Instruction Packing for a 32-bit Stack-Based Processor

Witcharat Lertteerawattana, Tanes Jedsadawaranon and Prabhas Chongstitvatana
Department of Computer Engineering, Chulalongkorn University
Bangkok 10330, Thailand
email: prabhas@chula.ac.th

Abstract - **This work proposed a design and development of a 32-bit stack-based processor for embedded systems. A reference processor has a 32-bit stack-based instruction set. This work proposed a technique of instruction packing which packs several instructions into one 32-bit instruction unit. Therefore, the instruction size is reduced. The result of the experiment shows that the proposed technique achieves around 30% reduction in code size.**

Keywords**:** Instruction packing, Byte code instruction, Stack-based processor

1. INTRODUCTION

A stack-based processor has many advantages such as its instruction set is close to high-level language, the data path is simple and it is low cost to implement. Its disadvantage is the limited performance because its instruction set operates on the evaluation stack which accesses the memory. This problem can be reduced with the instruction packing technique. Instruction packing is the assembling of many instructions into one unit. By reducing the size of instruction, there will be less instruction fetching form memory. This proposed technique is applicable to a general stack-based processor because the changes are only in the control unit. The proposed processor has been designed and implemented based on FPGA devices. The cycle accurate simulation shows that the program size is reduced about 30% compared to the reference processor.

2. REFERENCE PROCESSOR

The reference processor is a 32-bit stack-based processor [1]. This design is a descendant of an earlier 16-bit stack-based processor [2]. The evaluation stack is kept in the memory. The data path of the reference processor is simple. The processor is aimed to be a teaching tool at the architectural level. It is not aimed for high performance.

This reference processor has eight special purpose registers: `IR, PC, TS, SP, FP, NX, FF` and `AA`

`IR`	: Instruction Register
`PC`	: Program Counter register
`TS`	: Top of Stack register
`SP`	: Stack Pointer register
`FP`	: Frame Pointer register
`NX`	: First temporary register
`FF`	: Second temporary register
`AA`	: Array Allocation register

The `TS` register stores the top of stack value. The `SP` register stores the pointer to the second value in the stack (below the top of stack). `FP` register, as its name, keeps the frame pointer. `NX` and `FF` registers are used when some instruction needs a temporary storage during computing the result from ALU. The `AA` register stores the array pointer for dynamic memory allocation. The data path consists of one ALU that connects to the register bank and the result from ALU is connected to `tbus` and `tbus` connects to the register bank.

The `PC` register is designed with an auto- increment module, so fetching instruction can be done in one clock cycle. It also can be loaded with the value `PC+arg` or `tbus`. The memory interface is through Bus Interface Unit (BIU). The BIU connects data input (`din`) and output (`dout`) from to memory. The `din` can be selected from `TS` or `FP`. ALU is connected with two multiplexers `p1` and `p2` which `p1` interfaces to four registers `TS`, `SP`, `FP` and `NX` and `p2` links with `FF` and the instruction argument value.

3. INSTRUCTION PACKING

In the design of the processor's structure with packed instruction, the first point that needs consideration is the instruction set. How to put many instructions into one unit of instruction so that we can fetch them in one cycle?

The main idea of this design is demonstrated as follows. The base instruction set was designed with byte-code format. The instructions are divided into group according to the size of the operand. There are three patterns: zero operand, one-byte operand and three-byte operand as seen in Figure 1.

In this paper, we call the above pattern as follows:

1. Zero operand: S-format (one byte opcode)
2. One-byte operand: M-format (one-byte opcode, one-byte operand)
3. Three-byte operand: L-format (one-byte opcode, three-byte operand)

As they are variable size instructions and the reference processor fetches at word boundary, there are many waste blank space in the instruction of S-format and M-format (each instruction is allocated four bytes in memory even though the size of those instructions are different).

