The Art of Instruction Set Design

Prabhas Chongstitvatana
Department of computer engineering
Chulalongkorn University
Phayathai Road, Bangkok 10330, Thailand
E-mail: prabhas@chula.ac.th

Abstract

I will argue in this work that because of the flexibility offered by the field programmable devices, such as FPGA, the instruction set design becomes an essential part of designing digital systems of the future. There are advantages to experiment with micro-architecture and instruction set design to fit various applications. I will illustrate by a series of evolution of instruction set design for mobile devices. Driven by the mobile code requirement, the instruction set and micro-architecture of various alternatives are evaluated and compared. Some of the latest ideas will be described, including the evaluation of a proposed instruction set plus its micro-architecture.

Keywords: instruction set, code compression, mobile devices, stack machines.

1. Introduction

As PC becomes ubiquitous, the computing platform are becoming uniform. Only a few designs dominate the market for microprocessor chips. The present instruction set architecture (ISA) for common platforms are converged, there are not much differences between leading instruction set architectures. The conclusion seems to be that the instruction set design is not an active area of research.

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. However, the market force reduces the number of developers of new processors and new instruction set. Therefore, the platforms and their instruction sets are converged. Conclusion, the instruction set design is not an active area of research.

Contrary to the above views, I would like to present a different view. As programmable devices, such as the field programmable gate array (FPGA) devices, become popular, it enables a designer to innovate with new instruction sets. There are advantages to experiment with micro-architecture and instruction set design to fit various applications. These alternatives can offer many advantages such as application specific performance, the small code size, the reduction of chip resources, the custom made functionality etc. So, it is the aim of this work to try to convince the reader that the instruction set design and the special purpose micro-architecture is still an interesting and worthwhile area for research.

In the last decade the progress of microprocessor design has been phenomenal. Performance rises according to Moore's law. The performance is double every 18 months. Performance is the key driving force for the progress of the last decade. However, the new applications have shifted the design landscape once again to low power, portable devices [1]. The technology for implementing computational devices in small batch with fast turnaround time has opened up the issue of ISA design.

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 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.

I will illustrate through a series of evolution of instruction set architecture, what is achievable to meet different requirements by adapting the instruction set and micro-architecture. To reduce the confusion of the use of term "instruction set" which is used to mean both the virtual machine instruction set and the hardware level instruction set, I define instruction set to be instructions that can be directly executed on the hardware. The purpose of an instruction set is to be the interface between applications written in a high-level language and the actual processor.

2. Base instruction set

I will start with one instruction set which begins its life as a virtual machine. R1 [3] is a simple language which provides concurrency control, protection of shared resources, interprocess communication and real-time facilities. This language is designed for programming embedded applications. R1 instruction set is stack-based and byte-coded. The table below shows R1 instruction set with the following format: [instruction #argument] meaning.

Notation: CS code segment, DS data segment, SS stack segment. Both data segment and stack segment resided in the same address space and can be denoted by M.

Local variables are accessed through FP, the frame pointer, denotes the pointer to the activation record. IP denotes the instruction pointer.

Table 1 R1 instruction set

```
[Literal #n ]          push( n )
[Lvalueg #ref ]        push( ref )
[Lvalue #i ]           push( FP-i )
[Rvalueg #ref ]        push( DS[ref] )
[Rvalue #i ]           push( SS[FP-i] )
[Fetch]                push( M[ pop ] )
[Set]                  M[ pop1 ] = pop2
[Index]                push( base_ads + index )
[Jmp #ads ]            IP = ads
[Jz #ads ]             if pop = 0 then IP = ads
[Call #ads ]           push( IP ), IP = ads
[Func #nparam #nlocal ]     function call
[Proc #pid #npara #nlocal]  create new process
[Ret0]                 remove stack frame, restore state
[Ret1]                 similar to Ret0 but return a value
[Stop]                 terminate the process
[Aop]                  push ( pop1 Aop pop2 )
[Lop]                  push ( pop1 Lop pop2 )
[Uop]                  push ( Uop pop )
Aop (arithmetic operators): Add, Sub, Mul, Div.
Lop (logical operators): Lt, Le, Eq, Ne, Ge, Gt, And, Or.
Uop (unary operators): Minus, Not.
[Send ]                send message
[Receive ]             receive message
[Wait #sem]            wait on semaphore
[Signal #sem ]         signal semaphore
[gettime] store system time to address
[delay ]               send a timer
[tmwait]               wait with time-out
[tmsend]               send with time-out
[tmreceive]            receive with time-out
```

