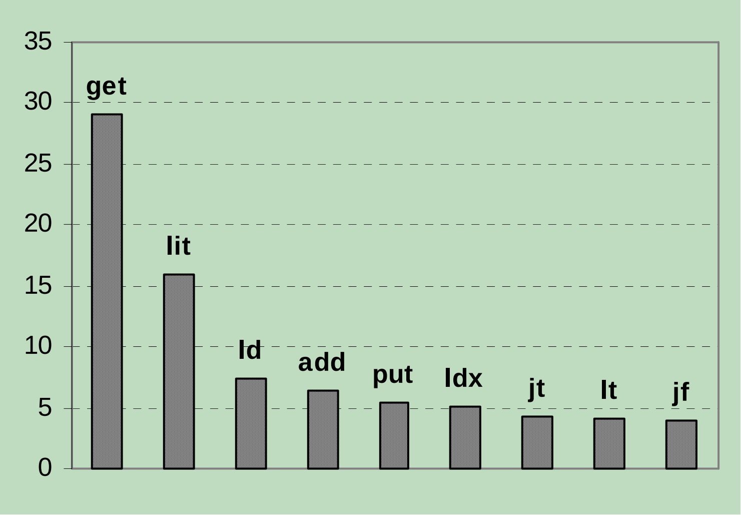
Chapter 7

Performance Enhancement

In this chapter, we will concentrate on performance. Performance improvement starts with an analysis of execution profile to understand where the data path spends most of the time. The effort is then directed to redesign the data path, usually by increasing the concurrent operations in the data path. There are interactions amongst choice of design. Choosing one will affect another. The gain in terms of performance must be weighted against the increased complexity in terms of circuit size or the cost. For the purpose of our study we will not change the instruction set. The performance improvement will come from the change of "micro-architecture" only.



<fig the most frequent used instructions>

In an analysis of the profile of execution of instructions at the level of clock cycle, most frequently used instructions are:

```
      get, lit, ld, add, put, ldx, jt, lt, jf
```

??? show the profile of execution the benchmark, frequency of each instruction and clocks

These instructions altogether are more than 80% of all instructions executed in the suite of benchmark programs (see the appendix for full details of the benchmark programs and the frequencies of all instructions)[1]

To improve the performance, the effort should be spent on improving these instructions.

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
```

The microprograms for the instructions: get, lit, ld, add, put, ldx, jt, lt, jf are:

```
      <get>
        push ts
        alu(fp-arg)->tbus, mR(tbus)->ts

      <lit>
        push ts
        arg->ts

      <ld>
        push ts
        mR(arg)->ts

      <bop>
```

[1] Notable is the “inc v” instruction which is not generated from this compiler (gen2.txt). It is used often but will not be included in this experiment.

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

<put>
  alu(fp-arg)->tbus, ts->mW(tbus)
  pop ts

<ldx>                          ; {ads idx}
  pop ff
  alu(ts+ff)->tbus, mR(tbus)->ts

<jt>
  alu(ts=0), ifT <j3>     ; if true, don't jump
<j2>
  pc+arg
  pop ts

<jf>
  alu(ts=0), ifT <j2>     ; if true, jump
<j3>
  pc+1
  pop ts
```

We can observe that all instructions perform push and pop. This is because two reasons. The first reason is that it is the nature of the stack-based instruction set and 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 Sx processor, push and pop do one memory access and use ALU to do increment/decrement SP.

Key ideas

There are two key ideas:

1. The operations push/pop can be done in one clock 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 most frequently used instruction, the local variable must be stored in a register instead of memory as push/pop also accesses memory. If it is done properly "get" will take just one clock. Let v[.] denotes the "caching" register bank. It is connected to TS in the data path (see Fig sx2-diagram).

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

Where $1 is positive-edge and $2 is 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 clock, 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 begining 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.

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

$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 x is
  mR(sp)->x
  sp-1->sp

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

With this new push/pop, other instructions will also improve. “lit” takes only one cycle for execution.

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

“ld” cannot be done in one cycle as it reads the memory twice, the first one for push ts, the second one for the value. Therefore “ld” takes 2 cycles for execution.

