A fast instruction fetch unit for an embedded stack processor

Alongkot Burutarchanai, Vishnu Kotrajaras, Prabhas Chongstitvatana
Department of computer engineering
Chulalongkorn university
Bangkok 10330, Thailand
E-mail : g46abl@cp.eng.chula.ac.th, vishnu@cp.eng.chula.ac.th,prabhas@chula.ac.th

Abstract

The purpose of this work is to improve performance of a 16-bit stack processor. This processor is suitable for embedded applications. A stack processor has an advantage of low complexity but its performance can be improved. Observing the instruction fetch consumes 53% of the execution cycle, focusing on improving instuction fetch is the primary goal of this work. The proposed scheme uses 16-bit fetch with some additional path to reduce the number of control cycle. The work also suggests the use of instruction compression. The result shows the performance improvement of 32% and 37% respectively, achieving the reduction in instruction fetch cycle by 61% and 70%.

Keyword : Instruction fetch, stack processor, embedded system

A fast instruction fetch unit for an embedded stack processor

Alongkot Burutarchanai, Vishnu Kotrajaras, Prabhas Chongstitvatana
Department of computer engineering
Chulalongkorn university
Bangkok 10330, Thailand
E-mail : g46abl@cp.eng.chula.ac.th, vishnu@cp.eng.chula.ac.th,prabhas@chula.ac.th

Abstract

The purpose of this work is to improve performance of a 16-bit stack processor. This processor is suitable for embedded applications. A stack processor has an advantage of low complexity but its performance can be improved. Observing the instruction fetch consumes 53% of the execution cycle, focusing on improving instuction fetch is the primary goal of this work. The proposed scheme uses 16-bit fetch with some additional path to reduce the number of control cycle. The work also suggests the use of instruction compression. The result shows the performance improvement of 32% and 37% respectively, achieving the reduction in instruction fetch cycle by 61% and 70%.

1. Introduction

Embedded systems are emerging as a major driving force for computing devices. The much larger volume of demand makes embedded systems a dominant factor in industries. Hand held devices are becoming prevalent in today society. Mobile phone, PDA and the like are increasingly more powerful, they will play media rich data: songs, movies and will be the center of communication for everyday life. The media rich application has driven the design into a new direction [1].

One example of the emerging trend is the mobile code [2]. In the server-client model, the server ships an executable code to the client to be executed there. The portability is the primary goal, that the mobile code is independent of the client platforms. The compactness of code [3] is the secondary goal, to reduce the time for transporting the mobile code through the network and to reduce the storage requirement on the client devices.

For mobile applications, bytecode becomes a popular choice of mobile code. Bytecode achieves a good code compaction [4]. A program in the form of the intermediate code of a stack-based instruction set is more compact than a program in the form of the machine code of a register-based instruction set. One reason is that the location of an operand is implicit in the stack pointer, on the other hand, the operand of a register machine must be declared explicitly. An example of a popular bytecode is the Class File or Java bytecodes [5] of the Java language.

We have designed a stack-based processor [6] aimed at a low cost device. This processor can execute bytecode directly. The motivation behind this design is to build a processor which consumes very small resource. Stack architecture is suitable for such goal. The processor has 16-bit data path. It contains a few number of dedicate registers. Its instruction is small and tends toward minimalist, hence the low complexity in the design.

The aim of this work is to improve the performance of this processor. However, the original goal of low resource devices must be preserved. So, the challenge is to improve the performance using as little as possible additional resource on the chip. We observe that instruction fetch consumes more than half of execution cycles. Improving instruction fetch will improve performance. This is the area focused by this work. The next section describes the stack processor architecture and our approach to improve the instruction fetch.

2. Stack Processor

It is a 16-bit processor. Its instruction set is stack-based instruction. The instructions are byte-coded to achieve compact executable. The data path is simple. It contains 4 registers: top of stack (TOS), program counter (PC), stack pointer (SP), frame pointer (FP). The top-of-stack register caches the top-most value of the evaluation stack. The stack pointer points to the data on the top of stack. The frame pointer points to the activation record which manages subroutine call.

There are two internal buses: data bus and temp bus. The data bus transfers 16-bit data to and from an external memory. The temp bus is connected to data bus by a buffer (buf). Two internal registers: FF for storing a temporaly value and MAR (memory address register).

