Stack Frame Caching

Prabhas Chongstitvatana
Department of Computer Engineering, Chulalongkorn University
Bangkok 10330, Thailand
Email: prabhas@chula.ac.th

Abstract

A stack-based processor where the instruction set acts on data resided on an evaluation stack, has its performance bottleneck due to the limit of access to the stack. Although the stack-based processor has low performance, it is very low cost to implement. This paper proposes a method to improve the performance of a stack-based processor. A number of fast registers are used to "cache" part of the evaluation stack. This can be achieved without affecting the instruction set and most of the data path. A detailed analysis of the control steps is given. The performance gain is around 30% of the original processor.

Key Words: stack-based processor, micro-architecture, stack frame caching

1. Introduction

A stack-based processor has the advantage of simplicity. The stack-based instruction set is also compact. This type of architecture is suitable for a low cost embedded system. However, it has one shortcoming, performance. Due to the nature of stack access, only one item can be accessed at a time. This paper proposes a simple scheme to improve the performance of stack-based processors, called stack frame caching. A number of fast registers are used to cache part of stack frame. This allows concurrent access of two items from the stack. Stack frame caching does not require any change in the instruction set; hence, it is applicable to wide range of stack-based processors. The experiment shows that the performance improvement is around 30%.

2. Reference processor

To illustrate our proposed method, a stack-based processor will be used as a reference processor which will be augmented with the stack frame caching. The reference processor is due to [1]. It is a simple stack-based processor aimed for embedded applications. The entire stack is stored in the memory.

There are seven special purpose registers (no visible user registers): `TS`, `FP`, `SP`, `NX`, `FF`, `IR` and `PC`. `TS` caches the top of stack value (Fig. 2).

`TS`	top of stack
`FP`	frame pointer
`SP`	stack pointer
`NX`	temp register
`FF`	temp register
`IR`	instruction register
`PC`	program counter

The program counter, `PC`, can be updated independent of other registers. This allows fetching an instruction in one cycle. The data path consists of one ALU connected to the register bank. The output of ALU, `tbus`, goes back to the register bank. The memory is interfaced to the processor through the bus interface unit (BIU). The BIU interfaces the data input, `din`, and the data output, `dout`, to the memory data bus. `din` is selected from `TS` or `FP`. The input of the register bank, `bus`, is multiplexed from `tbus`, `dbus` and `PC`. The address bus, `abus`, is multiplexed from `PC` and `tbus`. The `PC` can be updated with `PC`+1 or `PC`+`arg` or `tbus`. The ALU has two ports: `p1`, `p2` and can perform usual arithmetic and logic functions. There are two flags: Zero, and Sign.

Using 2-phase clock enables read-modify-write of registers in one cycle. Reading from registers and memory will be on the positive edge and writing to registers will be on the negative edge. The basic cycles are:

- read-modify-write registers
- register transfer
- memory read
- memory write

Register access

The basic read-modify-write starts at the positive edge of the clock. The data are read from the registers into the ALU ports through the multiplexor *x* and *y*. The ALU outputs the result to *tbus*. At the negative edge, *tbus* is fed back to the input of registers, *bus*, through the multiplexor *b* and is latched into the designated register.

Read-modify-write a register

```
pos edge: R -> alu -> tbus
neg edge: tbus -> R
```

Register transfer

```
pos edge: R1 -> tbus
neg edge: tbus -> R2
```

A number of changes have been made to improve the performance of the reference processor.

1) The instruction format is a fixed length 32-bit with 8-bit opcode and 24-bit argument. This allows fast instruction fetch.
2) The data width is increased to 32 bits to match with the instruction width.

Control steps notation

The notation used in describing an execution step is as follows.

```
src->dest
```

denotes the event that transfer data from a source to a destination where source and destination can be a wire or a register. A wire represents a connection or the input/output of a component.

```
alu(p1 op p2)->dest
```

denotes the ALU performing the "*op*" on its two input ports, *p1* and *p2*, and its output is connected to *dest*, where *dest* can be a wire or a register.

```
mR(ads)->dest
src->mW(ads)
```

mR denotes memory read with the address from the source *ads*, the data is transferred to *dest*. *mW* denotes memory write with the data sets to the source, *src*, and the address is *ads*. *src* and *dest* can be a wire or a register. The concurrent events are specified by writing them on the same line. Each event is separated from other event by ",". The order of events in the same line is unimportant because they occur in the same clock cycle. However, some event occurs on the positive edge of the clock, some event occurs on the negative edge of the clock. Reading from registers and memory will be on positive edge and writing to registers will be on negative edge.

```
src->dest, mR(ads)->dest, ...
```

