A Simple State Transition Control for FSM Programmable Protocol Processors

Yutai Ma, Axel Jantsch and Hannu Tenhunen
Department of Electronics, Royal Institute of Technology, Sweden

Abstract- We present a FSM model for programmable protocol processors. It removes the branch instruction penalty and thus it is suitable for control intensive protocol processing. Compared to the CAM based FSM model [5], our model improves state memory access speed by half and reduces state memory cost by around 8.5%. Also, state memory utilization in our model is improved to 100%.

I. Introduction

Due to the ever-evolving protocols and services used in computer communications, it raises a demand for protocol processors with flexibility so as to make it possible to keep up with the pace of protocol research and development and thus to shorten the time-to-market. On the other hand, although general purpose computers have powerful processing capability and performance is improved every year, because they are not designed particularly for protocol processing the potential high performance provisions can not be utilized efficiently when they are applied to high speed computer communications.

Programmable protocol processors provide an alternative solution to constructing high speed computer communication systems with flexibility by using special architectures which are suitable for protocol processing. The experiment of [3] indicates that at least 3–4 times performance gain can be achieved by using special instructions for protocol processing. [4] developed an architecture model for protocol processing. Their simulation results show that an aggregate gain of approximately 10:1 is obtained compared to general purpose processors. [2, 7] developed a protocol processor for gateways. Their protocol processor gains 12-fold performance of conventional system for transmission and 7-fold performance for reception.

Meanwhile FSM processors give an alternative and promising solution to high speed protocol processing. [5] designed a programmable protocol processor based on FSM (finite state machine). The experiment achieves 28-fold performance of a general purpose microprocessor. FSM based protocol processors have many advantages over traditional processors in protocol processing. The main advantages are that the penalty of branch instructions is removed and a branch or jump instruction can be combined with other operations. These provide a possibility for constructing flexible and high performance protocol processing systems. Two drawbacks of this design are that CAM (content addressable memory) is more expensive and performance of the CAM state memory is poor compared to SRAM and state memory space utilization in this model is low.

This paper is devoted to FSM protocol processor architecture with concentration on state memory organization and efficient branch instruction implementation. Since a SRAM is used to replace the CAM to store states and instructions and a simple logic is used for FSM control, the performance of FSM based programmable protocol processors can be improved greatly in state memory access speed, area cost and state memory utilization.

Search Fields				Read Fields	
Current State	Packet Signal	User Signal	Inside Transition	Next State	HP Instr.

Fig. 1: Word Format of the State CAM.

II. Matsuda's Model: A CAM Based FSM Protocol Processor

Matsuda [5] developed a programmable protocol processor based on FSM. Two important components of the processor are a protocol state CAM (content addressable memory) and task processing elements. The CAM word format is shown in Figrue 1.

The processor accesses the CAM by using a keyword register containing a current protocol state, a received packet type, a received user signal and an inside transition condition and then uses a read out HP instruction to control operations of the processing elements. The next protocol state field updates current protocol state in the keyword register for the next CAM read operation.

The main disadvantages of this model is that CAM access speed is lower than SRAM and CAM capacity is limited. This becomes a bottleneck to the performance of FSM programmable protocol processors. Other drawbacks will be discussed in section 4.

III. A FSM Model Based on SRAM and ALU Flags Decoding

An insight to improve Matsuda's design [5] is to replace the CAM with a SRAM to speed up memory access. On the other hand, for a CAM based FSM, both current and next protocol states must be included in each CAM word. Whereas for a SRAM based FSM, next protocol state is still provided by each SRAM word, but the field of current protocol state is not needed any more and thus the memory cost can be reduced. Another motivation is to improve the state memory space utilization.

A. FSM Processor Architecture and Word Format of The SRAM State Memory

An obstacle to the performance of SRAM based FSMs is how to generate addresses for different sate transitions from a protocol state. We want the address generation to be simple enough so that we can integrate it into address decoding of the state memory.

The word format is shown in Figure 2 and the FSM model is shown in Figure 3. The instruction field controls ALU operations, memory read/write, and I/O operations. The next state field specifies a base address to the state memory for next state memory read operation. We use ALU flags and the control field to capture a state transition or a branch/jump instruction. Since protocol processing is full of control and branch instructions, it deserves combining a branch or jump instruction with other ALU or memory read/write operations. The control field is used to specify a condition for state transition. Detecting multiple conditions simultaneously at a time is supported. For example, a 4-bit flag register can represent four status flags simultaneously. Also, it can support a case statement with up to 16 branches. The control field works with the branch bits to generate an n-bit address offset for branch target instructions, where n is the bit-width of the branch field. In this illustration, only one bit is used. Therefore, a base address specified by the next state field and a generated branch target address offset form address to the state memory. We discuss it in detail in the next subsection.