It is worth noticing that all real-time facilities are provided in the instruction set. This instruction set is a minimalist style, only 39 instructions are necessary to support the requirement of the source language. The use of stack-based instruction simplifies the addressing mode. R1 is an instruction set for a virtual machine. It is implemented as an interpreter running on a target device. However, this is not necessary the only way to realize an instruction set. Currently, we have created a direct realization of this instruction set in the actual hardware. The prototype, with 16-bit data path and omitted the real-time facilities, can be synthesised on a 20,000 gates field programmable gate array (FPGA) device running 20-40 MHz.

3. Extended code

To improve the performance of the base instruction set, some new instructions are added. In [4], some sequence of byte-codes can be represented by a shorter code which can be executed faster. The benchmark programs are profiled and the most frequently used sequences are collected (see Table 2)

We classified these sequences into 4 classes :

1. increment, decrement and combined operators (such as "+=" in C language).
2. array access
3. assignment
4. flow control

Table 2 The most frequently used sequences

byte-code sequence	correspond to
lval a, rval a, lit 1, plus, set.	a = a + 1;
lval b, lvalg c, ... , index, ...	b = c[..] ...
lvalg c, ... , index, ...	c[..] = ...
lval a, lit 0, set	a = 0;
lvalg c, ..., index, lit 0, set	c[..] = 0;
lval a, rval a, exp, plus, set	a = a + exp;
lvalg c,..,index,lvalg c,..,index, fetch,..,plus set	c[n] = c[n] + ...
rval a, rval b, EQ, Jz	if (a == b)
rval a, lit 0, EQ, Jz	if (a == 0)
lvalg c, ..., index, rval b, LE, Jz	if (c[..] <= b)
rval a, rval b, LT, Jz	while (a < b)

Corresponding to these sequence of codes, a number of extended byte-codes are designed. Totally 21 instructions are added to the instruction set. Table 3 shows the extended codes. The experiment shows that this technique yields 25% - 120% speedup (means it is faster as much as 2.2 times as before the optimisation) with 10% - 30% code size reduction (varies across the benchmark programs).

Table 3 Extended code

extended code	for the sequence
inc v (dec v)	lval v, rval v, lit 1, plus, set.
addset a	lval a, rval a, exp, plus, set.
set-var a	lval a, ... set.
set-0 a	lval a, lit 0, set
EQjz a b $1	rval a, rval b, EQ, jz $1
Jnz a $1	rval a, lit 0, EQ, jz $1
LEjz a b $1	rval a, rval b, LE, jz $1
LTjz a b $1	rval a, rval b, LT, jz $1

4. Register-based instruction set

In contrast to the stack-based instruction, the register-based instruction set has dominated the landscape of instruction set design in the last decade. In [5] we compared the stack-based instruction with the register-based instruction set. The work intended for the virtual machine implementation of the instruction set. We observed that for a stack-based machine the performance limit of the interpreter is likely to be the fetch-limit, i.e. the time spending on fetching and decoding an instruction. To improve the performance the number of executed instruction should be reduced. This can be achieved by designing an instruction set that each instruction performs as much work as possible.

To achieve this goal, a register-based instruction set (RVM) is designed. In the register-based architecture an instruction has access to a number of operands in the

registers instead of limit to the access to the stack. The registers can be accessed randomly unlike the stack.