We have a shorthand notation for *SP*.

```
sp-1   is alu(sp-1)->sp
sp+1   is alu(sp+1)->sp
```

Let the shorthand notation of *push/pop* be

```
push x is
  sp+1->sp
  x->mW(sp)

pop x is
  mR(sp)->x
  sp-1->sp
```

3. Stack frame caching

Almost all instructions of stack-based processors perform *push* and *pop*. This is because two reasons. The first reason is that it is the nature of the stack-based instruction set to access data from the evaluation stack. The second reason is that the top of stack is cached in *TS*, therefore there is a lot of traffic between *TS* and the stack segment. In the reference processor, *push* and *pop* do one memory access and use ALU to do increment/decrement *SP*. The most frequently used instruction is "*get*" (loading a local variable from the stack frame to top of stack register). ALU is used to calculate the address of variable to be loaded from the stack frame which is stored in the memory, *address = FP-arg*, where *arg* is the reference of the local variable. It has the following control steps.

```
<get>
sp+1
ts->mW(sp)
alu(fp-arg)->tbus, mR(tbus)->ts
```

There are two key ideas to improve the performance.

1) The operations *push/pop* can be done in one cycle if *SP* can be incremented/decremented independent of ALU and they can achieve pre-increment and post-decrement at the proper negative-edge of the clock.

2) To improve "*get*", the local variable must be stored in a register instead of memory as *push/pop* also access memory. If it is done properly "*get*" will take just one cycle.

Let *v[.]* denotes the *caching* register bank. It is connected to *TS* in the data path (see Fig. 2). Using the *caching* register bank will allow accessing a local variable and *TS* at the same time. The "*get*" can be done in one cycle.

```
<get>
$1 push ts, $2 v[arg]->ts
```

Where *$1* denotes positive-edge and *$2* denotes negative-edge, *v[.]* is the cache register. The old value *TS* is pushed into memory at *$1*, before the new value from *v[arg]* is written to *TS* at *$2*.

Push/pop

To push a register to memory in one cycle, the "*sp*+1" must appear at the address bus from the beginning of *$1*, *TS* is presented to data bus at the same time, at the beginning of *$2* memory write signal is ended (it is assumed that the value is written into memory here), the value of "*sp*+1" is also written to *SP* at this time. With the new scheme, *push* becomes

```
$1 sp+1->abus, ts->dbus, $2 mW(abus),
sp+1->sp
```

Popping a register can be done in one cycle. The value "*sp*" is presented to the address bus at *$1*. The memory is read. At *$2*, the data is latched to a register, at the same time, "*sp*-1" is written to *SP* (post-decrement). *pop* becomes

```
$1 sp->abus, mR(abus)->dbus,$2 dbus->
x, sp-1->sp
```

With this new *push/pop*, most instructions will be faster. For example, "*push a literal*" takes only one cycle for execution.

```
<lit>
$1 push ts, $2 arg->ts
```

"*load*" cannot be done in one cycle as it reads the memory twice, the first one to push *TS*, the second one for getting the value. Therefore "*load*" takes 2 cycles.

```
<load>
push ts
mR(arg)->ts
```

All the binary operations now take 2 cycles.

```
<bop>
pop ff
alu(ts op ff)->ts
```

Implementing the SP unit

To perform increment/decrement on *SP* in concurrent with other ALU operations, *SP* must be a separate unit. The SP unit performs pre-increment at *$1*, post-decrement at *$2*, and loads a value from bus at *$2*. There is a feed forward path from the adder "*sp*+1" to achieve the pre-increment. All multiplexors are asserted at *$1*, latching the register *SP* is at *$2* (Fig.1).