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

All the binary operations now take 2 cycles.

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

“put” can be done in one cycle. ts is read at $1 and transfer to v[arg]. A value in the evaluation stack is popped into ts at $2.

```
<put>
  ts->v[arg], pop ts
```

Similarly to bop, “ldx” takes 2 cycles. “jt” and “jf” take 3 cycles.

Implementing the SP unit

The sp unit performs pre-increment at $1, post-decrement at $2, and load value from bus at $2. There is a feedforward path from the adder “sp+1” to achieve the pre-increment. All multiplexors are asserted at $1, latching the register sp is at $2.

```
           sp+1
     ||<--------------
     ||               |
 <-- ||<---- sp --||---- +/- 1 <--
          |       ||<-bus         |
          |                       |
          -------------------------
```

<fig the sp unit>

Stack frame

A number of registers are used to cache part of stack frame. The stack frame remains unchange 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. This behaviour is the same as saving/restoring registers upon call/ret on a register-based processor. However, in Sx, these saving/restoring are performed at the microprogram level instead of at the instruction level.

```
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 microprograms 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 big-O(3(n+1)).

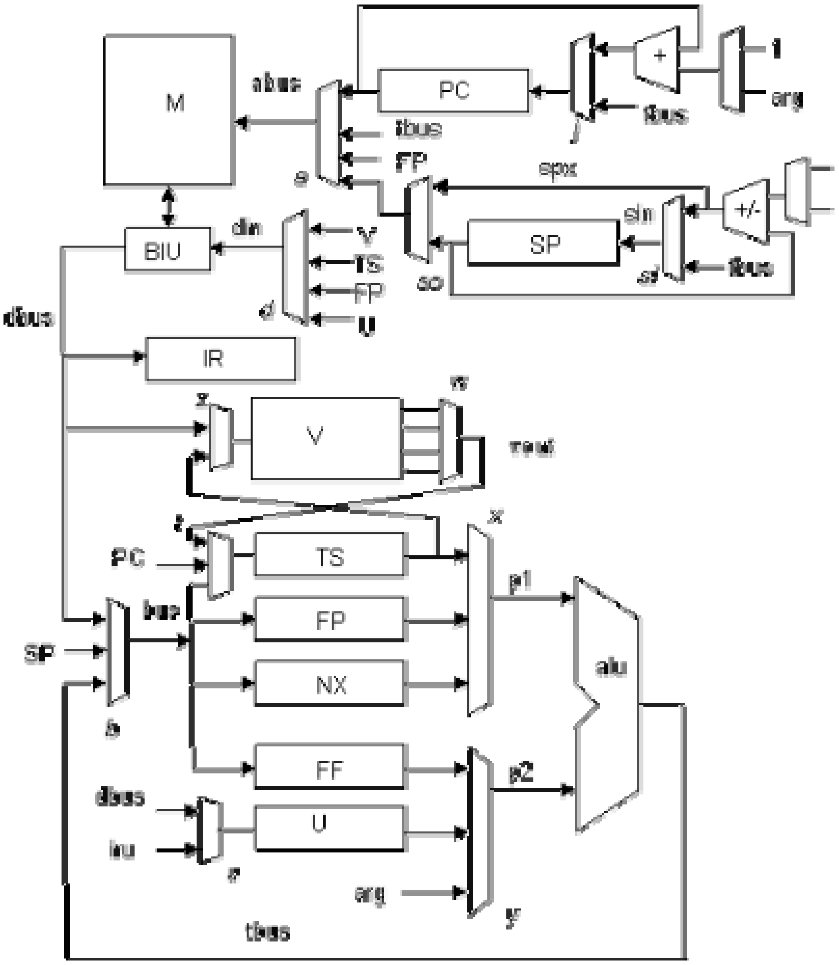
New data path

The enhanced Sx, or Sx2, has many additional functional units, notably the sp unit and the v[.], cache registers. The number of v[.] is 4. However, the major change is in the control unit. There are many more control signals to control the additional functional units and there are more steps of control.

The events are defined as follows.

multiplexor x selects {*ts, fs, nx*}
multiplexor y selects {*ff, u, arg*}
multiplexor b selects {*tbus, dbus, sp*}
multiplexor d selects {*fp, ts, v, u*}
multiplexor a selects {*pc, tbus, fp, spu*}
multiplexor j selects {*pc+1, pc+arg, tbus*}
multiplexor si selects {*sp+1, sp-1, sp+arg, tbus*}
multiplexor so selects {*spx, sp*}
multiplexor z selects {*dbus, ts*}
multiplexor w selects {*v1, v2, v3, v4, varg*}
multiplexor t selects {*vout, pc, bus*}
multiplexor u selects {*dbus, iru*}

alu events are {*add, sub, inc, dec, z, eq, op, p1, p2, add2*}
load registers events are {*ir, ts, fp, sp, nx, ff, pc, v1, v2, v3, v4, varg, u*}
memory events are {*mR, mW*}
next micro-address events are {*ifT, ifF, decode, ifu0, ifp0, ifargm, skipu, trap*}



<fig Sx2 data path>

The new events on the next micro-address {`ifu0, ifp0, ifargm, skipu`} and the register U require further explanation. They are necessary for the control of saving/caching the stack frame. The simple analysis of the previous section has the worst case additional running time for using stack frame caching is big-O(3(n+1)) cycles. However, it is not the case that a function 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 microprogram words to achieve this effect. The control signal is “skipu”. “skipu” sets the next microprogram address to *mpc+(maxv-u)*. This offset is already stored in the next microprogram address field. The microprogram below shows 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, <fetch>
```