The RVM is a 3-operand register machine. It is a load-store architecture with 32 registers. The operand in the opcode can be accessed using 5 addressing modes: (1) absolute, (2) intermediate, (3) base/index, (4) base/displacement and (5) register deferred. The RVM has 17 simple instructions. All of the instructions support only 32-bit word data type. The instructions are encoded using 32-bit fixed length encoding. Figure 1 shows the RVM instruction set.

```
op:5 mod:2 rd:5 rs1:5 rs2:5 ud:10
op:5 mod:2 rd:5 rs1:5 imm:15
op:5 mod:2 rd:5 ads:20
op:5 mod:2 rd:5 rb:5  disp:15
op:5 mod:2 rd:5 rb:5 rx:5 ud:10

Data transfer: LOAD, STORE, LOADI, SAVE, RSTO.
Control flow: CALL, JUMP, HALT
Alu op: ADD, SUB, AND, OR, XOR, SHIFT, MUL, DIV.
Others: SAVE, RSTO save and restore value of registers
```

Figure 1 RVM instruction set. rd = destination register, rs = source register, rb = base register, rx = index register, imm = immediate value, ads = address, disp = displacement, ud = undefined.

RVM is designed to minimise the dynamic instruction count. Using the similar benchmark suite to the R1 extended code experiment, the experiment shows that RVM has reduced the instruction count 35 - 60% compared to stack-based instruction set, accordingly, the performance of RVM is is 1.5 to 2 times faster than the stack-based virtual machine interpreter. The RVM is realizable as a hardware-level instruction set as it can be mapped directly to almost any 3-operand ISA that exists today.

5. Code compression

The other requirement of mobile devices is the compact code size. Reducing the size of machine code has benefit in two aspects. The first one is obvious in reducing the storage requirement, both in code segment and in instruction cache memory. This is often the reason behind many classic instruction set architecture, to achieve very compact executable code. The second one is related to power requirement. As the instruction bandwidth is reduced, the power consumption is also reduced [6, 7].

How to make the program as small as possible? In [8] we proposed a method, called "nibble coding", compresses two instructions into one byte. The experiment is carried out to compare conventional byte-code instruction and a typical 32-bit machine code with the nibble coding. The result shows the proposed scheme achieves a smaller instruction bandwidth than a byte-code virtual machine and is much smaller than the conventional executable machine code. The reduction is 50% and 57% of static and dynamic code size. In other words, using the static code size is half of the normal code and the dynamic code size is less than half of the normal code.

The method to pack instructions into a smaller space is based on the following techniques:

1. Extended instruction, making a special instruction that replace several simple instructions.
2. Specialization, eliminate arguments by making an instruction special to a particular argument thus reduce the size of instruction.
3. Reduce the size of arguments, by using a literal table that stores a number of full-size arguments. The argument of the instruction can be replaced by the index into this table. The index is much smaller than full-range of argument as there are small finite number of different variables in a program.
4. Packing two instructions into one instruction, this technique is called "nibble coding".

We also investigate the use of nibble coding technique with Java byte-code based on a subset of JVM [9]. The details of the technique and the results are reported in [10]. The nibble code format is shown in Figure 2.

```
normal and extended
0|op:7                      zero argument
0|op:7 a2:8                 local: get, put, inc, dec
0|op:5|a1:2  a2:8           jmps, call, ld, st

nibble
1|op1:3|op2:4               zero argument
1|op1:3|op2:4 a2:8          one argument
```

Figure 2 Instruction encoding of the nibble code.

6. Concern with micro-architecture

With byte-coded instruction set, the data path is simplified, the complexity is shifted to the control unit. I draw an example from the micro-architecture of the prototype 16-bit stack-based processor. The high level description language for this chip is only 5 pages long but the control unit is 22 pages long!