MAR drives the address of the external memory. The intruction fetch unit contains three 8-bit registers: instruction register (IR), operand-0 (OPR0), operand-1 (OPR1). The OPR0 and OPR1 can be concatenated to form a 16-bit operand. (See Fig 1)

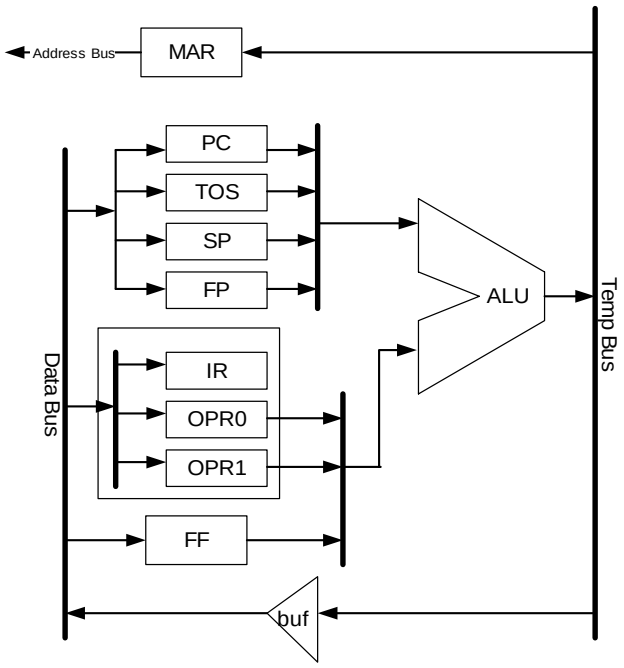

Fig 1 the micro architecture of the stack processor

The instruction format consists of 8-bit op-code and zero or one operand. The operand is either 8-bit or 16-bit. The addressing for data is on word-boundary but for instruction is on byte-boundary as the instruction is byte-addressed. Each instruction fetch is one byte at a time. The data is accessed on word (16-bit) basis.

The current design requires approximately 6000 equivalent gates on a Xilinx FPGA device.

3. Improving the instruction fetch

The control unit fetches byte-coded instruction byte by byte through a 16-bit data bus. This is an obvious goal to pursue more instruction bandwidth through the data bus. Four incremental improvements are done:

1) 16-bit fetch and caching the low-address byte.
2) add increment PC
3) add direct path to MAR
4) instruction compression

In the following section the micro-cycle of the original control unit will be described and the suggested improvement will be explained. The original control scheme is denoted m1. The improved version with 16-bit fetch (1+2+3 above) is denoted m3. The version with instruction compression (4) is denoted m4. The source-> destination notation is used to describe data transfer in the data path.

Each micro-cycle operates on bus->register and register-> bus basis. It can not transfer register to register in one cycle. The instruction fetch works as follows:

```
<m1 baseline>

1       PC->alu->tbus
2       tbus->MAR, Mread
3       dbus->IR
4       PC->alu(+1)->tbus
5       switch(num_of_argument_byte(IR))
6   0:  tbus->PC, dispatch
7   1:  tbus->MAR, tbus->PC, Mread
8       dbus->OPR1, sign-ex->OPR0,
            PC->alu(+1)->tbus
9       tbus->PC, dispatch
10  2:  tbus->MAR, tbus->PC, Mread
11      dbus->OPR0, PC->alu(+1)->tbus
12      tbus->MAR, tbus->PC, Mread
13      dbus->OPR1, PC->alu(+1)->tbus
14      tbus->PC, dispatch
```

For zero argument, it takes 5 cycles. For 8-bit argument, it takes 7 cycles and for 16-bit argument it takes 9 cycles for instruction fetch. Line 5 does not counted as it is internal branching operation of the control unit. Line 3 fetches the op-code byte. For 8-bit argument, line 7-8 fetches the argument to OPR1 and sign extends it to OPR0 to form a 16-bit operand. For 16-bit argument, line 10-11 fetches the first byte to OPR0, line 12-13 fetches the second byte to OPR1.