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 microprogram such as this due to the ordering the variables. Therefore, a tradeoff has been made not to exploit this fact. One special case has been implemented, when *p = 0* to bypass the passing parameter caching (using the event “ifp0”). These two parameters, *p* and *u*, are encoded in the argument of “fun” instruction with the following format.

```
fun.p.u.k     p:8 u:8 k:8 op:8
```

Where *k* is the frame size, *p* is the arity, *u* is *max(fs,maxv)*. This is done by the code generator or at the loader of the processor simulator. The U register is valid throughout the current context; it is used when “call” and “ret”.

Microprogram of Sx2

Here is the microprogram of the Sx2 processor in whole with the explanation.

```
<fetch>
  mR(pc)->ir, decode
```

The effect of concurrency of sp unit with other operations can be observed in almost every instruction.

```
<bop>
  mR(sp)->ff, sp+1->sp
  alu(ts op ff)->ts, pc+1, <fetch>

<uop>
  alu(ts op ?)->ts, pc+1, <fetch>
```

When *arg* > *maxv*, the "get" accesses normal memory. Even in this case the step of execution is shortening due to the sp unit. When *arg* <= *maxv*, the access in on v registers and the execution takes only one cycle. The "decode" event performs a check on the argument of "get" and branches to the proper "get x" microprogram address 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, <fetch>

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

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

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

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

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

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

```
<put1>
  ts->v[1], mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<put2>
  ts->v[2], mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<put3>
  ts->v[3], mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<put4>
  ts->v[4], mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<ld>
  ts->mW(sp+1), sp+1->sp
  mR(arg)->ts, pc+1, <fetch>

<st>
  ts->mW(arg)
  mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<ldx>                               ;  {ads idx}
  mR(sp)->ff, sp-1->sp              ; pop ads
  alu(ff+ts)->tbus, mR(tbus)->ts, pc+1, <fetch>
```

“stx” benefits from the sp unit the most as it pops the stack many times. In the original Sx, “stx” takes 7 cycles, now it takes 4 cycles.

```
<stx>                               ;  {ads idx val}
  mR(sp)->nx, sp-1->sp              ; pop idx
  mR(sp)->ff, sp-1->sp              ; pop ads
  alu(nx+ff)->tbus, ts->mW(tbus)
  mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<lit>
  ts->mW(sp+1), sp+1->sp, arg->ts, pc+1, <fetch>

<jmp>
  pc+arg, <fetch>

<jt>
  alu(ts=0), ifT j3                 ; if true, don't jump
```

```
<j2>
  pc+arg, mR(sp)->ts, sp-1->sp, <fetch>

<jf>
  alu(ts=0), ifT j2                    ; if true, jump
<j3>
  pc+1, mR(sp)->ts, sp-1->sp, <fetch>
```

