# Chapter 7

# Performance Enhancement

In this chapter, we will concentrate on performance of the processor. Performance improvement starts with an analysis of the execution profile to understand where the data path spends most of its 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 choices of design. Choosing one will affect another. The gain in terms of performance must be weighted against the increased complexity in terms of the circuit size or the resource. 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.

## Profile analysis

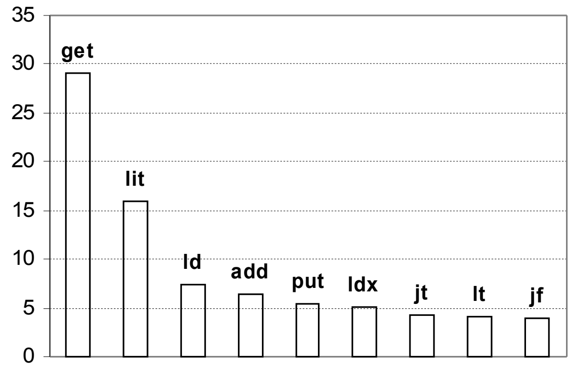

Figure 7.1 The most frequently used instructions

The profile is collected from running the benchmark programs in Chapter 6 (Table 6.3). In an analysis of the profile of execution of instructions (Fig. 7.1), most frequently used instructions are:

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

These instructions altogether are more than 80% of all instructions executed in the suite of benchmark programs[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>
 pop ff
  alu(ts op ff)->ts

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

<ldx>                          ; {ads idx}
```

[1] Notable is the "`inc v`" instruction which is not generated from this compiler (gen.txt). It is used often but has not been included in this experiment.

```
  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 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 Sx processor, `push` and `pop` do one memory access and use ALU to do increment/decrement *`SP`*.

## Key ideas

There are two key ideas to improve the performance of Sx data path.

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 most frequently used instruction, 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. 7.3).

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.

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

With the new scheme, it 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 x is
  mR(sp)->x
  sp-1->sp
```

It becomes

```
$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 to push `TS`, the second one for getting the value. Therefore "`ld`" takes 2 cycles.

```
<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 transfers 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

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. 7.2).

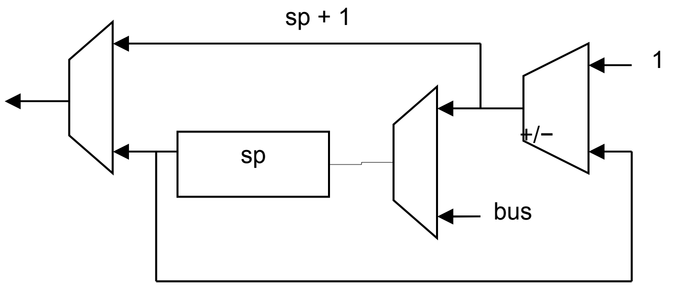

Figure 7.2 The SP unit

## Stack frame

A number of registers are used to cache a part of stack frame. This is called the *stack frame caching* [CHO06]. 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. 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. The execution steps of `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 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 O(3(*n*+1)).

## New data path

The enhanced Sx, or Sx2, has many additional functional units (Fig. 7.3), 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 in the data path 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`}

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 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

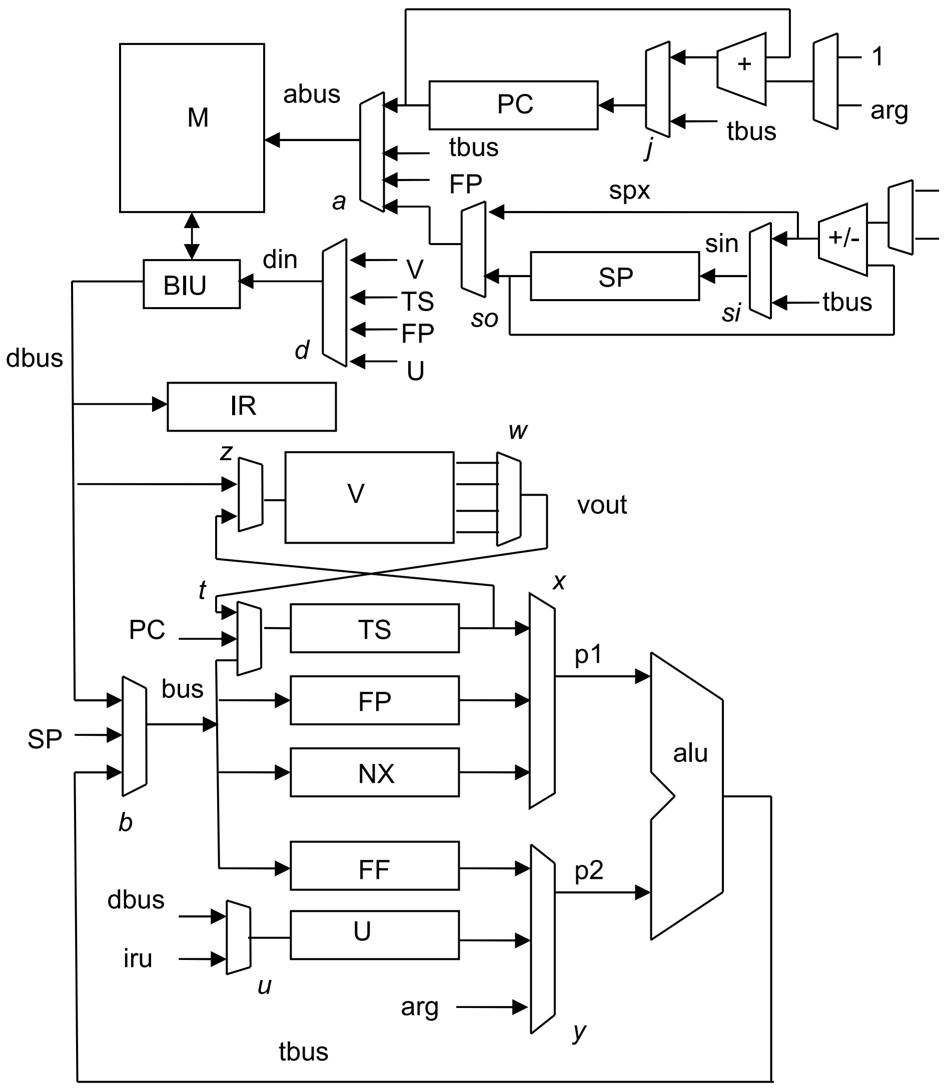

Figure 7.3 The Sx2 data path

`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`

| 8 | 8 | 8 | 8 |
|---|---|---|---|
| p | u | k | op |

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>  [micro 205]
  mR(pc)->ir, decode
```