The choice of instruction set for a stack machine has a close relationship to a high level language. The "semantic gap" between its machine language and a high level language is narrow. One can almost write a stack machine language directly from a high level language source program. The higher semantic content, especially on the function call and parameter passing, helps to simplify the task of programming. See the following example:

```
to sum a b | s =    //sum from a to b, s is a local
  s = 0
  while a <= b
    s = s + a
    a = a + 1
```

```
to main =
  print sum 1 10
```

can be written in a machine code for our stack machine as follows:

```
:sum
lit 0, put s,
:loop
get a, get b, <=, jF exit, get s, get a, +, put s, get a,
lit 1, +, put a, jmp loop
:exit
get s, retv

:main
lit 1, lit 10, call sum, call print
end
```

7. Current work

The R1 extended code and RVM are comparable in term of performance. To investigate the issue of performance improvement further we are experimenting with a stack-based register machine (SR).

A stack-based instruction set relies on an evaluation stack to store arguments and intermediate values. For local variables, an activation record is used to give direct access to local variables. Because a stack has only one port to access it, and because the stack is implemented in the memory, it slows down the computation. If registers are used in places of the activation record to store local variables, and the instruction set can access to these registers directly, it will not be necessary to use a stack. However, the registers used in the callee must be saved and restored. The activation record has advantage that local variables need not to be saved and restored as the creation and deletion of the activation record occurs naturally on call/return of functions. Combining storing local variables in registers and last-in-first-out (LIFO) behaviour of activation records is the stack-based register machine.

7.1 How it can be accomplished?

To access a variable in an activation record, a frame pointer is needed. The access to a local variable is indexed by an offset from this frame pointer. To enable registers to have this behaviour, there must be a "register renaming" mechanism. With register renaming, all register indexing is relative to a "frame pointer". The register bank can be implemented as a circular buffer. The visible registers set are finite and are stored in the buffer. When the buffer is full it can be spilled into the memory, and vice versa for the underflow of buffer. Hence, the "stack segment" is in the memory with the front of the segment cached in the register buffer.

7.2 SR Instruction Set

The instruction set becomes similar to a 3-operand register instruction set. From a programmer's point of view, this is a register machine with the twist that no registers need to be saved/restored on call/return. The activation record creation/deletion is done in hardware and also the spilling/pulling between registers-memory is done transparently to the programmer. Because all local variable accesses are to registers it is fast. Occasionally, spill/pull to memory is necessary but by amortized analysis, it should be a net win (reducing the number of memory access). Figure 3 shows SR instruction set. All instructions are 32-bit, fixed length.

```
op:6 r1:5 r2:5 r3:5 xop:11   (register)
op:6 r1:5 r2:5 imm:16        (immediate)
op:6 r1:5 ads:21             (absolute)
op:6 lads:26                 (long)

addressing mode   mnemonics       meaning
absolute          ld r1 ads       r1 = M[ads]
displacement      ld r1 @disp r2  r1 = M[disp + r2]
index             ld r1 +r2 r3    r1 = M[r2 + r3]
```

Figure 3 SR instructions set

7.3 Register renaming

The size of register bank is determined by the amount of resources. To implement "register windowing", the register buffer is used. The register buffer must be larger than this size so that more than one activation record can be resided in the buffer. A register FP is a rename register. FP is similar to a frame pointer. The mapping of a register name to the actual register is:

$$r_n = R[FP + n]$$

where R[.] is the register bank. The width of FP is lg(size of register bank), 8 bits for a 256-register buffer.

7.4 Register spilling

When the register buffer is full, to get more registers, the older registers are spilled into the memory. Allocating more registers occur at the function call to create an activation record. At the return from a call, the activation record is deleted. An underflow of register deallocation can occurs. The value of registers will be pulled from the memory.

The size of register bank is the amount of register a programmer will see (determine by the number of bit of register addressing, 5 bits is 32 registers). The size of the buffer is determined by the resource of a processor (for example, 256 registers). The register buffer is implemented as a circular LIFO list. There are two pointers: F (Front), B (Back). F and B always point to the existing elements in the list. Initially F = B = 0. Let n denotes the size of the register buffer, m denotes the request for allocating new register for a new activation record.

The following invariance holds:

$$F' = (F + n) \bmod n = F$$

If the request for m registers occurs, the allocation will be:

F' = F
F = (F + m) mod n
return F'

F' points to the beginning of the newly allocated block. And vice versa for freeing m registers:

F' = (F - m) mod n

When an overflow occurs the oldest block is saved to the memory.

