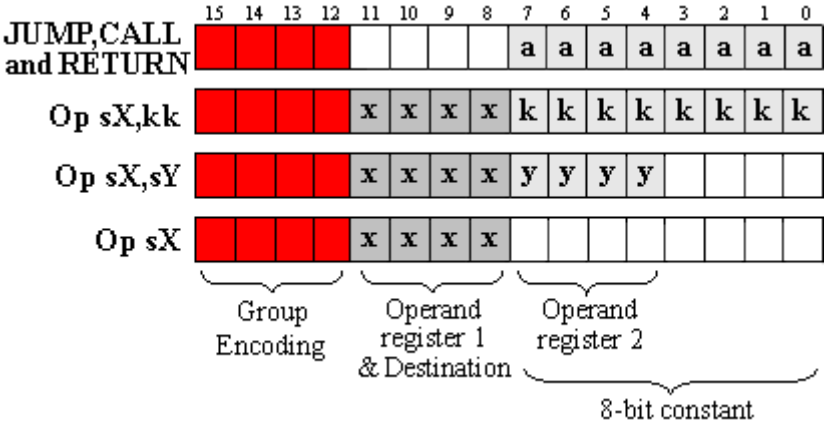
concept behind a PSM is to perform complex state machine tasks that are not particularly time critical; therefore, adding logic to the ALU so that programs will execute faster is not high on my priority list, but obviously your objectives may be different. The more significant advantage of adding a special operation is linked to the phrase that programs become "easier to write". The ability to specify one instruction instead of multiple operations not only makes programming easier, but implies a smaller program. Since the most restrictive aspect of a PSM is the length of the program stored in block RAM (256 locations in Virtex™-E or Spartan™-II, and 1024 locations in Virtex™-II), being able to write more compact code can be vital for a successful fit.

A program that nearly fills the available block RAM and executes at an acceptable rate indicates efficient use of the device via time-sharing of the logic resources. Use this to gauge whether it is necessary to include special instructions to make the code smaller and faster.

Some Constraints

The format of the ALU instructions for my PSM macro was partially defined in Part 2. The first operand will be a register (sX), which also implies the destination for any result. The second operand can be register (sY) or an 8-bit constant (kk). There are also instructions that only manipulate the contents of a single register (sX). Part 3 illustrated how JUMP, CALL, and RETURN instructions provide flexible flow control of the program. Although not strictly an ALU operation, we must also consider that the PSM processor will need to communicate externally using some form of input and output ports. My choice for these input/output (I/O) operations was to allow values to be passed to or from a register and a port specified by an 8-bit address. This has the same operands as other ALU operations except that the second operand is used to provide the port address.

Instructions may be formed into groups with common operand types:

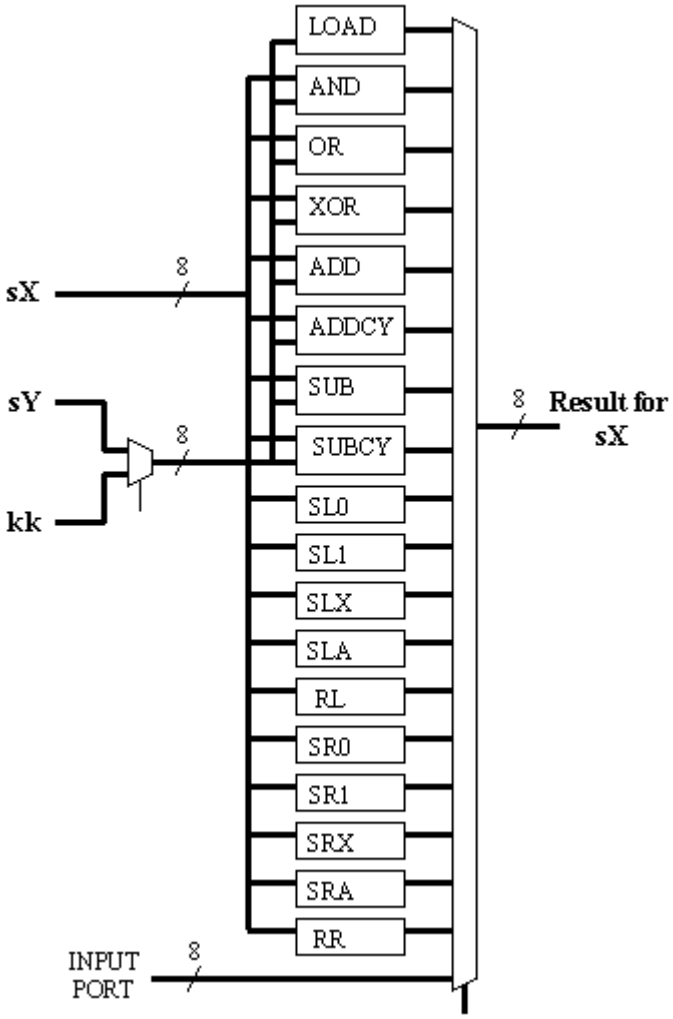


Given that the instruction words are 16-bits, it is clear that only 4 bits are available to uniquely encode the principle instruction groups. Fortunately, this does not mean that only 16 (2^4) instructions are allowed in total, because most of the groups contain "spare bits" that can be used to encode instructions within the same group. You can see that the "OPsX" group has 8 "spare bits" with the potential for up to 256 (2^8) instructions within the single operand group. The "OPsX,sY" and JUMP, CALL, and RETURN groups each have 4 "spare bits" with the potential for a further 16 (2^4) instructions in each group. Therefore, the greatest constraint comes in the "OPsX,kk" group where no "spare bits" exist.

The "OPsX,kk" is the original reason behind the decision to infer the destination register to be the same as the first operand (see Part 2). It is now obvious that this instruction group must be fully encoded by the primary 4 bits, which must also be used to identify the other groups. Therefore, the maximum number of instructions in this group is 13 (or 14 for those who really want to push it!).