Sx2 breaks call/fun into two instructions to reduce the maximum length of any single instruction. 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 the return address is on ts
  ts->mW(sp+1), sp+1->sp, pc+1        ; flush ts
  pc->ts, arg->pc, if u=0 <fetch>     ; save ret ads
<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, <fetch>

<fun>                                ; fun.p.u.k
  fp->mW(sp+k), sp+k->sp             ; save old fp, new sp
  sp->fp                             ; new fp
  u->mW(sp+1), iru->u, sp+1->sp      ; push u
  pc+1, if p=0 <fetch>
<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, <fetch>

<ret>
  sp-1->ff
  alu(fp=ff), ifF <r2>               ; test for retv
  ts->pc                             ; <do ret> retads 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
```

In writing the microprogram for the instructions “inc” and “dec”, a different style is used. Instead of decoding to “inc1”...”inc4”, a test is made to check the range of the argument. If *arg* > *maxv* then it is a normal operation, else the access is on v registers. The event “ifargm” does the test. The ts is saved to nx as the operation uses ts. When the operation is completed, ts is restored from nx.

```
<inc>
  ts->nx, v[arg]->ts, ifargm <inc2>  ; save ts to nx
  alu(ts+1)->ts                      ; op on v reg
  ts->v[arg], nx->ts, pc+1, <fetch>
<inc2>
  alu(fp-arg)->tbus, mR(tbus)->ts    ; a normal op
  alu(ts+1)->ts
  alu(fp-arg)->tbus, ts->mW(tbus)
  nx->ts, pc+1, <fetch>

<dec>
  ts->nx, v[arg]->ts  ifargm <dec2>
  alu(ts-1)->ts
  ts->v[arg], nx->ts, pc+1, <fetch>
<dec2>
  alu(fp-arg)->tbus, mR(tbus)->ts
  alu(ts-1)->ts
  alu(fp-arg)->tbus, ts->mW(tbus)
  nx->ts, pc+1, <fetch>
<sys>
<array>
<end>
  trap, pc+1, <fetch>
```

14

Performance

The table x shows the number of cycle used by each instruction. The number in parentheses is the number of cycle of the original Sx for comparison. Please observe that almost all instructions are faster. The “call/fun”, “ret” are slow in the worst case, for example, call+fun is 16 cycles (Sx is only 8 cycles). “inc” and “dec” is normal case are the same as Sx (due to the test for the range of argument) but they are twice as fast if the argument is in the cache register.

Table x The number of cycle used by each instruction of Sx2. (n) shows the number of Sx.

bop 3 (4)	uop 2 (3)	get 3 (4)	get1..4 2 (4)
put 3 (4)	put1..4 2 (4)	ld 3 (4)	st 3 (4)
ldx 3 (4)	stx 5 (8)	lit 2 (4)	jmp 2 (2)
jt 3 (4)	jf 3 (4)	call max 7 (8)	fun max 9 (0)
ret max 12 (8)	retv max 12 (7)	retv max 12 (7)	inc1..4 3 (6)
dec 6 (6)	dec1..4 3 (6)		

A number of benchmark programs are compiled and then run on the Sx2 processor simulator. The table below reports the number of instruction, the number of cycle and the cycle-per-instruction number for each program.

Table x the performance of Sx2 processor