F' = (F + m) mod n

if F < B then ovf = (F' < F) or (F' >= B)
else ovf = (F' < F) and (F' >= B)

and vice versa for underflow.

F' = (F - m) mod n

if F < B then udf = (F' > F) and (F' < B)
else udf = (F' > F) or (F' < B)

Example the previous sum(a,b) program can be written in SR:

```
:sum
mv i #1, mv s #0
:while
jgt i s exit, add s s i, add i i #1, jmp while
:exit
```

It is 6 instructions and only 4 instructions are in the loop. Compare this to the R1 instructions, the number of instructions executed is less than half.

7.5 Experimental results

The following benchmark programs are used:

bubble	**sort 20 items**
hanoi	**move 6 disks**
matmul	**multiply two 8x8 matrices**
perm	**permuting 4 digits of 0,1,2,3**
queen	**all solutions of 8-queen**
quick	**sort 20 items**
sieve	**find all primes <= 500**

Comparing the number of executed instructions between R1 and SR shows that the reduction in the number of executed instruction of SR over R1 is 50% - 80%. In other words, SR executed 2 times to 5 times less instruction than R1. This result agrees with the previous result [5] that the register-based instruction reduces the number of executed instruction 35% - 60%. (see Table 4)

Table 4 Comparison of the number of executed instructions between R1 and SR

	R1	SR	SR/R1
bubble	**13879**	**4621**	**0.33**
hanoi	**2631**	**1313**	**0.50**
matmul	**20608**	**7109**	**0.34**
perm	**6514**	**2030**	**0.31**
queen	**753261**	**282458**	**0.37**
quick	**3522**	**851**	**0.24**
sieve	**18665**	**3744**	**0.20**

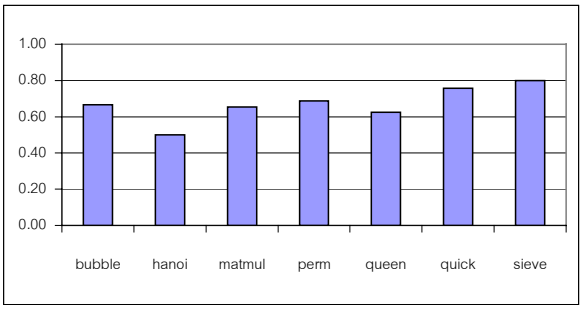

Figure 4 The reduction of the number of executed instruction SR over R1

The effectiveness of using the register buffer is measured in Table 5. The figures show that the number of access to the memory (in term of accessing the activation record) is greatly reduced. In all cases, it is reduced to zero using the register buffer size of 128 for this benchmark (the benchmark is not a realistic one because the size is too small). Please remember that for any implementation of stack, there will be some spilling to the memory because the depth of call is determined at run-time. One can compares this scheme of register buffering with the compile time saving/restoring registers to an internal (on chip) stack.

Table 5 The number of register spilling.

buffer size	**16**	**32**	**64**	**128**
bubble	**0**	**0**	**0**	**0**
hanoi	**98**	**16**	**0**	**0**
matmul	**0**	**0**	**0**	**0**
perm	**26**	**0**	**0**	**0**
queen	**17991**	**10437**	**2507**	**0**
quick	**70**	**43**	**11**	**0**
sieve	**0**	**0**	**0**	**0**

8. Related work

There are volume of work on customised instruction set for specific applications for example [11, 12] including using programmable gate array for realizing these instruction set embedded in an ordinary processor [13]. The flexibility of customizing instruction set for specific applications has been commercialized by a number of companies for example Xtensa [14]. For multimedia work load the most well-

known is MMX instruction from Intel [15]. These are aimed for high performance. Portability issue is investigated in [16]. The object code is a highly compact, architecture-neutral intermediate program representation which is used to generate native code of high quality on-the-fly. Many works investigate the code compression, aiming to reduce the size of executable code [17 ,18]. For example, Pugh [19] achieved compressing of Java classfiles by a factor of 2 to 5. The proposed stack-based register machine is reminiscent of PicoJava chip [20].

