Design of A Programmable State Machine for Packet Processing

Wangyang LAI(laiw@agere.com), Chin-Tau LEA(eelea@ee.ust.hk), Hoi-Chi Huang(eehuang@ust.hk)

Abstract—The recent trend in packet processing is to use packet processors. Although packet processors are programmable and can handle protocol changes easily, their complexity and cost is also very high. Some tasks only need mild programmability that can’t justify the use of a commercial packet processor, and a different approach is needed. A Programmable State Machine (PSM) described in this paper represents such a scheme. It is intended to replace any FSM that may require programmability later on.

Index Terms— Network processor, Packet Processor, PSM, RISC

I. INTRODUCTION

The Internet is changing rapidly and more related protocols have been added—ATM, IP, MPLS, etc. Some protocols are also evolving and new features are created constantly. To introduce products into the market quickly, many companies have adopted a common hardware platform for a variety of products. Specialized processors tailored for packet processing are used to handle multiple protocols and feature changes. A packet processor usually has multiple RISC engines inside and these processors can be configured as several instances of parallel processors working simultaneously or in a pipelined fashion [1-7]. Sometimes multiple packet processors, interconnected with high-speed serial links [4], work together to implement the entire processing task. In either case, the complexity and cost of using packet processors is very high.

Packet processing, however, has many levels of programmability requirements. Some tasks only require mild programmability and can’t justify the use of a full-fledged packet processor. For example in the Fig. 1, the processing inside the switch fabric requires a level of programmability much less than that in the packet processor (see Fig. 6 later). It is simply not economical to use a packet processor in this case. A finite state machine (FSM) has the benefit of performance, but lacks the flexibility to adapt changes. What we need is something in between: fast, programmable, but simple. A programmable state machine (PSM) is such an idea. In the following we present a new PSM architecture to achieve simplicity and speed. We also detail one application example.

II. A CASE FOR PSM

The major components in a generic router/switch are shown in Figure 1. A line card terminates a transmission link of different types of physical media. After the framer processes the physical layer protocol, it passes the packet to a packet processor for layer 2 and 3 processing. The processing includes IP table lookup and packet classification. Packets are then stored in the Traffic Manager (TM) that handles queuing. Incoming packets are normally divided in cells in the TM for easy buffering. The cells are then sent to the switch fabric for forwarding. When cells arrive from the switch, TM will put them back into packets. So maintaining cell sequence in the switch fabric is important. Otherwise, the TM has to perform packet assembly. Line cards are linked by a switch fabric. Standards interface between the TM and the switch fabric has been proposed and one of them is the Common Switch Interface (CSIX) [8].

One way to implement the switch fabric is to use a crossbar (Figure 2). Port processors in the switch fabric buffer cells before sending them through the crossbar. The programmability issue also arises in the port processor. For example, some reserve bits are set aside in the CSIX header and different vendors may use them for different purposes. This type of programmability can never justify the use of a full-fledged packet processor. What we need is a design that is as simple as a FSM, but has a mild programmability. The PSM in Figure 3 is such an idea.

This work was supported by the grant CRC98/01.EG15 from Hong Kong RGC.

III. THE PSM ARCHITECTURE

A PSM should not sacrifice speed for programmability. Our proposed PSM adopts a pipelined architecture. By taking advantage of the characteristics of a PSM’s main function—FSM emulation—we are able to remove almost all the complexities in hazards control existing in a conventional pipelined processor.

A. Functional Blocks

The proposed pipelined architecture PSM is a stripped-down RISC type machine as shown in Figure 4. It has only four stages—Instruction Fetch (IF), Instruction Decode (ID), Executive (EX), and Write Back (WB). It does not have the Memory (MEM) stage and hazard control in our PSM is greatly simplified. The main blocks are the following.

1. Instruction Memory (I_Mem). It stores instructions. Current design only holds 128 instructions.
2. Program Counter (PC)
3. Instruction Decoder (ID). It decodes the instruction and generates control signals.
4. Arithmetic and Logic Unit (ALU).
5. Branch Arbitration Unit (B_Arb). When a branch instruction is met, ID decides the type of the branch. Based on this information and the comparison results given by ALU, B_Arb decides if branch will be taken or not.
6. Forwarding Unit (FU). Bypass logic. With this block, result of the first instruction can be used by second instruction immediately before it is actually written to register files.
7. IF_ID, ID_EX and EX_WB: Pipeline registers. They store temporary values and control signals of each pipeline stage. When the NOP (no operation) instruction in the instruction set is executed, the values in these registers remain unchanged for one cycle.