Multiplexing

The key to an efficient ALU implementation is reducing the amount of multiplexing logic. In the most primitive implementation of an ALU, every operation is performed in parallel regardless of the instruction being executed at the time. The instruction decoding logic then selects the appropriate result. To make the following descriptions easier to understand, I will focus on my implementation of the KCSPM (PicoBlaze) instruction set. The 18 ALU-based instructions certainly require a great deal of multiplexing. There is also the simple 2:1 multiplexer to choose either a register or constant value for the second operand (introduced at the end of Part 2).

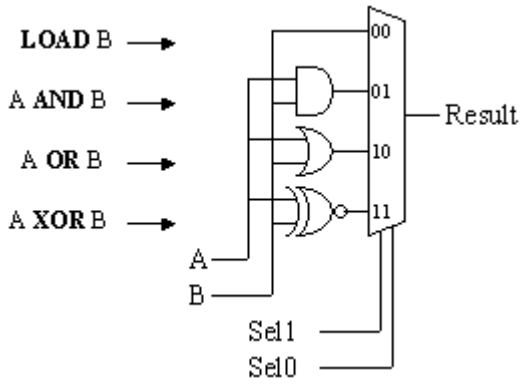


The apparent requirement for a 19-input multiplexer is alarming! Even with the dedicated multiplexers within the logic slices (MUXF5 etc), the size of this multiplexer for the 8-bit data width would be 48 "slices". This is very expensive, considering that each actual ALU function has yet to be implemented. This multiplexer would also require 5 select lines -- it would be an interesting task to encode these from the 4 primary bits and various "spare bits" of the instruction words.

Obviously, the amount of multiplexing may be reduced by decreasing the number of available ALU operations, but this hardly the most desirable solution! In Part 2, we saw that multiplexers could be very expensive in a register-based processor structure, but we avoided this situation by using the look-up tables in a RAM mode. We must now see how the look-up tables may be exploited to combine some multiplexing with the ALU operations that must be implemented.

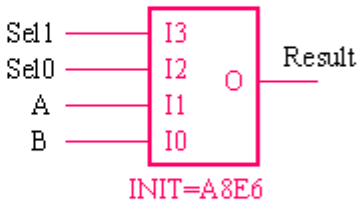
Logical Group

The logical group of instructions provide bit-wise logic operations between two operands. If the set of operations is drawn out in more detail together with a multiplexer, we make a powerful observation:



For each bit of the operands, we can see a 4-input function -- a perfect fit for a look-up table. In this way, the entire 8-bit logical group (including the multiplexer) can be realised in just 4 "slices" rather than 20 slices (4 "slices" for each AND, OR, and XOR, plus 8 for a 4:1 multiplexer). This reduction in size again provides the benefit of higher performance.

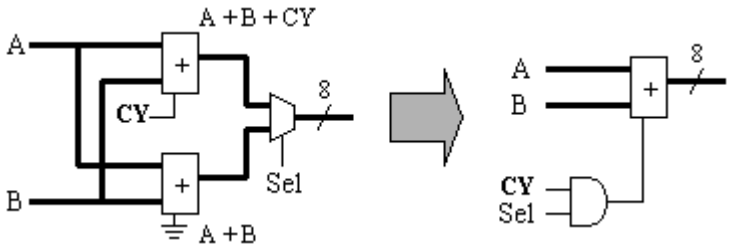
Logical Group					
I3	I2	I1	I0	O	
Sel1	Sel0	A	B	Result	INIT=A8E6
0	0	0	0	0	6
0	0	0	1	1	
0	0	1	0	1	
0	0	1	1	0	
0	1	0	0	0	E
0	1	0	1	1	
0	1	1	0	1	
0	1	1	1	1	
1	0	0	0	0	8
1	0	0	1	0	
1	0	1	0	0	
1	0	1	1	1	
1	1	0	0	0	A
1	1	0	1	1	
1	1	1	0	0	
1	1	1	1	1	



The above truth table derives the INIT value for each LUT forming the logical group. The same mapping should be achievable via synthesis tools now that we are breaking the description of the ALU down into manageable pieces.

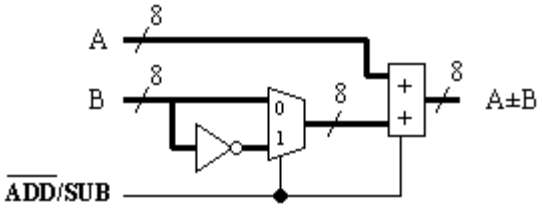
Arithmetic Group

The Arithmetic group performs an 8-bit addition or subtraction with or without the inclusion of a CARRY flag. The first reduction can be made simply by reordering the description of the optional CARRY operation:

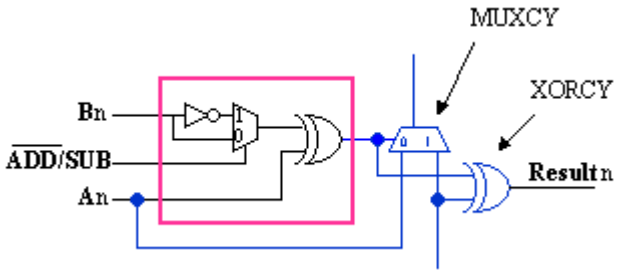


Rather than use the select signal as a multiplexer control, we now use it as a masking control to the CARRY (CY) signal. Since this is a single AND gate, it only costs a ½ "slice" compared with 4 "slices" for a multiplexer and a further 4 "slices" for a second adder. Although this is a simple observation, you must consider this when writing any HDL code.

More knowledge about the arithmetic capability of the Xilinx devices tells us that the addition and subtraction operations may also be combined. This is also a case of reordering the functions in order to place the multiplexer in front of the carry logic. Subtraction is performed in the "slices" -- the addition of the "A" input with the twos complement of the "B" input. To perform the twos complement of "B", each bit of "B" is inverted (the one's complement), and the addition of "1" is achieved by applying a carry input to the adder.

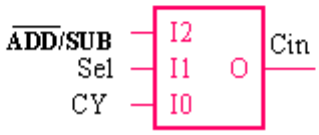


Making the complement of the "B" optional requires a multiplexer, but there is no problem absorbing this into the function generator, which performs the "half add" preceding the dedicated carry logic. This rather complex-looking function generator reduces to a 3-input function and is more typically represented as a 3-input XOR gate.