To reduce the number of memory read, the fetching should be done 16-bit at a time. The gain will come from the instruction that has 8-bit argument as it takes only one memory read when the address is word-aligned. The first byte is op-code. The second byte is 8-bit argument. For 16-bit argument, it always a gain because only two memory reads are required instead of three byte-reads. For unaligned address, the odd-address byte of the previous memory read can be cached and reused. Let denote an even address as high byte and odd address as low byte (big-endian). MLO is an 8 bit register caching the low byte (See Fig. 2)

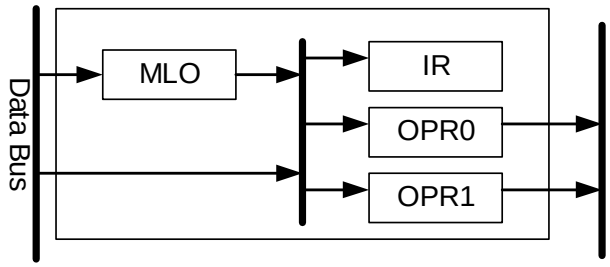

Fig. 2 show MLO register in the instruction fetch unit

The initial state for each instruction fetch is determined by the least significant bit of the current PC. There are 4 distinguished states: fetch even, fetch odd, refetch even, refetch odd. On refetch odd, the old low byte has already been in the cache so no additional memory read is needed. At the end of execution, if no branching is done, "refetch" is performed. If the instruction is control-transfer, cache must be refresh and "fetch" is performed. To reduce the number of micro-cycle, increment PC operation is added to the data path, denoted by PC+1. The micro-cycle for four instruction fetch states are presented separately as follows:

```
fetch even

PC->alu->tbus
tbus->MAR, Mread
low->MLO, high->IR, PC+1

switch num_of_argument_byte(IR)
0: dispatch
1: MLO->OPR1, sign-ex->OPR0, PC+1,
   dispatch
2: MLO->OPR0, PC->alu->tbus
   tbus->MAR, Mread, PC+1
   low->MLO, high->OPR1, PC+1,
   dispatch
```

```
fetch odd

PC->alu->tbus
tbus->MAR, Mread
low->IR, PC+1

switch num_of_argument_byte(IR)
0: dispatch
1: PC->alu->tbus
   tbus->MAR, Mread, PC+1
   low->MLO, high->OPR1, sign-ex->OPR0,
   dispatch
2: PC->alu->tbus
   tbus->MAR, Mread, PC+1
   high->OPR0, low->OPR1, PC+1,
   dispatch
```

refetch even is similar to fetch even.

```
refetch odd

MLO->IR, PC+1
switch num_of_argument_byte(IR)
...
the rest is similar to fetch odd
```

The above four states are combined into one control sequence.The least significant bit (right most) of PC when it is latched into MAR is used. It is stored in a one-bit register, LB. LB is used to branch on even/odd address. Let selectbus(LB) denotes a multiplexor of high/low data bus to IR. The control sequence is as follows:

```
1        PC->alu->tbus
2        tbus->MAR, Mread
3        low->MLO, selectbus(LB)->IR, PC+1
4        switch(num_of_argument_byte(IR))
5    0:  dispatch
6    1:  if even (LB = 0)
7          MLO->OPR1, sign-ex->OPR0, PC+1,
           dispatch
8        else odd (LB = 1)
9          PC->alu->tbus
10         tbus->MAR, Mread, PC+1
11         low->MLO, high->OPR1,
           sign-ex->OPR0, dispatch
12   2:  if even (LB = 0)
13         MLO->OPR0, PC->alu(+1)->tbus
14         tbus->MAR, Mread, PC+1
15         low->MLO, high->OPR1, PC+1,
           dispatch
16       else odd (LB = 1)
17         PC->alu->tbus
18         tbus->MAR, Mread, PC+1
19         high->OPR0, low->OPR1, PC+1,
           dispatch
```

For refetch odd, line 1-3 is changed to

```
1-3 MLO->IR, PC+1
```

Line 4, 6, 8, 12, 16 are internal branching of the control unit. The number of micro-cycle (for instruction started at even/odd address) is reduced to 4/4 cycles for zero argument, 4/6 cycles for 8-bit argument, 6/6 cycles and for 16-bit argument. Refetch odd reduces another 2 cycles.