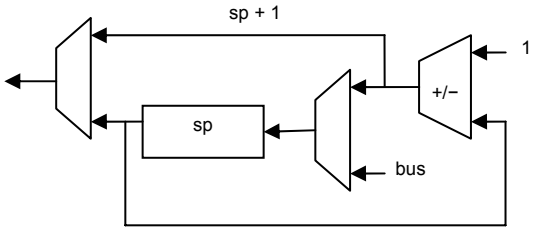

Figure 1 The SP unit

Stack frame

A number of registers are used to cache a part of stack frame. The stack frame remains unchanged from the original design. The local variables, $lv_1..lv_n$, are cached into *v[1]..v[n]* the cache registers. When the context is changed by *call/ret*, these registers are affected. Before a new activation record is created the *old* cached registers must be written back to the current activation record. And vice versa, upon returned from a call, after the activation record is deleted and the old one restored, the cache registers must be refreshed (re-cached) from the activation record. The pseudo code *call/ret* are as follows.

```
<call>
* save v to the current stack frame
  push ts (flush stack)
  create a new frame
  save fp' and return address
* cache v from the new frame
  update sp

<ret>
  restore return address, sp
  restore the old frame
* cache v of this current frame (restore old v)
  if it is "ret" pop ts
```

The lines with * are the additional work that must be done to do stack frame caching. The control steps for `call/ret` for saving/caching `v[.]` are as follows.

```
<save v>
alu(fp-n)->fp
vn->mW(fp), alu(fp+1)->fp
...
v1->mW(fp), alu(fp+1)->fp

<cache v>
alu(fp-n)->fp
mR(fp)->vn, alu(fp+1)->fp
...
mR(fp)->v1, alu(fp+1)->fp
```

If the size of caching register is *n* then the extra cycle in `call/ret` instruction is O(3(*n*+1)).

The simple analysis of the previous section has the worst case additional running time for using stack frame caching in O(3(*n*+1)) cycles. However, it is not the case that a function call will use all `v` registers. Let *maxv* be the number of `v` registers, *fs* be the size of activation record. If the size of activation record is less than *maxv* then only `v[1]..v[fs]` must be saved/cached. Let `u` be *max(fs, maxv)*; it is stored in the register `U`. The `U` register is used to skip a number of control steps to achieve this effect. The control signal "`skipu`" sets the next control step to *mpc+(maxv-u)*, where *mpc* is the current control step. Assume the size of caching register is 4 (*maxv* = 4). The control steps below show the part to save `v` registers at the function call.

```
<save v>
alu(fp-u)->fp, skipu
v[4]->mW(fp), fp+1->fp
v[3]->mW(fp), fp+1->fp
v[2]->mW(fp), fp+1->fp
v[1]->mW(fp), fp+1->fp
```

Caching `v` registers can be achieved similarly. In fact, when calling a function, not even `u` registers need to be cached, only the passing parameters (*p*) need to be cached from the evaluation stack (it is a save when $p < u$). However, it becomes too complex to do in a simple control unit such as this due to the ordering the variables. Therefore, a tradeoff has been made not to exploit this fact.

The "`call`" instruction saves the return address to `TS` and saves `v` registers. The "`fun`" creates the new activation record and caches the passing parameters from the evaluation stack to `v` registers.

```
<call>
; store  ret ads on ts
ts->mW(sp+1), sp+1->sp, pc+1 ; flush ts
pc->ts, arg->pc, if u=0 <fetch>
<save v>
alu(fp-u)->fp, skipu
v[4]->mW(fp), fp+1->fp
v[3]->mW(fp), fp+1->fp
v[2]->mW(fp), fp+1->fp
v[1]->mW(fp), fp+1->fp
<fun>
fp->mW(sp+k), sp+k->sp ; save fp, new sp
sp->fp                 ; new fp
u->mW(sp+1), iru->u, sp+1->sp   ; push u
pc+1
<cache v>
alu(fp-u)->fp, skipu
mR(fp)->v[4], fp+1->fp
mR(fp)->v[3], fp+1->fp
mR(fp)->v[2], fp+1->fp
mR(fp)->v[1], fp+1->fp

<ret>
sp-1->ff
alu(fp=ff), ifF <r2>   ; test for retv
ts->pc                 ;  ret ads on TS
mR(sp)->u              ; pop u
alu(fp-arg)->sp
mR(sp)->ts, sp-1->sp, if u=0 <r3>
; if u=0 skip cachev
mR(fp)->fp, <cachev>
<r2>
alu(fp+2)->tbus, mR(tbus)->ff
; ret ads on frame
ff->pc
alu(fp+1)->tbus, mR(tbus)->u   ; pop u
alu(fp-arg)->sp, if u=0 <r3>
; skip cachev
mR(fp)->fp, <cachev>
<r3>
mR(fp)->fp, <fetch>      ; restore fp
```