The input to the carry chain must combine the effects of the twos complement addition of "1" with the optional addition of the CARRY flag. This reduces to a 3-input function and therefore occupies no more space that the simple masking AND gate required for the optional carry input adder. The most complicated cases of this table involve subtracting. The simple SUB operation means that the input to the carry chain must be forced High to provide the twos complement addition of "1". When performing a SUBCY operation, the CY input must be inverted such that when CY=0, the twos complement addition of "1" still takes place. When performing a SUBCY operation with CY=1, the the inverted CY prevents the twos complement addition of "1" so that the result is the required A-B less one (i.e., A-B-CY where CY=1).

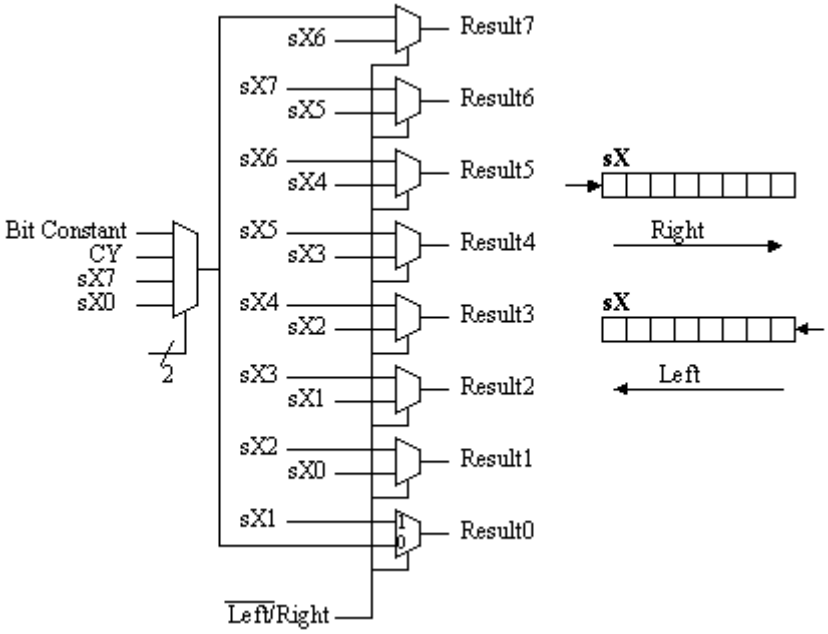
Instruction	$\overline{\text{ADD}}$/SUB	Include CARRY	Carry Chain Cin
ADD	0	0	0
ADDCY	0	1	CY
SUB	1	0	1
SUBCY	1	1	not CY



Now, the entire arithmetic group is implemented by just 4 ½ "slices" rather than 24 "slices" (4 "slices" for each add or subtract, plus 8 for a 4:1 multiplexer).

Shift and Rotate Group

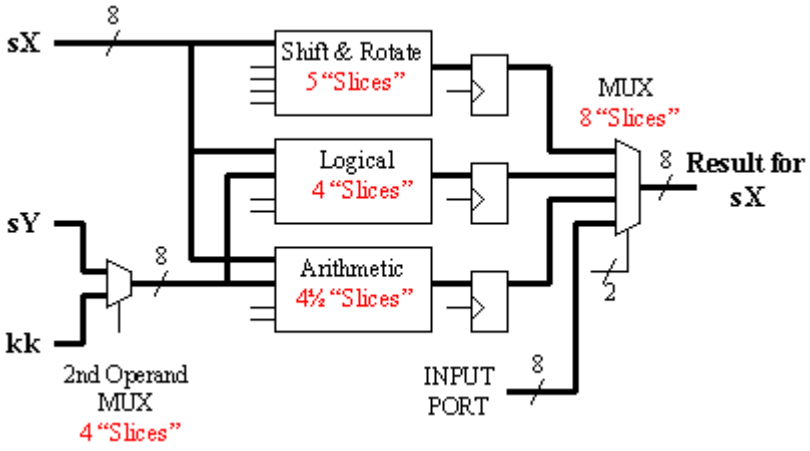
The shift and rotate group is a simple case of selecting the correct bits via multiplexers; it is obviously reduced to the following structure:



Anyone familiar with KCPSM (PicoBlaze) may have wondered why the shift and rotate group is so comprehensive. When examining the above structure, we can see that all the operations share the same 2:1 multiplexer created in 4 "slices". A single "slice" is then required to select the information to be injected into the MSB or LSB of the result. The multiplexer select lines and the "Bit Constant" are easily provided using 4 of the 8 "spare bits" available in the "OPsX" group.

Completing the ALU

Some final multiplexing is required to combine the three main ALU instruction groups and provide an input data port. This final 4:1 multiplexer is efficiently implemented by 8 "slices", including the MUXF5 dedicated multiplexers. If you create your own ALU, this final multiplexer is always worth careful consideration. The 4:1 is a perfect fit in the Xilinx architecture. There is nothing to be gained from having a 3:1 multiplexer, but a 5:1 would be larger and add delay with associated lower performance. The next natural fit would be an 8:1 multiplexer at a cost of 16 "slices".