This control sequence can be improved further. Observing that transfering PC to MAR is 2 cycles. Creating a direct path PC->MAR will shorten this to one cycle. However for fetch even with an argument 2 bytes, line 13-15:

```
13      MLO->OPR0, PC->alu(+1)->tbus
14      tbus->MAR, Mread, PC+1
15      low->MLO, high->OPR1, PC+1, dispatch
```

Having a direct PC->MAR will not reduce the cycle on this sequence because the need to increment PC. So, a functional unit PC+1->MAR is added to the data path as Fig. 3.

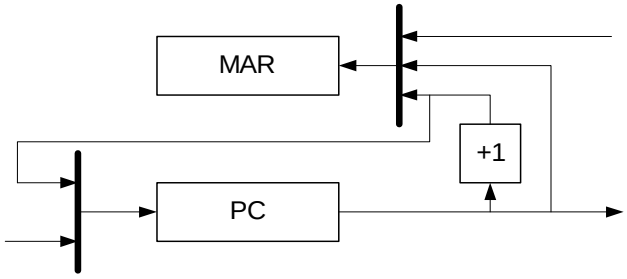

Fig. 3 show circuit changed by add adder at pc

Including these two changes into the control unit, the micro-cycle becomes:

```
<m3 fetch16>

1         PC->MAR, Mread
2         low->MLO, selectbus(LB)->IR, PC+1
3         switch(num_of_argument_byte(IR))
4  0:        dispatch
5  1:        if even (LB = 0)
6              MLO->OPR1, sign-ex->OPR0,
               PC+1, dispatch
7             else odd (LB = 1)
8              PC->MAR, Mread, PC+1
9             low->MLO, high->OPR1,
               sign-ex->OPR0, dispatch
10 2:         if even (LB = 0)
11             MLO->OPR0, PC+1->MAR,
               Mread, PC+1
12             low->MLO, high->OPR1, PC+1,
               dispatch
13            else odd (LB = 1)
14             PC->MAR, Mread, PC+1
15             high->OPR0, low->OPR1, PC+1,
               dispatch
```

The number of micro-cycle (for instruction started at even/odd address) is 3/3 cycles for zero argument, 3/4 cycles for 8-bit argument, 4/4 cycles and for 16-bit argument. Refetch odd reduces another cycle.

Ultimately, to reduce the number of cycle of instruction fetch, the size of instruction must be reduced. One approach for instruction compression [7] [8] [3] [4] is by restricting the range of argument and/or combining a small argument with the instruction into one byte. This requires change in instruction format. For illustrative purpose, we conduct an experiment, choosing to compress two most frequently used instructions, load/store local and two instructions that are easy to modify, jump conditional. Load/store local has 8-bit argument. Jump conditional has 16-bit argument. For load/store, the range of local variable is restricted to 1..16 (4-bit), so the instruction can be combined with its argument into one byte. For jump conditional, the range of offset is restricted to 12-bit, and combine the instruction with 12-bit argument into two bytes. This choice is a small set that can illustrate the effect of code compression on instruction fetch cycles. The set covers both 8-bit argument and 16-bit argument compression. This modification does not change the micro-cycle of the instruction fetch except for different decoding of the instruction. It only affects the amount of memory read for instructions. The micro-cycle sequence for m4 is the same as m3.

4. Experiment

The effect of instruction fetch on the improved control unit is measured and compared with the original control unit. The number of cycles of the processor executing benchmark program are measured under cycle-accurate simulation. The detail of the benchmark programs is shown in Table 1. Table 2 shows the number of cycles in executing benchmark programs. Columns f1, f3, f4 are the number of cycles of instruction fetch of m1 -- the baseline machine with the original control unit, m3 -- with 16-bit fetch and additional direct path, m4 -- with code compression consecutively. Table 3 shows the number of memory read for instruction fetch from code segment of three machines, c1, c3, c4 for m1, m3, m4 consecutively. Table 4 shows the total number of instruction executed of each programs.