We list the main differences between our PSM and a conventional pipelined processor [9].

1. Our PSM does not have the MEM stage and the FU can be implemented with less than 100 gates (section 3.C).
2. The task for PSM is FSM emulation. I_Mem (instruction memory) rarely needs more than 128 entries. This allows for a fast IF implementation.
3. No interrupt instructions are needed in our PSM.
4. Hazard control is the PSM is simplified by the predictability of the task for the PSM—FSM emulation.
5. Registers of the PSM are divided into two groups: the internal registers and the input/output registers. The input/output registers interface with other FSMs/PSMs. Generating control signals to the outside world are done by writing the registers. The internal registers are used as general-purpose registers.

B. Instruction Set

Section 3.A provides the framework for a low-complexity PSM architecture. How to select the instruction set obviously depends on the type of task for which the PSM is intended. The task for our PSM is packet processing in the Port Processor (Figure 2). We design only 18 instructions and all instructions have a fixed length: 29 bits. If used for other applications, the instruction set can certainly be extended. These instructions are classified into three categories based on their format:

- **Register type**

<table>
<thead>
<tr>
<th>Hdr</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>Tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

- **Immediate type**

<table>
<thead>
<tr>
<th>Hdr</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

- **Branch type**

<table>
<thead>
<tr>
<th>Hdr</th>
<th>rs</th>
<th>rt</th>
<th>target</th>
<th>Tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

When we classify these instructions in terms of usage, they are:

- **Arithmetic and Logic Instructions**

  - add DestReg, SrcReg1, SrcReg2 :Addition
  - addi DestReg, SrcReg, Imm :Addition with immediate number
  - and DestReg, SrcReg1, SrcReg2 :Logical AND
  - andi DestReg, SrcReg, Imm :Logical AND with immediate number
  - or DestReg, SrcReg1, SrcReg2 :Logical OR
  - ori DestReg, SrcReg, Imm :Logical OR with immediate number
  - sll DestReg, SrcReg, Shamt :Shift logic left
  - srl DestReg, SrcReg, Shamt :Shift logic right
  - xor DestReg, SrcReg1, SrcReg2 :Logical XOR
  - xori DestReg, SrcReg, Imm :Logical XOR with immediate number

- **Constant—manipulating Instruction**

  - li DestReg, imm :Load immediate number

- **Branch Instructions**

  - beq Reg1, Reg2, LABEL :Jump to Label if (Reg1==Reg2)
  - beq Reg1, Reg2, TargetReg :Jump to addr given by TargetReg if (Reg1==Reg2)
  - bgte Reg1, Reg2, LABEL :Jump to Label if (Reg1>=Reg2)
  - bgte Reg1, Reg2, TargetReg :Jump to addr given by TargetReg if (Reg1>=Reg2)
C. Data and Control Hazard Removal

In a general-purpose RISK processor, hazard removal has a high complexity. But this is not the case with our PSM, as shown below because the processor is designed to emulate a FSM and to perform a fixed function. There are two types of hazards in the pipeline processor: data and control hazards.

1) Data Hazards

Data hazards are checked in the forward unit. Consider two instructions N and M, with N occurring before M. The possible data hazards are:

- RAW (read after write)—M tries to read a source before N writes it, so M incorrectly gets the old value.
- WAW (write after write)—M tries to write a register before it is written by N. The write ends up being performed in the wrong order, leaving the value written by N rather than the value written by M in the destination.
- WAR (write after read)—M tries to write a destination before it is read by N, so N incorrectly gets the new value.

This hazard is not present in our PSM. It is present only in pipelines where write is performed in more than one pipeline stages or in pipelines that allow an instruction to proceed even when a previous instruction is stalled. Both scenarios do not exist in our PSM (writes are done only in WB).