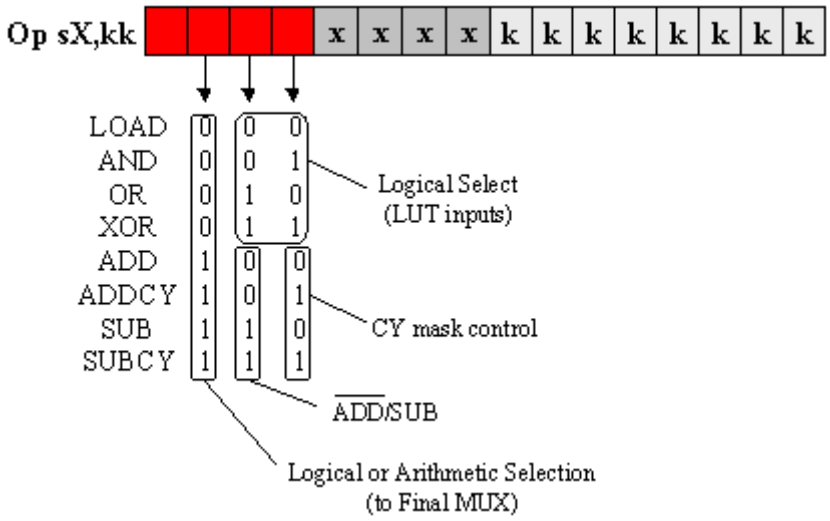


In Part 3, we saw that the PSM would work with a constant 2-clock cycles per instruction. This means that the ALU has 2 clock cycles in which to complete any operation. The path through the ALU would certainly become the critical delay path and set the performance of the the PSM as a whole. Although time specifications could be applied to indicate these 2-cycle paths, there are plenty of flip-flops available to insert a pipeline stage and simplify all time specifications to the normal clock period.

ALU Details

The size of an ALU will depend on the instruction set that you define. However, I hope you can apply similar techniques that I have used to implement the KCPSM (PicoBlaze) ALU in 25½ "slices". Remember to see which functions can be reordered and combined to minimise the actual number of logic blocks working in parallel and reduce the size of the final multiplexer.

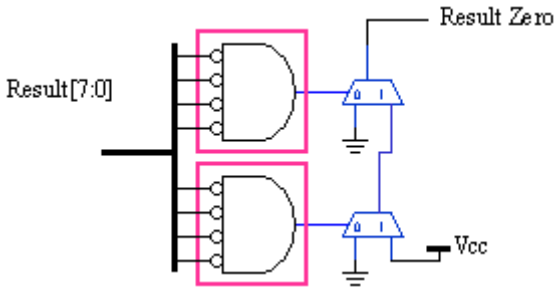
All the reduction also helps to simplify the task of decoding the instruction words and controlling the select lines on each of the ALU blocks. Indeed, if you are allocate the instruction encoding carefully, many of the bits can be applied directly to the ALU blocks, as illustrated by the most constraining "OPsX,kk" operations of KCPSM (PicoBlaze).



The ALU also generates flags used in some operations (i.e., ADDCY) and program flow control. KCPSM has a CARRY flag and a ZERO flag, but you may decide that other flags are more useful in your own PSM processors. Each flag may be implemented using a simple clock enable flip-flop to ensure that they are only updated during appropriate ALU operations.

The CARRY flag is forced to zero during logical operations and captures either the carry chain output during arithmetic operations, or the bit "shifted out" during shift and rotate operations. The CY flag is therefore driven by a simple 1-bit multiplexer.

The ZERO flag is set when all bits of the result from any operation are low. This is a simple 8-bit NOR operation and can be efficiently implemented within a single "slice". As well as saving a LUT, the use of the carry chain in this way ensures a minimum delay. Delay in forming the ZERO flag value can be critical, as it follows the final MUX of the ALU.

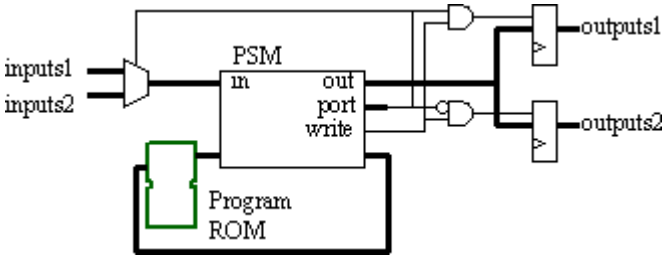


Extending Program Size

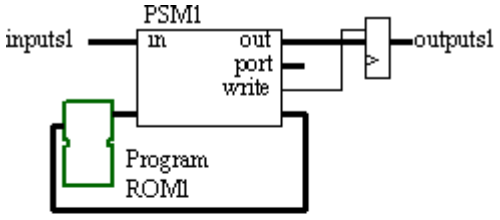
In my PSM macros, I chose to utilise a single block RAM to store the program. This obviously restricts the size of the program to 256 instructions in Virtex™-E and Spartan™-II devices when we use the 16-bit aspect ratio, which enables the operands to be included with each instruction. Many of you have told me how you have reached this memory limit; therefore, it seems likely that you will elect to have a larger program space if you implement your own PSMs. I will therefore focus on techniques suitable for supporting these larger programs.

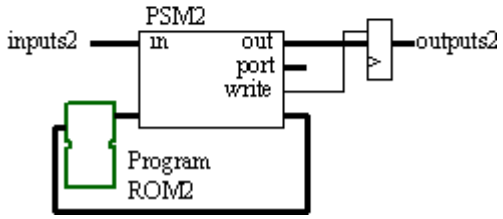
Before we take the "easy" option, let us consider ways to work with the PSM structure we have studied so far in this series. It is not unusual to hear from PicoBlaze™ (KCPSM) users who have included two or three processors in a single design. They realise that distributed processing is the solution, which means that

each PSM processor then has its own program memory. Although this may sound extravagant at first, this approach reduces the amount of I/O interface logic and has the advantage of totally independent code, which is easier to develop and test. The following diagram illustrates the interface logic to work with just two simple inputs and outputs:

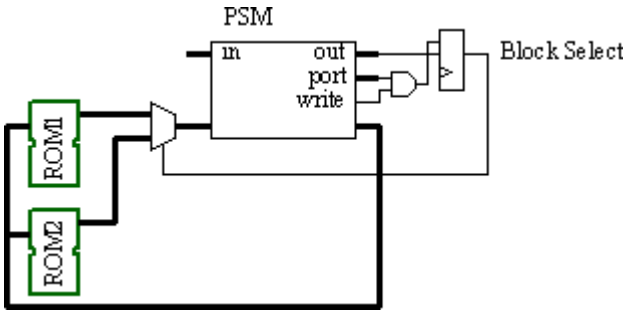


The diagram below shows how the use of two PSM macros simplifies the I/O interface. It may also help with the layout of a design in a large device. (After all, if you were implementing hardware state machines, they would be implemented separately.)