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

```
<bop>  [micro 207]
  mR(sp)->ff, sp+1->sp
  alu(ts op ff)->ts, pc+1, <fetch>

<uop>  [micro 210]
  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>  [micro 212]
  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>  [micro 223]
  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>  [micro 235]
  ts->mW(sp+1), sp+1->sp
  mR(arg)->ts, pc+1, <fetch>

<st>  [micro 238]
  ts->mW(arg)
  mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<ldx>  [micro 240]                    ; {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>  [micro 243]                    ;  {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>  [micro 247]
  ts->mW(sp+1), sp+1->sp, arg->ts, pc+1, <fetch>

<jmp>  [micro 249]
  pc+arg, <fetch>

<jt>  [micro 251]
  alu(ts=0), ifT j3                 ; if true, don't jump
<j2>
```

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

```
<jf>   [micro 256]
  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>   [micro 261]                     ; 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 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>   [micro 270]                      ; 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>   [micro 281]
  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
```

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>  [micro 300]
  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>  [micro 310]
  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>  [micro 320]
<array>
<end>
  trap, pc+1, <fetch>
```

## Performance

Table 7.1 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`" in a 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 7.1 The number of cycle used by each instruction of Sx2. (n) shows the number of cycle 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 (noi), the number of cycle (cycle) and the cycle-per-instruction number (cpi) for each program.

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 | $38 \times 62 = 2356$ bits |
| Sx2 | width 71 | length 74 | $71 \times 74 = 5254$ bits |

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

Table 7.2 The performance of Sx2 processor

| | **Sx** | | | **Sx2** | | |
|---|---:|---:|---:|---:|---:|---:|
| program | noi | cycle | cpi | noi | cycle | cpi |
| 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 |

## Summary

To improve the performance of Sx processor, 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.

## 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 [HEN03]. 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 [KOG81]. 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 [THO70].

## References

[CHO06] Chongstitvatana, P., "Stack frame caching", Proc. of National Computer Science and Engineering Conference, Thailand, 2006.

[HEN03] Hennessy, J., and Patterson, D., Computer Architecture: a quantitative approach, 3rd ed. Morgan Kaufmann, 2003.

[KOG81] Kogge, P., The architecture of pipelined computers, McGraw-Hill, 1981.

[THO70] Thornton, J., Design of a computer: the Control Data 6600, Scott, Foresman and Company, 1970.

## Exercises

7.1 Run Sx2, try to write a microprogram for some new instruction and test it.

7.2 Compare the performance of new instructions in Exercise 7.1 with Sx.

7.3 Discuss the finding, suggest some way to improve the performance by adding some new instruction (counting the total cycle used to complete a task).

7.4 Improve the microstep of some instruction. You don't have to simulate the execution. You can calculate the number clock from the profile.

7.5 If the number of cache registers is changed, for example, 8, what is the impact on the performance?

7.6 The memory latency is one of the most important factors in determining the performance of a processor. Suppose the latency of memory is increased to 2 cycles for read and write to memory. What is its impact on performance? Assume the cache register latency is one cycle.