2) Control Hazards

Since our PSM has no interrupts, we only need to deal with branches. Again the characteristics of FSM emulation simplify the design. Consider the following example:

```
And r8, r1, r2
Add r5, r6, r7
Beq r3, r4, (Next)
Xor r9, r10, r11
......
(Next):  Addi r4, r3, 7
Xor r3, r7, r6
```

The branch instruction Beq is executed in the ALU of the EX stage. If r3 = r4, the PC is loaded with the target address—the address of the “Next” instruction. The pipeline stages IF and ID will be stalled (doing nothing) until the EX stage gives out the correct next instruction address (see table 1).

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump to Label if (Reg1 &gt; Reg2)</td>
<td>Jump to addr given by</td>
<td>TargetReg if (Reg1 &gt; Reg2)</td>
<td></td>
</tr>
</tbody>
</table>

Pipeline stall can be reduced by using branch prediction. Many prediction mechanisms are available [9]. But given the small instruction set of our PSM, we choose a simpler approach: delayed branch [9]. This technique inserts useful instructions (delay-slot instructions) after the branch instruction so as to save cycles wasted when branch is taken. Consider the following example where two NOP instructions are inserted by the compiler after branch instruction.

```
And r8, r1, r2
Add r5, r6, r7
Beq r3, r4, (Next)
NOP
NOP
Xor r9, r10, r11
......
(Next):  Addi r4, r3, 7
Xor r3, r7, r6
```

We can replace the NOP operations by the useful instructions, which may comes from

a. instructions which are in front of the branch (as shown in the following).

b. the branch-taken instructions
c. the branch-not-taken instructions.

Whatever the delay-slot instructions are, they should not change the results regardless of the branch instruction getting executed or not. Because the program in the PSM is simple and predefined, the compiler can easily find two instructions, if they exist, that can replace the NOP operations after branch. One example is shown below.

```
Beq r3, r4, (Next)
And r8, r1, r2
Add r5, r6, r7
Xor r9, r10, r11
......
(Next):  Addi r4, r3, 7
Xor r3, r7, r6
```

D. Interfacing with other PSMs/FSMs

Many chips designed for packet processing may contain tens of FSMs/PSMs. The PSM can replace each FSM that needs programmability. Those FSMs that don’t need programmability can remain the same. One such example is the DMA in Figure 6 for writing a cell into memory. There is no need to use a PSM for this task.

A PSM interfaces with the other FSMs or PSMs through registers. There are 32 registers in our PSM and each is 16-bit wide. Registers are divided into two groups: general purpose
registers and special purpose registers. General-purpose registers are used by the PSM itself. They are invisible to the external world. The special purpose registers are the interface registers. They can be further divided into input and output registers (Figure 5). The PSM can read, but not write, the input registers. The contents are changed by other FSMs/PSMs. Output registers of a PSM are used to send signals or data to other FSMs/PSMs. They can be read only for other FSMs/PSMs.

IV. APPLICATION EXAMPLE

We use cell parsing in the port processor as an application example for PSM. The TM (Figure 2) will send fixed-length packets, called cells, through the CSIX interface to the switch. Cells are queued in the port processor. Each destination has its own queue, called a virtual output queue (VOQ). The port processor is implemented with many FSMs. One such FSM is for header parsing of an incoming cell. We use this as an application example for the PSM.

Figure 9 shows the tasks in header parsing. One is to check flow-control thresholds. There are two levels of flow control: VOQ-level and link level. Each level is controlled by two thresholds (high and low mark). When the buffer level exceeds the high mark, flow control is turned on. Flow control will be turned off later when the buffer size drops below the low mark. The high and low marks for the VOQ level are denoted by CloseGateValue and OpenGateValue, and for the link level denoted by MaxTotalCell and MinTotalCell. When a cell arrives, the port processor updates the queue size and checks the high mark thresholds at both levels to see if the VOQ flow control and the link level flow control should be turned on. (Similarly when a cell departs, the port processor will check the low-mark thresholds to see if the VOQ and the link level flow control should be turned off. But this is not done in header parsing for incoming cells).

A. Traditional FSM Approach

Figure 6 shows the hardware block in a port processor for header parsing. Each incoming cell is stored in a temporary buffer. Its CSIX header is stored in a separate header buffer. The Queue Lookup Table holds queue pointers and associated flow-control thresholds for each VOQ. The table is accessed by the combination of the destination address and the priority field.