Others have also reported success through implementing a form of memory-swapping under software control. Typically, a sub-routine located at the same position in each program ROM controls the selection by writing to an output port. In practice, it is highly likely that other sections of code will need to be repeated in each of the memory blocks. Hence if two blocks are used, this will not actually yield twice the available program space. This technique does allow a common PSM macro and support tools to be used in a wide range of applications requiring small or "large" programs.

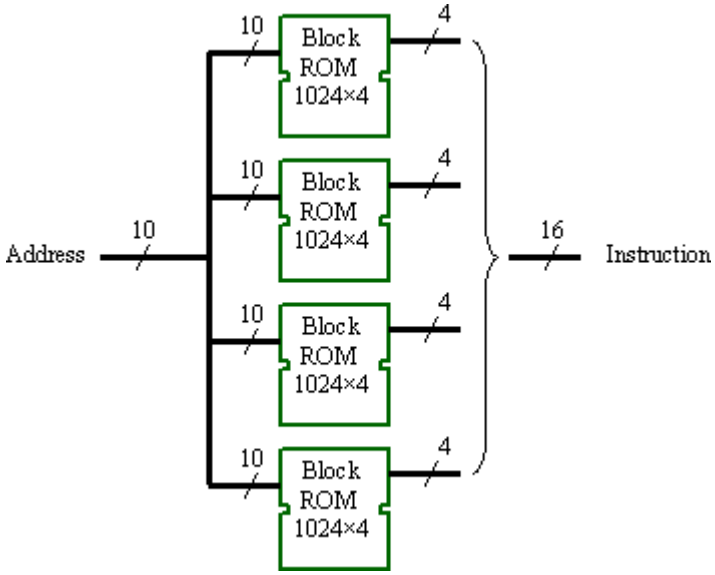


The multiplexer in the above diagram could be saved by using the block RAMs in 512×8 aspect ratio, then using the select bit to address the MSB (9th) address bit. However, the instructions will then be split across the RAM blocks, and programming the ROMs will become more challenging!

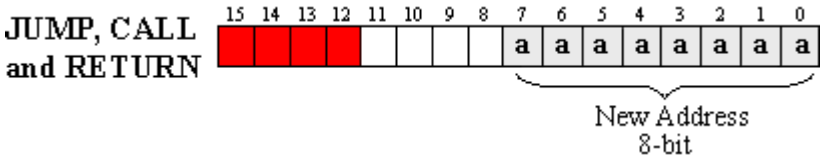
Larger Program Memory

Although block-swapping is possible, it does distract from the ease of use that a PSM should offer to a designer. So if a larger program memory really is going to make things easier, we should just consider the impact and use the best techniques.

Larger program memory may be constructed by combining several block RAMs. Using CORE Generator™ is probably the easiest way to construct such a memory, given the need to initialise the contents of the memory with the program. Each instruction code will be split across the blocks rather than using the memory in "block pages". Larger memories constructed in this way require no additional logic, as the multiplexing and address decoding is achieved directly by the block RAMs. In the diagram below, a memory of 1024 locations of 16-bits is implemented. Consider the impact of using the same block RAMs organised in 256×16.

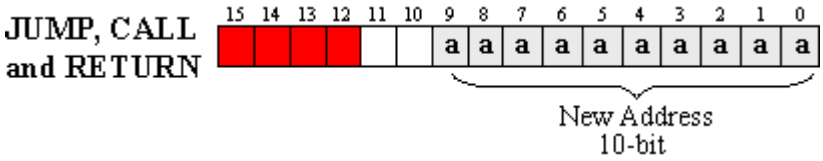


Increasing the size of the program memory sounds like an easy thing to do, but the PSM is now required to support the extended address range. To increase from 256 locations to 1204 requires that the program counter grow from 8 to 10 bits. Likewise, the width of the CALL/RETURN stack must grow. However, these only increase the size of the PSM by a few "slices". The real issue is determining how to specify the address operand in the JUMP and CALL instructions.



In Part 4, we examined the pressure that existed on the four "primary" bits of the 16-bit instruction word. For this reason, the "spare" bits associated with the flow control group will be used to further encode these instructions. Not only must we distinguish between JUMP, CALL, and RETURN, but we also must encode if the instruction is conditional. This was a pretty tight fit in my own KCSPM design, which allows unconditional and conditional instructions with conditional tests for zero, not zero, carry, and not carry. This leads to a total of 15 combinations, which obviously puts pressure on the 4 "spare" bits.

Now, consider what happens when the address operand grows to 10-bits...



It is clear that there would be no way to encode all the desired instructions in the two "spare" bits that remain. Retaining a 16-bit instruction format would either require a reduction in the number of conditional flow control instructions or place more pressure on the primary bits, causing a potential reduction in ALU instructions. Neither of these is desirable; we should also remember that the more encoded the instructions become, the larger and slower the PSM decoding logic will become.

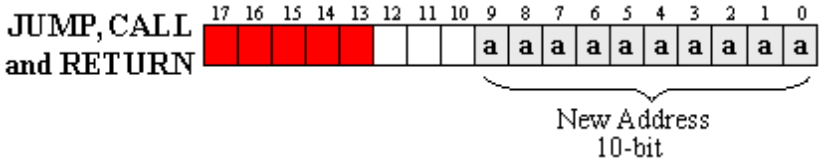
Traditional processors solve this same issue using a variety of methods:

***Operand Fetch*:** Rather than keep each instruction fully self-contained, operands are obtained from the program memory as required by using subsequent memory locations. This uses up some of the additional memory space we are trying to provide and leads to fetch cycles that complicate the PSM internal state machine.

Relative JUMP: This limits the distance you can "jump" to an address relative to the current location. An 8-bit address operand is typically a twos complement value that allows the program counter to be increased by up to 127 and decreased by up to 128. This works well with small routines, but doesn't make movement between routines very nice. Calls to subroutines outside the range would be impractical. Programming of this type of processor really requires an assembler supporting labels, and the program counter logic must implement a signed addition.