When *arg* > *maxv*, the "`get`" accesses normal memory. Even in this case the step of execution is faster due to the SP unit. When *arg* <= *maxv*, the access in on `v` registers and the execution takes only one cycle. The instruction decoder performs a check on the argument of "`get`" and branches to the proper "`get x`" where `x` is *1..maxv*. The pre-increment using "`sp`+1" feed-forward path can be seen.

```
<get>
ts->mW(sp+1), sp+1->sp      ; push ts
alu(fp-arg)->tbus, mR(tbus)->ts, pc+1

<get1>
ts->mW(sp+1),v[1]->ts,sp+1->sp, pc+1
```

```
<get2>
ts->mW(sp+1),v[2]->ts,sp+1->sp, pc+1

<get3>
ts->mW(sp+1),v[3]->ts,sp+1->sp, pc+1

<get4>
ts->mW(sp+1),v[4]->ts,sp+1->sp, pc+1
```

"*put*" is similarly decoded. The post-decrement of *SP* unit allows the instruction to be executed in one cycle.

4. Performance

A number of benchmark programs are compiled and then run on the augmented processor. Table 1 below reports the number of instruction (noi) and the number of cycle (cycle) for each program.

"bubble" is a bubble sort program sorting an array of 20 integers, initially the value in the array is in descending order and sort to ascending order. "hanoi" is a program to solve Hanoi problem with 6 disks. "matmul" is a matrix multiplication program; the input is two matrices of the size 4 × 4. "perm" is a program to do all permutation of {0,1,2,3}. "queen" is a program to find all configurations of 8-queen problem. "quick" is a quicksort program with a similar input to "bubble". "sieve" is a program to find prime numbers less than 1000 using "Sieve of Eratosthenes" algorithm. "aes" is a program to do AES (Advanced Encryption Standard) block cipher (128, 128) bit key. The average cycle-per-instruction number of the reference processor is 4.3.

The average CPI of the augmented processor is 2.9. From the table, comparing the number of clock between the reference processor and the processor with stack frame caching, the average ratio is 0.70. That is, the augmented processor is 30% faster than the reference processor.

Table 1 The performance comparison

	Ref		Improved	
program	noi	cycle	noi	cycle
bubble	10068	44214	10262	32090
hanoi	2312	10092	2377	7544
matmul	3043	12880	3097	9348
perm	4868	20932	4935	14663
queen	618665	2576210	620724	1717782
quick	3172	13539	3224	9551
sieve	28026	124338	28029	75204
aes	30579	131560	30724	90498

5. Related work

The most well-known stack-based instruction set is JVM, the Java virtual machine [2]. It has many implementations, a commercial one is from SUN, PicoJava [3, 4, 5]. The other one from research community is JOP [6]. PicoJava uses a form of "register window" to cache stack frame. It also employs "instruction folding" to merger two instructions into one special instruction for a faster operation. JOP uses special microcode to accelerate the operation, it also employs pipeline. A low cost commercial stack-based processor is also available [7]. It aims for embedded applications. The proposed method is quite different from these works. Stack frame caching uses fast registers in the processor to cache the stack values.

6. Conclusion

To improve the performance of stack-based processors, we employ the technique of stack frame caching. The stack frame caching relies on fast registers to cache a part of the stack frame so that the access to these variables takes only one cycle. The separation of *SP* from the ALU path to have its own increment/decrement, the SP unit, helps to shorten the cycle of the push/pop values from the evaluation stack. There are many approaches to enhance the performance of a processor. In general, the memory sub-system has the major impact on performance. However, in our presentation, the speed of memory, its access time, is assumed to be one cycle, therefore it does not affect our design. This is not a realistic assumption for a general purpose processor but in the context of implementing the design on FPGA with its internal memory block, this is correct.

7. References

[1] Burutarchanai, A., Nanthanavoot, P., Aporntewan, C., and Chongstitvatana, P., "A stack-based processor for resource efficient embedded systems", Proc. of IEEE TENCON 2004, 21-24 November 2004, Thailand.

[2] Lindholm, T. and Yellin, F. The Java™ Virtual Machine Specification, Addison Wesley, 1997.