Table 1 Stanford benchmark programs

Bubble	Sort 20 numbers by bubble sort algorith
Hanoi	Find a solution to move 6 disks in tower of Hanoi
Matmul	Multiply two 4x4 matrices
Perm	Permute 4 digits of 0, 1, 2, 3
Quick	Sort 20 numbers by quick sort algorithm
Queen	Find all solutions of 8-queen problem
Sieve	Find all prime numbers less than 500

5. Discussion

The column q1 in Table 5 shows the ratio of cycles of instruction fetch over total cycles (q1 = f1/t1) of the baseline control unit. This shows that 53% of cycles are consumed in the instruction fetch. The column q2 shows the percent of reduction of cycles in instruction fetch of m3 (fetch 16-bit) over m1 (1-f3/f1). The proposed control unit reduces the number of cycle in instruction fetch by 61%. The column q3 shows the similar figure for m4 (with code compression) over m1 (1-f4/f1). With code compression the number of cycle in instruction fetch is reduced 70%. Fig. 3 shows the comparison of the number of cycle in instruction fetch of three machines (%).

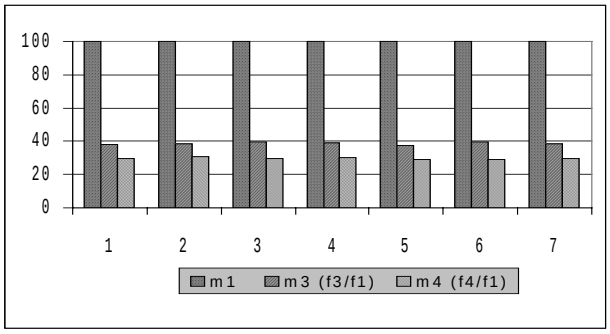

Fig 3 the number of cycle in instruction fetch of m1, m3, m4. for each program

With 16-bit fetching the number of memory read in code segment is also reduced. The column q4 (1-c3/c1) in Table 5 shows the percent reduction in the number of CS read of m3 (fetch 16-bit) over m1 is 46%. The column q5 (1-c4/c1) in Table 5 shows the similar figure for m3 (fetch 16-bit) over m1, 59%. Fig. 4 shows the comparison of the number of memory read for instruction fetch of three machines.

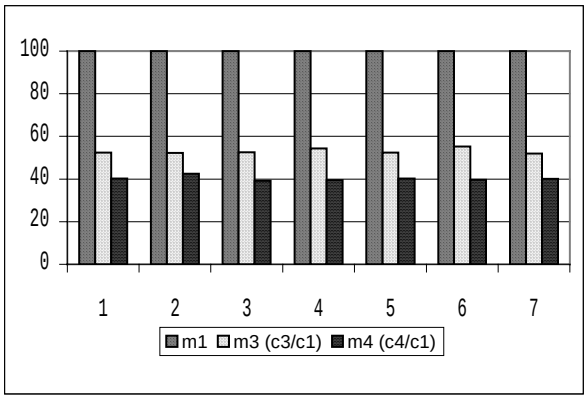

Fig 4 the number of memory read in instruction fetch of m1, m3, m4

All these numbers translate to performance improvement. The column q6 indicates
the speedup of m3 over m1 (1-t3/t1), 32%. The column q7 indicates the speedup of m4 over m1 (1-t4/t1), 37%. The performance can be shown in terms of the number of cycle per instruction (CPI). m1's CPI is (t1/I1) 12.58. m3's CPI is (t3/I1), 8.44. For m4, its CPI is (t4/I1), 7.83. In addition, the instruction compression also reduces the static code size by 16% (data is not shown).

In summary, the proposed control sequence for instruction fetch, m3 which fetch 16-bit, reduces the number of cycles of instruction fetch more than half over the original scheme. It also reduces the number of memory read for instruction fetch by almost half. Using it with instruction compression, m4, reduces the number of cycle of instruction fetch further to 61%. In terms of performance, the 16-bit fetch scheme shows 32% speedup over the original. This is quite a remarkable result considering how little additional hardware the proposed scheme required.

6. Conclusion

This work presents an improvement of instruction fetch control sequence of a stack-based embedded processor. Instead of fetching byte-coded instructions byte by byte, the proposed scheme fetches 16-bit. The result is an improvement in performance stemmed from reduction of cycles in instruction fetch. The experiment is carried out to measure the effect of the proposed scheme via cycle-accurate simulation of executing benchmark programs. The design presented here achieves 32% speedup over the original design. Moreover, using with instruction compression, it can achieve 37% speedup. The modification of data path is kept to a minimum. This indicates a good return on investment and is applicable to a small embedded processor design.