***Pre-Fetch Instruction*:** An instruction is used simply to provide operand information that is then available for use by a subsequent instruction. The operand is loaded into a holding register inside the processor. Once again, this requires a memory location and additional logic inside the processor.

These methods are ideal when the operands can be large in comparison to the available program memory width. They were the only sensible methods to use for full 8-bit data and 16-bit address range processors using external byte wide memory. However, a PSM is intended to be 100% embedded, and it is good to further exploit the flexibility of the devices and the "virtual pins" that the embedded state offers. Since we are only trying to make the address operand a few bits longer, a solution would be to make the program memory wider. At 18-bits wide, the additional 2-bits are provided, and the number of bits available to encode the instruction is restored.



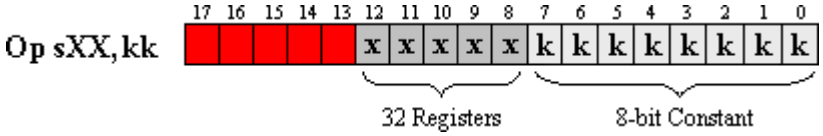
This is not an obvious solution unless you are familiar with Xilinx FPGAs, and it still may not be obvious now. People tend to think in terms of 8, 16 and 32 bits when it comes to processors and memory, so a memory width of 17, 18, 19, or 20 bits sounds strange. In Part 1, we opted to make all instructions 16-bits wide, as that was the widest aspect ratio of the Virtex-E and Spartan-II block RAM. To increase the width by a few more bits does not seem to be a naturally good fit until you combine this with the fact that a larger program memory is implemented by joining multiple block RAMs together, each configured in a deep but narrow aspect ratio.

Hence, five block RAMs in 1024×4 mode implement a 1024×20 program memory. The additional 4-bits now available to describe each instruction will also help reduce the pressure on the primary encoding and make the PSM smaller, faster, and easier to design. Alternatively, they may be used to provide the PSM with more features and instructions.

Virtex-II

Rather fortuitously for PSM macros, the block RAMs provided in Virtex-II devices are four times bigger than those in Virtex-E and Spartan-II devices, and generally provide adequate memory for PSM-based applications. But the good fortune doesn't stop there. In addition to supporting a 1024×16 aspect ratio on the main data port, these blocks provide an additional bit for each byte of data, with the intention of storing parity information. This means that the memory is actually 18-bits wide when 1024 locations are provided.

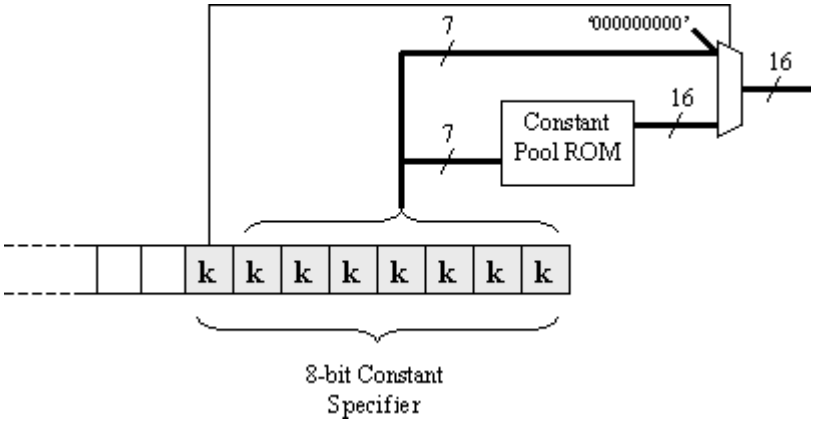
In my PSM macro that is specific to Virtex-II (KCPSM-II), which supports the 1024 locations of program space, I also used the additional bits to increase the number of internal registers to 32 (requiring 5 bits to identify) and make the instruction encoding easier. The most demanding "OpsXX,kk" instruction then had 5 primary bits remaining for instruction encoding.



Fetch without Fetch!

There will still be occasions when a PSM would benefit from the ability to specify large operands. Personally, I think it would cease to be a "programmable state machine" if the program length exceeded much more than 1024 instructions. However, it may make sense to implement a PSM with more than 8-bit data, and as previously discussed, that would probably mean a leap up to 16-bits. This really makes it impractical to include constant information within an instruction word because the program memory would have to become so wide that the additional bits would not be used in other instruction groups. It appears that we should return to the concept of a fetch cycle to obtain constant values when required, but we can modify this in the FPGA implementation.

CONSTANT POOL - Given the relatively small size of PSM programs, only a limited number of constants will be in use. Even when the data bus supports 16-bit values, further study of typical programs reveals that many constants are small values (such as "0" used for clearing registers and "1" used when a software counter is incremented). A "constant pool" is an additional memory in which constant values required by the program are stored separately from the program. The operand of the instruction is then used as an index address to this memory in order to locate the required constant. Small constants can still be contained directly in the instruction so that the constant pool may be kept small. Distributed memory is ideally suited to this function.

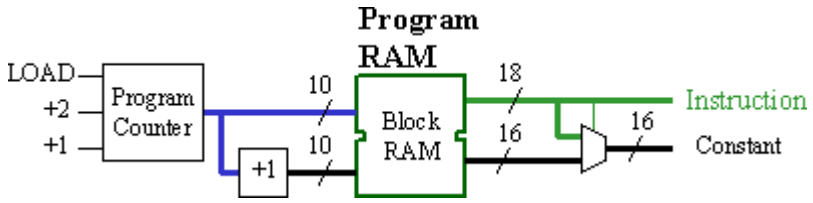


In the above diagram, the MSB of the constant operand is used to specify whether the lower 7 bits should be used directly to supply the constant value in the range 0-to-127, or used as an address to the constant pool to access 16-bit constants in the range 0-to-65535. It is useful to have an assembler to identify the constants that exceed the limit of 127 and build the indexed table of values automatically. Programming requires that both the program memory and the constant pool memory be initialised.