[3] McGhan, H., O'Connor, M., "PicoJava: a direct execution engine for Java bytecode", Computer, Vol.31, No.10, Oct. 1998, pp. 22-30.

[4] Hangal, S., O'Connor, M.,"Performance analysis and validation of the picoJava processor", IEEE Micro, Volume 19, Issue 3, May-June 1999, pp. 66-72 .

[5] Jianjie, Z., Feihui, L., Yuanqing, G., Zhenwu, Y., Zhilian, Y., "A Java processor suitable for applications of smart card", Int. Conf. on ASIC, 23-25 Oct. 2001, pp.736-739.

[6] Schoeberl, M., “JOP: A Java Optimized Processor for Embedded Real-Time Systems”, Thesis, Vienna University of Technology, 2005.
[7] MARC4 Atmel 4-bit single-chip microcontroller, ATR080, document rev. 4680C-4BMCU-01/05. www.atmel.com

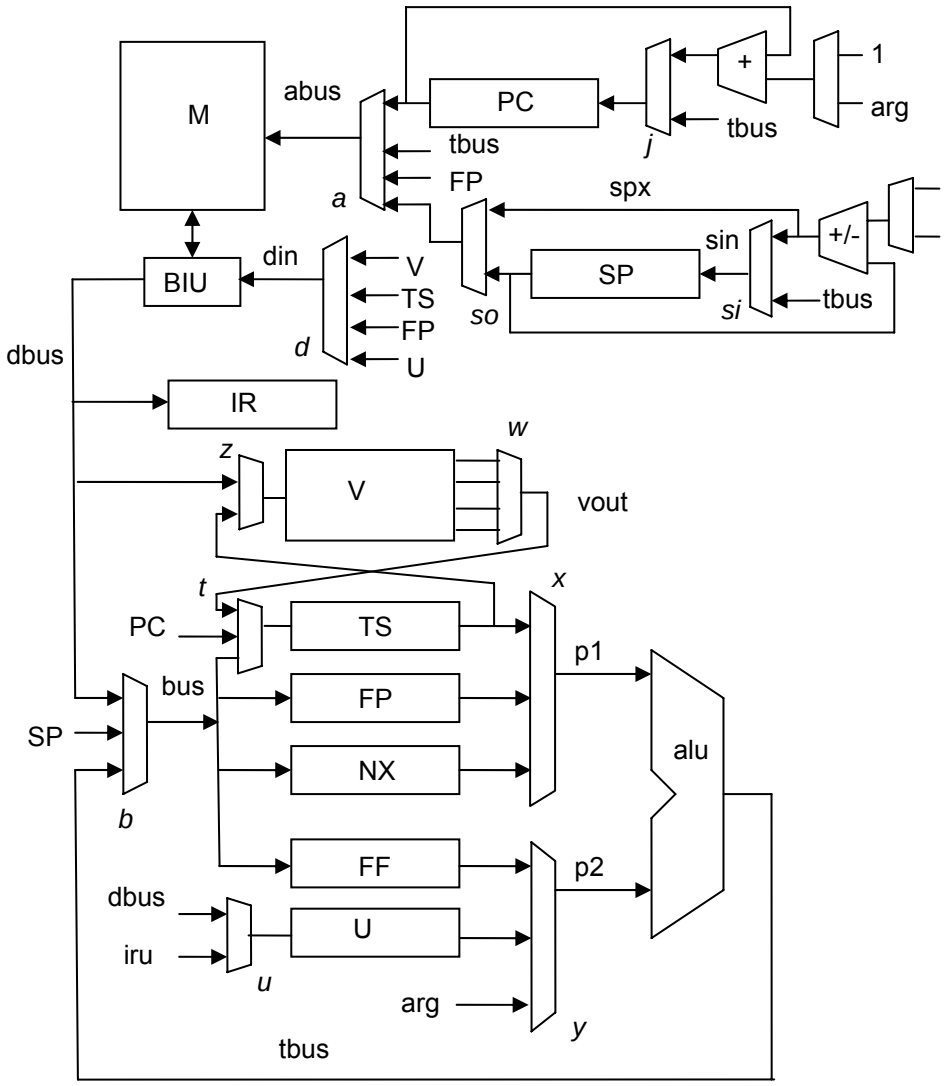

Figure 2 The augmented data path