9. Conclusion and future work

It has been illustrated in this work, that the instruction set design is a vital part for future generation of hardware devices. The flexibility offered by the field programmable devices allows designers to explore a different architectural landscape. Low power design is currently one of the most active research in the processor design and implementation. There are many indicators that the instruction set design has a role to play in this area [21, 22]. Future processors will be totally different from what we know today. We have proposed one design in which the flexibility of programmable devices is exploited. By time-multiplexing the circuits into a limited resource, high performance can be achieved at the same time as conversing resources and energy [23]. I think we are now in the age of energy-aware computing. Anybody wants to design a processor that adapt itself to the amount of available energy?

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] P. Chongstitvatana, "A multitasking environment for real-time control", The Engineering Research Fund, Faculty of Engineering, Chulalongkorn University, research project number 132-MRD-2537. http://www.cp.eng.chula.ac.th/faculty/pjw/r1/

[4] P. Chongstitvatana, "Post processing optimization of byte-code instructions by extension of its virtual machine", Conf. of Electrical Engineering, Bangkok, 1997.

[5] C. Wongsiriprasert, P. Chongstitvatana, "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.

[6] R. Gonzalez, "Low-power processor design", Technical Report No. CSL-TR-97-726, June 1997, Computer Systems Laboratory Departments of Electrical Engineering and Computer Science, Stanford University.

[7] R. Krishnamurthy, "Mixed Swing Techniques for Low Energy/Operation Datapath Circuits", Ph.D. thesis, Electrical and Computer Engineering, Carnegie Mellon University, December 1997.

[8] P. Chongstitvatana and V. Kotrajaras, "Instruction compression by nibble coding: war on the old front", IEEE Thailand section: Silver Jubilee Symposium, 15 Nov 2002.

[9] B. Joy (Ed), G. Steele, J. Gosling, G. Bracha, Java(TM) Language Specification (2nd Ed), Addison Wesley Pub., 2000.

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

[11] R. Leupers and J. Elste and B. Landwehr, "Generation of interpretive and compiled instruction set simulators", Proc. of the Asia and South Pacific Design Automation Conference, Jan. 1999.

[12] S. Pees, A. Hoffmann, V. Zivojnovic and H. Meyr, "LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures", Design Automation Conference, 1999, pp. 933-938.

[13] T. Glokler and S. Bitterlich, "Power Efficient Semi-Automatic Instruction Encoding For Application Specific Instruction Set Processors", ICASSP, 1999.

[14] R. Gonzalez, "Xtensa: a configurable and extensible processor", IEEE Micro, March/April 2000, p.60

[15] MMX technology, http://developer.intel.com

[16] M. Franz and T. Kistler, "Slim binaries", Comm. of the ACM, vol.40 no. 12, Dec. 1997, pp.87-94.

[17] W. Evans and C. Fraser, "Bytecode Compression via Profiled Grammar Rewriting", in ACM Sigplan Conference on Programming Language Design and Implementation, 2001, pp.148-155.

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

[19] W. Pugh, "Compressing java classfiles", In ACM SIGPLAN Conference on Programming Language Design and Implementation, 1999, pp.247-258.

[20] H. McGhan and M. O'Conner, "PicoJava : a direct execution engine for Java bytecode", IEEE Computer, Vol.31 No. 10, 1998.

[21] D. Kirovski, C. Lee, M. Potkonjak, W. Mangione-Smith, "Synthesis of Power Efficient Systems-on-Silicon", Asia and South Pacific Design Automation Conference, 1998, pp.557-562.

[22] T. Burd and R. Brodersen, "Processor design for portable systems", Journal of VLSI Signal Processing, 13(2/3):203-222, August 1996.

[23] K. Piromsopa, P. Bavonparadon, P. Chongstitvatana, "Hardware multiplexing: towards a resource efficient reconfigurable processor", 3rd Inter. Symposium on Communications and Information Technologies, Thailand, 2003.