A trade-off occurs when the size of a constant pool is selected. In this example, although 7 bits are available for addressing the constant pool, it is likely that only 4 or 5 bits would be required to address 16 or 32 locations in a distributed memory. If the pool is too small, it is possible that certain programs will require more values than can be stored. If the pool is too large, the PSM will become larger, and there is a point at which the memory would be better used to make the instruction format wider.

PARALLEL ACCESS - This technique is very similar to an operand fetch cycle in that the operand is stored in the program memory location following the instruction. Although this uses program memory space, it has the flexibility to store any number of constants required by the program. As with the constant pool technique, it would still be a good idea to represent small constant values within the single instruction word and reserve the "fetch" for larger values.

The parallel access technique exploits two features of the PSM concept. First, it exploits the dual-port ability of the block RAM to read two locations of program memory simultaneously. Less obviously, but more significantly, it exploits the 100%-embedded nature in order to allow all the additional "virtual pins" that are required to connect the PSM to the program memory for a second time.



The above structure assumes that a 16-bit PSM is being formed in Virtex-II. An additional 26 "virtual pins" allow parallel access to the program memory to access operand information when required. An incrementer is used to access the location following the current instruction at all times. The multiplexer following the memory is used to select between small and large operand values. The program counter must be enhanced to allow it to jump forward by two addresses in the cases when a large operand is stored.

Software Support

If you design your own PSM, it won't be very long before you are frustrated with manually encoding machine code for it to execute. Besides the time consuming effort of programming in this way, it is also rather prone to errors. As with any complex state machine, a PSM has a potentially large number of "illegal states" that correspond to all the unused instruction word combinations. It is likely that you will need to develop and debug your software program anyway, so having to deal with incorrect machine code is unacceptable. It is therefore important that you also invest some time in providing an assembler to accompany your PSM macro.

Given the restricted size of the program memory, and the anticipation that a PSM works very closely with the hardware that surrounds it, it is my opinion that an assembler is highest level of abstraction that should be considered when programming a PSM. On all occasions that I have been asked if there is a C compiler available for PicoBlaze (KCPSM), a discussion about the application has rapidly revealed that a PSM would not be suitable and that MicroBlaze™ would be a much better choice. Since MicroBlaze is a 32-bit RISC processor it is fully supported with a C-compiler and indeed this would become the most suitable way to write programs for it.

So where do you get an assembler for the PSM you have created? Well I understand that there are some assemblers on the market that can be tuned to a given instruction set, but it may just be fun to write one yourself! Even though my software skills are limited and I am more comfortable writing assembler for a PSM, it only took a couple of days to produce a simple assembler for KCSPM (written in Microsoft QuickBASIC for DOS). If you don't feel like writing one yourself, try asking your software friends. You would be amazed how many of them can produce you something usable in a day and enjoy doing it (although it may cost you a couple of beers!).

I do not intend to explain how an assembler works here, but the principle is relatively straightforward. The majority of my assembler source code is used to identify syntax errors and provide constructive feedback. If

you are prepared to be more careful about entering your PSM assembler code, the assembler can focus on the generation of machine code only and will be much easier to write.

TEMPLATE MANIPULATION - Once the assembler has done its job, you are faced with how to get the machine code into the block RAM of the design. Template manipulation is a very easy way to accelerate this process and allows the assembler to output a file that is immediately suitable for use by the Xilinx tools. The assembler reads a template text file of the required format except that the actual data values for the program memory are replaced by a special string of characters. The assembler then identifies these special strings and replaces them with the actual data before writing the modified file out.

The most simple template file for a PSM is a coefficient file for use with the Core Generator. This really only needs to have the "memory_initialisation_vector=" string at the start, but may also include other configuration information used to define the "single-port Block memory" or "dual-port Block memory" cores more completely (e.g., "width_a=16;" and "depth_a=1024;"). This flow requires that CORE Generator be executed each time a change is made to the program, but it is the easiest method when creating larger memories from multiple blocks.

If you are prepared to put a little more effort into formatting of the data values, then a template file could be of a type that is even more readily integrated into the Xilinx ISE tool flow. In this case, the template could be an EDIF net list or VHDL description in which a block RAM primitive component is instantiated. The initialisation strings would not be nice to construct manually and emphasises the benefit of CORE Generator in most cases. The following is an example of one of the 72 initialisation strings required for a Virtex-II block memory instantiated in VHDL (XST):

```
INIT_02 => X"3A01608354083A106083608B6120170A60C2C000607F1804400860D592185022",
```

I am currently investigating a template required to use the DATA2BRAM utility that would allow the PSM program to be changed directly in the configuration bit stream; this would allow very rapid code iteration cycles in the same way that MicroBlaze users enjoy already.

Applications of PSM Processors

I'm pleased to note that PicoBlaze (KCPSM) is very well-used and continues to be a popular download from the Xilinx Web site (over 9,000 in the first six months of 2002 alone). I am sure that PicoBlaze is not suitable in all cases, but this is a clear indication that there are many applications that fall into the "complex state machine" category for which timing is not so critical. It is so nice to know how many "boring" things PicoBlaze has been used to accomplish! However, other applications exist that really include "processing" as well as other innovative applications.

"Make it easy" applications - The vast majority of PSM-based applications are those in which a hardware design could be employed, but the complexity of the state machine design would make it difficult to enter in a schematic or HDL. A digital clock with a timer is not that complex in itself; however, when you begin to consider how the display should change, and how to allow the user to set the time and alarms using just two press switches...this makes it very ugly in hardware!