```
program noi/clk/cpi    sx1        sx2

bubble   10068   44214 4.39  10262   32090 3.13
hanoi     2312   10092 4.37   2377    7544 3.17
matmul    3043   12880 4.23   3097    9348 3.02
perm      4868   20932 4.30   4935   14663 2.97
queen   618665 2576210 4.16 620724 1717782 2.77
quick     3172   13539 4.27   3224    9551 2.96
sieve    28026  124338 4.44  28029   75204 2.68
aes      30579  131560 4.29  30724   90498 2.95
```

<fig some graph???>

The average cpi of Sx2 is 2.9. From the table, comparing the number of clock between the original Sx and Sx2, the average ratio is 0.70. That is, Sx2 is 30% faster than the original Sx.

Other interesting observation is the size of microprogram. Sx2 is obviously more complex. The size of its microprogram is larger. We calculate the size of microprogram as the number of bit in the ROM. Here is the comparison.

Sx	width 38	length 62	38x62 = 2356 bits
Sx2	width 71	length 74	71x74 = 5254 bits

Therefore, the complexity in the control unit of Sx2 is double of Sx.

Summary

To improve the performance of Sx processor, we employ the technique of stack frame caching. The stack frame caching relies on the fast register to cache a part of 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.

Further reading

The conventional approaches to performance enhancement are to use pipeline and multiple functional units. These techniques have been used successfully in every commercial processor available today. Most computer architecture textbook described these methods. The most widely used text written by the computer architects who invent the concept of reduced instruction set computer (RISC), is the text by Hennessy and Patterson [ref]. The pipeline technique is perhaps the earliest technique for performance enhancement. It has been used for many complex functional units such as floating-point calculation [ref Kogge].

Multiple functional units were the landmark of “super computer” in its era. In fact, the first one to employ multiple function units successfully is CDC6600, the most exciting computer architecture of its day [ref].

References

[1] [my ncsec 2006 paper on stack frame caching] (being written)

[2] [my combined instruction paper] Chongstitvatana, P., "Post processing optimization of byte-code instructions by extension of its virtual machine", 20th Electrical Engineering Conference, Thailand, 1997.

[3] [my series of performance improvement of stack processor] Sattayawiboon, C., Sripornprasert, J., Tansutthiwess, S., Tonteerawong, P., and Chongstitvatana, P., "A stack processor with integrated display circuit for a low cost CD-ROM reading device", ECTI International Conference, May 10-13, Thailand, 2006.

[4] Chongstitvatana, P., "A compact code 16-bit processor for embedded applications", Joint conf. of computer science and software engineering, Nov 2005, Thailand.

[5] Nanthanavoot, P., Burutarchanai, A., and Chongstitvatana, P., "Instruction packing for a 32-bit resource efficient processor," National Science and Technology Development Agency (NSTDA) Annual Conference, Thailand, 27-30 March 2005 (in Thai).

[6] 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.

[7] Burutarchanai, A., Kotrajaras, V. and Chongstitvatana, P., "A fast instruction fetch unit for an embedded stack processor", Proc. of Int. Conf. on Information and Communication Technologies (ICT 2004), 18-19 November, 2004. Thailand.

[8] Burutarchanai, A., and Chongstitvatana, P., "Design of a two-phased clocked control unit for performance enhancement of a stack processor", National Computer Science and Engineering Conference, Thailand, 21-22 Sept. 2004, pp.114-119.

[9] Nanthanavoot P. and Chongstitvatana, P., "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.

[10] Wongsiriprasert, C. and Chongstitvatana, P., "Performance comparison between two virtual machine interpreters : stack-based vs. register-based", Proc. of 3rd Annual National Symposium on Computational Science and Engineering, Bangkok, 1999, pp. 401-406.

[11] CAQA text

[12] Kogge pipeline

[13] CDC6600

[IBM94] International Business Machines, Inc., The PowerPC architecture: A specification for a new family of RISC processors. San Francisco: CA, Morgan Kaufmann, 1994.

[ILI82] Iliffe, J., Advanced computer design, Prentice-Hall, London, 1982.

[LEV89] Levy M., and Eckhouse, R., Computer programming and architecture: the VAX, Bedford, Mass., Digital Press, 1989

[LUN77] Lunde, A., “Empirical evaluation of some features of instruction set processor architecture”, Comm. of the ACM, March 1977.

[PAT82] Patterson, D., and Sequin, C., “A VLSI RISC”, Computer, 15, no. 9, September, 1982, pp. 8-21.

[PAT85] Patterson, D., “Reduced instruction set computers”, Comm. of the ACM, 28, no.1, January 1985.

[STA88] Stallings, W., “Reduced instruction set computer architecture”, Proc. of the IEEE, vol. 76, no. 1, January 1988, pp. 38-55.