Control Field	1-Bit Branch	Next State	Instructon Field

Fig. 2: Word Format of State SRAM.

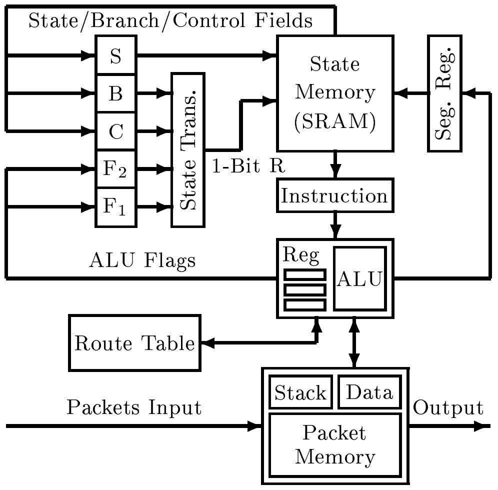

Fig. 3: A FSM Model for Programmable Protocol Processors Based on SRAM and Flags Decoding.

B. Address Offset Generation of Branch Target Instructions

Condition evaluation for state transition and branch instruction (block "State Trans." in Figure 3) is illustrated in Figures 4 and its generated address offset of branch target instructions is shown in Table 1.

When a control bit is set to "1", current state will transfer to a next state only when the corresponding

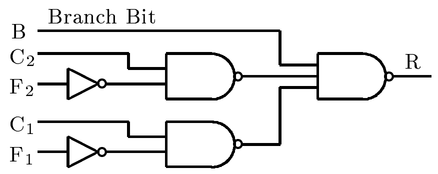

Fig. 4: State Transitions Evaluation.

Table 1: Branch Target Address Offset

Control Field	Cond. Eval.	1-Bit Branch	R: Branch Target Address Offset
> 0	True	"1"	"0"
	False	"1"	"1"
$= 0$	–	"0"	"1"
	–	"1"	"0"

flag is set up, in this case the branch bit should be set to "1". If specified conditions are satisfied, a bit of "0", otherwise a bit of "1", is generated. If a control field is set to "0", no condition is required for the corresponding flag. For multiple possible state transitions from a state, we use multiple branch instructions to implement them. If no control bit is set to "1", then no condition is required for this state transition. In this case the branch bit can be either "0" or "1". This is useful for unconditional jump instructions and tasks which cannot be completed in one instruction cycle. This insight is illustrated in Table 1.

As shown in Figure 3, the branch target address offset (R in Figure 4) and the next state field form address to the state memory for next protocol state. Compared to considerable performance improvement on branch instruction execution, it deserves combining this state transition evaluation circuit with address decoder of the state memory.

It is worth noting that to clear ambiguity among multiple flag pattern matches in a state when the bit-width of address offset is less than that of the control field (here, the bit-width of the address offset is one), the pattern with more '1's should be matched first. For example, for flag patterns of "11" and "10", pattern match of "11" should be done first.

C. State Memory Capacity

To reduce the state memory complexity, the field length of next state is restricted. For example, with 10-bit representation of the branch bit and next state fields 1k instructions can be accommodated. To accommodate more instructions/programs into the state memory, a virtual memory management unit like the one used on general purpose microprocessors can be used. We can also choose a simplified form by using a segment register to indicate a memory segment as shown in Figure 3. This segment register can be set up by a specific ALU operation. When a program needs to roam into another segment, an instruction is used to update the segment register and a following instruction use its own next state field and the segment register to fetch next instruction in another state memory segment. Since jumping across segments is much less frequent compared to normal operations, this overhead is light.

D. A FSM Program Example

We use a send subroutine of ARQ protocol [6] to illustrate how our model works. To simplify this illustration, we do not focus on these statements implementation but assume that some statements are implemented by one instruction. The send subroutine is described below.

```
s.kind = fk;                    /* Instruction 1 */
s.seq = frame_nr;
s.ack = (frame_expected+MAX_SEQ) %
        (MAX_SEQ+1);
stop_ack_timer();               /* End of Instr 1*/
if fk = data then               /* Instruction 2 */
    s.info = buffer[frame_nr % NR_BUFS];
    start_timer(frame_nr % NR_BUFS);
end if;                         /* End of Instr 2 */
if fk = nak then                /* Instruction 3 */
    no_nak = false;
end if;                         /* End of Instr 3 */
to_physical_layer(&s);          /* Instruction 4 */
```

Assume two flag bits are used to represent the frame type data, nak and ack and they are encoded as "11", "10" and "01" respectively. The program on our model is shown below, where we assume that this subroutine returns to an instruction in address "011".