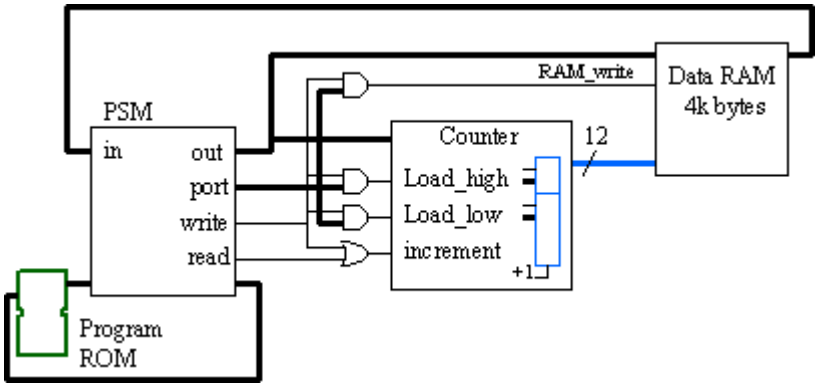
FPGAs are also used to "glue" many systems together. Spartan-II is an ideal device for this, given its low cost. However, so many devices to which you interface expect to be communicating with a processor of some kind. For example, I recently used an LCD display module of the type that can display one line of 16 characters. This module is able to respond to commands and fully understands ASCII characters.

However, the module expects commands and character data to be presented in a very specific order and with particular timing. There needed to be a delay of at least 40us between the writing of characters, but with a delay of more than 1.64ms after a command to clear the display was issued . This would all be very messy in pure hardware and would require some interesting counters in order to establish the timing. In contrast, this was very easy to write in assembler including software delay loops. So, a PSM was a natural interface to this LCD module and was an excellent way of interfacing a high-performance FPGA with a much slower component.

Including PSM macros in a design can also make other processing applications easier. MicroBlaze is a very capable 32-bit RISC processor, but if tasks are offloaded to a PSM, MicroBlaze is allowed to focus on the data processing at which it excels. As we look at Virtex™-II Pro devices with their embedded PowerPC processors, I can envision systems in which MicroBlaze and PSM processors are also included to provide a hierarchy of processing options each suited to their tasks. Distributed software processing is coming to an FPGA near you soon!

PROCESSING APPLICATIONS - It is important to remember that a PSM is a state machine and not really a data processor -- most significantly, it really doesn't have the concept of a memory map. PicoBlaze has an 8-bit port identifier that provides up to 256 input and 256 output ports. Although some of this port map could be used to access memory, again, this is a relatively limited space; therefore, data processing applications must generally be restricted to a minimal number of variables held in registers and small data sets in external memory (relative to the PSM but probably still inside the FPGA). Applications such as motor control fall nicely into this limited data space and also match well with the available performance.

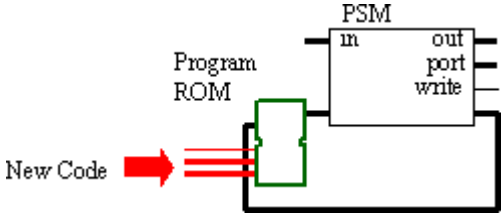
Access to larger data sets may be possible if we add hardware support and allow the PSM to act as a controller. In the diagram below, we see that a large data memory must be accessed by the PSM. Rather than directly address the memory, a hardware address counter that is controlled by the PSM is provided.



Registers can be used to compute start addresses from which data will be accessed (such as the beginning of text strings). This value is then loaded into the hardware address counter using a couple of output port locations. The hardware counter is then used to access the main data memory. As far as the PSM program is concerned, all reads and writes to the data memory occur at the same I/O port address. The added advantage of this technique is that the address counter can be made to increment automatically each time the PSM reads or writes information.

INNOVATIVE APPLICATIONS - These applications are innovative in their continued exploitation of the way in which a PSM is implemented inside the Xilinx FPGA. They build on the idea that there is now a software-programmable element inside a hardware-programmable device. The degree of flexibility that this offers is vast and is no longer limited to the larger and more costly devices. Now anyone with the most basic of Spartan-II evaluation boards may investigate these programmable options.

The most common observation is that the program for a PSM is stored in a block RAM. Typically, this is used as a ROM that is initialised by the configuration bit stream. This block RAM is, of course, RAM; therefore, the contents can be modified. The key factor here is that the memory is dual-port, which allows the PSM to read one port whilst the other is available to modify the code.



Although we could enter the scary world of self-modifying code, applications tend to be based on a completely new program being loaded into the memory for the PSM to execute. These may be the appropriate programs for handling different data types, protocols, or standards. This again means that the program space does not have to be very large and provides a method to support new protocols and standards in future.

Probably the most innovative concept is that of using the PSM as a sequencer. Each program becomes a one-shot event without the normal repeating loop. New execution sequences are then loaded into the program memory, possibly several hundred times per second. Since there would be so many different sequences to generate, and each sequence is by its very nature a sequential process, these sequences are much easier to develop in the software environment.

Closing comments

Thank you for following this series of articles -- I hope it has inspired you to either create your own programmable state machines or simply to use the ones that I have made available.

I have always found it interesting to study the implementation of a processor, as it consists of so many common digital logic building blocks. When memory, registers, multiplexers, adders, logical functions, and decoding logic can be efficiently implemented for a processor structure, it is a firm indication that similar techniques may be employed to implement virtually any digital function. It would be nice to think that you will also be able to exploit the Xilinx FPGA resources in similar ways in your own designs in future.

Although this is the last article of this series, please do continue to send me your e-mails to discuss this subject further, or simply to share your own PSM designs and applications of PicoBlaze (KCPSM) with me.

If you want to have a look at KCPSM (PicoBlaze), it is downloadable at the address below. (Full documentation and an assembler are also available for your use.) Over 1700 copies of KCPSM were downloaded from the web site in May, 2002.

XAPP213 "8-bit Microcontroller for Virtex Devices"
http://www.xilinx.com/xapp/xapp213.pdf
ftp://ftp.xilinx.com/pub/applications/xapp/xapp213.zip

This PSM is suitable for all Virtex, Virtex-E, and Spartan-II devices. If you would like a PSM specially tuned to the Virtex-II architecture, drop me an e-mail at ken.chapman@xilinx.com and I will be pleased to send it to you.

NOTE: Xilinx has recently given the KCPSM reference design the name "PicoBlaze," to indicate the complementary nature of the Programmable State Machines with the high performance 32-bit RISC soft processor called "MicroBlaze" that was released in October 2001. With PicoBlaze and MicroBlaze™, designers now can choose from a range of "right-sized" solutions, from 8 to 32 bits.