7. References

[1] C. Kozyrakis and D. Patterson, "A new direction for computer architecture research", IEEE computer, Nov. 1998, pp.24-32.

[2] M. Franz, "Code-Generation On-the-Fly: A Key to Portable Software", Doctoral Dissertation No. 10497, ETH Zurich; published by Verlag der Fachvereine, Zurich, ISBN 3-7281-2115-0; March 1994.

[3] V. Kotrajaras, P. Chongstitvatana, "Nibbling Java byte code for resource-critical devices", National Conf. of Computer Science and Engineering, Thailand, 2003.

[4] P. Nanthanavoot, P. Chongstitvatana, "Code-Size Reduction for Embedded Systems using Bytecode Translation Unit", Conf. of Electrical/Electronics, Computer, Telecommunications, and Information Technology (ECTI), Thailand, 13-14 May 2004.

[5] B. Joy, G. Staddle, J. Gosling and G. Bracha, JAVA Language Specification, 2nd ed, Addison Wesley Pub, 2000.

[6] A. Burutarchanai, P. Nanthanavoot, C. Aporntewan, P. Chongstitvatana, A stack-based processor for resource efficient embedded system, TENCON 2004.

[7] I. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. "Code compression," Proc of the ACM SIGPLAN'97 Conf on Programming Languages Design and Implementation 32 (15–18 June 1997): 358-365.

[8] H. Lekatsas, J. Henkel, W. Wolf, "Code Compression for Low Power Embedded System Design", Proc. of the 37th Conf. on Design automation, 2000.

Table 2 the number of cycles

Program	f1*	f3**	f4***	t1****	t3****	t4****
bubble	78847	30062	23320	147731	98946	92204
hanoi	15729	60535	4817	29696	20002	18784
Matmul	98054	39079	28871	182235	123260	113052
perm	37635	14699	11347	69301	46365	43013
queen	4826363	1794447	1408594	8909675	5877759	5491906
quick	26990	10762	7899	49687	33459	30596
sieve	117677	45640	35112	221670	149633	139105

*f1 no of cycle of instruction fetch of m1 baseline
**f3 no of cycle of instruction fetch of m3
***f4 no of cycle of instruction fetch of m4 (with code compression)
****t1, t3, t4 total no of cycle for m1, m3, m4

Table 3 the number of CS read (byte)

program	C1*	C3**	C4***
Bubble	21729	11384	8730
Hanoi	4454	2323	1888
Matmul	28410	14937	11141
Perm	10681	5802	4228
queen	1267531	664375	510114
quick	7588	4192	3011
sieve	33124	17225	13276

*c1 no of CS read of m1 baseline
**c3 no of CS read of m3
***c4 no of CS read of m4 (with code compression

Table 4 total number of instruction executed (instruction)

program	I1
bubble	11925
hanoi	2317
matmul	13886
perm	5469
queen	765141
quick	3972
sieve	17151

Table 5 computed results from Table 2 and Table 3

program	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Buble	53.37	61.87	70.42	47.61	59.82	33.02	37.59
Hanoi	52.97	61.63	69.38	47.84	57.61	32.64	36.75
Matmul	53.81	60.15	70.56	47.42	60.78	32.36	37.96
Perm	54.31	60.94	69.85	45.68	60.42	33.10	37.93
Queen	54.17	62.82	70.81	47.59	59.76	34.03	38.36
Quick	54.32	60.13	70.73	44.75	60.32	32.66	38.42
Sieve	53.09	61.22	70.16	48.00	59.92	32.50	37.25
average	53.72	61.25	70.27	46.98	59.80	32.90	37.75

q1 % num. of cycles in instruction fetch
q2 % reduction of cycles in instruction fetch of m3 over m1
q3 % reduction of cycles in instruction fetch of m4 over m1
q4 % reduction of cycles in num. of CS read of m3 over m1
q5 % reduction of cycles in num. of CS read of m3 over m1