FSM Program for the ARQ Send Subroutine

Addr.	Instr.	Control	Branch Bit	Next Sate
000	Instr1	11	1	01
001	Instr2	10	1	10
010	Instr3	00	0	10
101	Nop	10	1	10
110	Instr4	00	1	11

IV. Performance Comparisons

A. Comparison with Matsuda's CAM Based FSM Model of Protocol Processors [5]

In [5] the state memory is constructed by using a CAM. From the principle of CAM [8] we know that memory access speed of SRAM is twice that of CAM. Therefore, the operating speed of our model can be twice that of Matsuda's model [5].

For Matsuda's model [5] both current state and next state fields are included in each CAM word. For our model only the next state field is included in each SRAM word. Matsuda's design contains 500 CAM words by 70 bits, totally 12 bits are used to represent the current state and next state. Our model reduces the state memory cost by around 8.5%.

Furthermore, our model supports complex and large scale protocol processing by using a segment register or a virtual state memory management unit. While due to restrains on CAM capacity, capability of Matsuda's Model [5] is highly restricted.

Another problem in Matsuda's Model [5] is poor memory space utilization related to state transitions and branch instructions. For example, when using the inside transition field as shown in Figure 1 to implement a branch instruction, all possibilities of the packet signal and user signal fields must be taken into account and thus these branch target instructions may occupy 8 CAM words, instead of 2 SRAM words in our model.

B. *Comparison with High Performance General Purpose Microprocessors*

Protocol processing is full of control operations. The control information appear in packets and other data structures or actions, for example routing tables and TCP flow and congestion control. Therefore, the execution efficiency of branch and jump instructions has a severe impact on protocol processing performance. From Figure 3 and Figure 4 we see that the main advantages of our model over general purpose processors are that the hazard of branch and jump instructions is removed and case statement is supported. On the other hand, in our model branch and jump instructions can be combined with other operations. These advantages lead to a high efficiency of protocol processing.

We focus on branch and jump instructions issue here, however our model also shares other properties of general purpose processors. For example, virtual memory management and interrupt handle mechanisim. This means that our model can provide flexiable and powerful processing capability as general purpose processors while suitable for protocol processing.

V. Conclusion

FSM is an effective model for protocol description and protocol processing. In this paper we have proposed an architecture for FSM programmable protocol processors. Compared to general purpose processors, our model removes branch instruction hazard. On the other hand, we see that compared to the CAM based FSM model the state memory access time can be reduced by half by using our model, meanwhile the memory cost is reduced by around 8.5% and state memory space utilization is improved to 100%. Our FSM model can be used for constructing complex protocol processing systems by using a virtual state memory management unit or by using multiple such protocol processors.

References

[1] Ilija Hadzic, Jonathan M. Smith and Williams S. Marcus, "On-the-fly Programmable Hardware for Networks", *Proceedings of Globecom'1998.*

[2] Tetsuhiko Hirata, Susumu Matsui and Tatsuya Yokoyama, "A high speed protocol processor to boost gateway performance", *Proceedings of Globecom'1990.*

[3] Axel Jantsch, Johnny Oberg and Ahmed Hemani, "Is there a nich for a general protocol processor core?", *Proceedings of Norchip'1998.*

[4] Baiju D. Mandalia, Mohammad Ilyas and Eduardo B. Fernandez, "Performance evaluation of the communications protocol processor", *Proceedings of IEEE ICC'1990.*

[5] Takao Matsuda, Kazuhiro Matsuda, "A new protocol processing architecture for high-speed networks", *Proceedings of Globecom'1996.*

[6] Andrew S. Tanenbaum, "*Computer Networks*", pp.216–217, Third Edition, Prentice-Hall, 1996.

[7] Matsuaki Terada, Tatsuya Yokoyama, "A high speed protocol processor to execute OSI", *Proceedings of INFOCOM'91.*

[8] Neil H. Weste and Kamran Eshraghian, "*Principles of CMOS VLSI Design: A Systems Perspective*", Second Edition, Addison-Wesley, 1993.

[9] Naoaki Yamanaka, Eiji Oki, Haruhisa Hasegawa, "User-programmable flexible ATM network architecture: active-ATM-experimental results", *Proceedings of the Third IEEE Symposium on Computers and Communications, 1998.*

[10] Michael Yang and Ahmed Tantawy, "A design methodology for protocol processors", *Proceedings of the Fifth IEEE Workshop on Distributed Computing Systems, 1995.*