Figure 7 shows the FSM implementation. Note that for ingress cell parsing, the FSM only checks the high marks of the two flow control levels. To simplify the discussion, we do not consider multicast cells (it is an optional feature in the CSIX standard). All incoming cells are either idle cells or unicast cells.

Figure 8 shows the CSIX header in which two bytes are used for based header and four bytes are used for extension header. For idle cells, only based header is included. Figure 9 shows the flow diagram of the FSM where the VOQ_Length and the Total_Cell stores the length of the corresponding VOQ and the length of the entire link respectively.

B. The PSM approach

We can replace the FSM with a PSM. We describe the implementation and demonstrate the capability of handling protocol changes of a PSM.

1) Register definition

We construct our register file as shown in Table 2. The first sixteen registers are used as the general purpose registers. The rest are used as interface with other FSMs. For header parsing, only a small portion of the general-purpose registers is used. The cell’s header is stored in rHdr. The last bit of the rHdrV is used to indicate if the header is valid. The remaining bits are not used for this application. rCmd is the command word register. Every bit of the register represents a control signal. The exact
meaning of each bit is given in Figure 10. To PSM, rCmd is the same as the other output registers and its value is kept valid for only one cycle. The Default value is zero. The external blocks sample these bit every cycle. For example, to issue a write command to the queue lookup table, an instruction li rCmd, 0x0040 is used. WrTable bit will be asserted for only one cycle.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Register</th>
<th>Generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>rHdr</td>
<td>Header</td>
<td>r13</td>
<td>Input register</td>
</tr>
<tr>
<td>rHdVR</td>
<td>Hdr Valid</td>
<td>r17</td>
<td>Input register</td>
</tr>
<tr>
<td>rFPCpot</td>
<td>Free Cell Pointer</td>
<td>r18</td>
<td>Input register</td>
</tr>
<tr>
<td>rTtCIn</td>
<td>TotalCell In</td>
<td>r19</td>
<td>Input register</td>
</tr>
<tr>
<td>rTtCO</td>
<td>TotalCell Out</td>
<td>r20</td>
<td>Input register</td>
</tr>
<tr>
<td>rTtCin</td>
<td>VOQ Tail Pt In</td>
<td>r21</td>
<td>Input register</td>
</tr>
<tr>
<td>rCmd</td>
<td>Command Word</td>
<td>r22</td>
<td>Input register</td>
</tr>
<tr>
<td>rPri</td>
<td>Priority</td>
<td>r23</td>
<td>Output register</td>
</tr>
<tr>
<td>rDest</td>
<td>Destination</td>
<td>r24</td>
<td>Output register</td>
</tr>
<tr>
<td>rQLenU</td>
<td>VOQ Len U</td>
<td>r25</td>
<td>Output register</td>
</tr>
<tr>
<td>rQLen0</td>
<td>VOQ Tail Pt Out</td>
<td>r26</td>
<td>Output register</td>
</tr>
<tr>
<td>rHdrV</td>
<td>Hdr Val</td>
<td>r27</td>
<td>Output register</td>
</tr>
<tr>
<td>rDestV</td>
<td>Dest Val</td>
<td>r28</td>
<td>Output register</td>
</tr>
<tr>
<td>rPriV</td>
<td>Pri Val</td>
<td>r29</td>
<td>Output register</td>
</tr>
<tr>
<td>rCmdV</td>
<td>Cmd Val</td>
<td>r30</td>
<td>Output register</td>
</tr>
<tr>
<td>rCmdO</td>
<td>Cmd Out</td>
<td>r31</td>
<td>Output register</td>
</tr>
</tbody>
</table>

Table 2. I/O register definition

Fig 9 Flow diagram of the header parsing hardware

2) Program in the PSM

The program is designed in two phases. In the first phase, we produce the code following the flow control diagram in Fig 9.

| (1) | INITIAL: li r0, 0 |
| (2) | li r1, 1 |
| (3) | li r5, MaxTotal |
| (4) | li rCmd, 1 |
| (5) | beq r0, r0, (WAIT1) |
| (6) | li rCmd, 0x0060 |
| (7) | addi rQLenO, rQLenIn, 1 |
| (8) | li rCmd, 0x0006 |
| (9) | li rCmd, 0x0108 |
| (10) | li rCmd, 0x1000 |
| (11) | li rCmd, 0x0100 |
| (12) | li rCmd, 0x0808 |
| (13) | li rCmd, 0x0806 |
| (14) | li rCmd, 0x0800 |
| (15) | li rCmd, 0x0102 |
| (16) | li rCmd, 0x0101 |
| (17) | li rCmd, 0x0100 |
| (18) | li rCmd, 0x0109 |
| (19) | li rCmd, 0x0108 |
| (20) | beq r0, r0, (SOF) |

Fig 11 The code of the first phase.

The resulting program has 5 instructions in SOF subroutine (Figure 11), 1 instruction in idle subroutine, and 20 instructions in unicast subroutine. We then use standard compiler techniques to translate it into a more efficient one. These techniques include the following.

1. Minimize the number of branch instructions. This can be done by
   a. replacing the unconditional branch by the other instruction(s) if possible
   b. replacing the unconditional branch by replicating the whole target subroutine.
2. Reorganize the instruction sequence by replacing the two NOP instructions after the branch with useful instructions.

The optimized program (Figure 12) contains 7 instructions in its SOF subroutine, 3 instructions to process the idle cell and 24 instructions to process the unicast cell. Red instructions are in the delay slot, they must be executed even the branch condition of the former branch instruction is satisfied. After optimization, nearly all the delay slots of the branch instructions are filled with useful instruction. The PSM achieves the maximum performance of one instruction per cycle.

The numbers of cycles needed for different types of cell parsing are shown in Table 3.

To get an idea of the processing speed, we synthesize the circuits of each pipeline stage with the TSMC 0.13um technology. The preliminary results are: IF (1.61ns), ID (1.51ns), EX (3.80 ns) and WB (0.75 ns). The values tend to be conservative because the synthesis is done without extensive

![Diagram](image-url)
optimization iterations. If we assume the clock speed is 250 MHz, cell parsing will take 96 ns for almost all cells. Assume that the CSIX cell size is 69 bytes (64 bytes of data + 5 bytes of header), then about one cell will arrive every 200 ns for an OC-48 link (2.5 Gbps). So the implementation is more than adequate for the OC-48 rate. We can improve the speed further by using more advanced technologies, or we can use two pipelines in parallel in the PSM as we find that segments in the program are independent and can run simultaneously.

![Fig 12 The code after optimization](image)

### C. Handling Protocol Changes

Minor protocol changes often occur after the products are finished. A PSM-based design can handle the changes easily. Without loss of generality, we just show one common example experienced in packet processing. There are 12 bits in the address filed of the CSIX header. This number may not be large enough if there are many sub-ports located on each line card because each sub-port needs a separate address. One way to solve the problem is to use the reserved bits defined in the CSIX header for that purpose. We assume that the new protocol header format is given in Figure 13. For the sake of discussion, Figure 13 also includes other minor changes just to demonstrate the capability of a PSM.

![Figure 13 Header format of the new protocol](image)

With a PSM, we can easily handle these minor protocol changes. To extract the type field, we replace instruction (3) of the SOF subroutine (from the compiled program) with

```assembly
andi r2, rHdr, 0x000F
```

We still use the TYPE field as the entry address of the subroutine. The amount of shift in instruction (4) should be changed to 3 because the location of the type field is changed in the new format.

```assembly
srl r3, r2, 3
```

In the new unicast header, the destination field now occupies the whole byte 2 and byte 3, while the priority field is in the middle of the byte 4 and 5. So the corresponding instructions should be revised. Instruction (2) is replaced by the following instructions:

```assembly
andi rDest, rHdr, 0xFFFF
```

And (6)(7) should be replaced by

```assembly
andi r4, rHdr, 0x01E0
srl rPri, r4, 5
```

We thus accomplish the new header parsing without any change to the hardware platform.

### V. Conclusion

Many tasks in packet processing only require mild programmability. They can’t justify the use a packet processor. In the paper, we proposed a Programmable State Machine (PSM) architecture to handle these tasks. The architecture is simple, fast (CPI 1), and hazard-free. Its simplicity makes it a candidate for replacing any FSM that may need programmability later on. Although the application studied in the paper is for packet processing, our architecture and design techniques for achieving simplicity can be applied to PSMs intended for other applications as well.

### REFERENCES