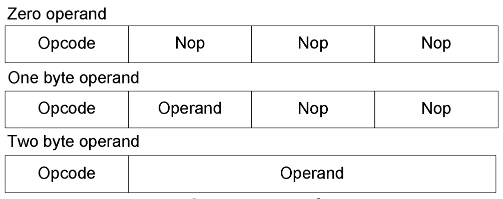

Figure 1. Three pattern of instructions

The proposed design creates new instruction formats that can be fetched four-byte at a time. The new format will have better memory space utilization.

Entry type

The new instruction format will be a superset of the reference instruction set. The new format defines a field called "entry type" which determines the packing pattern. The reference processor has 36 instructions hence the instruction encoding requires six bits. For one-byte opcode there will be the remaining 2 bits for this entry type. The entry type has the encoding as follows:

1. *Entry type = 0* means the follow instruction is NOP (No Operation) and the processor has to fetch a new word for the next instruction.
2. *Entry type = 1* means the follow instruction is S-format.
3. *Entry type = 2* means the follow instruction is M-format.
4. *Entry type = 3* means the instruction is L-format

With this entry type, the instruction would have the pattern as follows:

While considering the entry type, it is necessary to decode the byte header to decide the size of the sub-instruction. Therefore it is important to access each byte freely. The BP (Byte Pointer register) is used to point to the byte that would be accessed next.

A number of additional registers are included in the data path. The `BP` register and the `ARG` register are used to store the argument value. In the packed instruction the argument size is variable and the argument position is not fixed.

Although accessing instruction at the byte-level can be done, there is a problem when performs jump. In an ordinary jump, there are two kinds of jump, M and L-format (short and long jump). The jump instruction generally accesses the code segment at word boundary but the packed instruction needs to access at the byte-level. The modified jump instructions are shown in Figure 3.

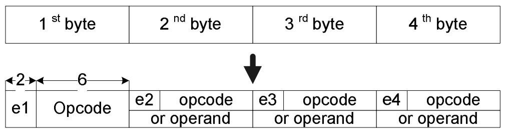

Figure 2. The sub- instructions with entry type.

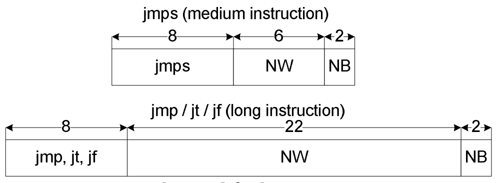

Figure 3. The modified jump instructions.

The *NW* (Next Word argument) is an offset value of the destination word-address. The *NB* (Next Byte argument) points to the next byte in the destination word-address. If a jump is taken, *NB* is saved to the *BP* register. With this instruction encoding, there are 12 new instruction patterns as shown in Figure 4.

The instruction decoding has two states. The flow chart of the first decode state is shown in Figure 5.

The first decoding state, *BP* is tested first, and then the entry type of each byte that *BP* pointed to is tested. If the entry type is zero, go to Pre Instruction Fetch state to prepare to fetch the next instruction. Otherwise, go to the second decoding state (Decode short, medium or long instruction).

After the size of current instruction is known the byte that should be accessed by the first decoding state is determined, *BP* must be updated to the change in the *ARG* register. If it is S-format instruction, *BP* is incremented by one. If it is M-format instruction, *BP* is increment by two. If it is L-format instruction, *BP* is set to 0.

With the new *BP* register, the control steps for the instruction call and return are changed. They are shown below. First, we describe the control step notation.

Control step notation

Each execution step can be notated as follows:

```
src -> dst
```

it denotes the value transferring from source "src" to destination "dst" where src and dst can be a wire or a register. A wire represents a connection or the input/output of a component.

```
alu(a op b) -> dst
```

denotes the ALU operation "op" that performs on two inputs, a and b and then sends the result value to the destination "dst" where dst can be a wire or a register.

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

the *mR(ads) -> dst* means reading value from memory address "ads" and store such value into destination register, the *src -> mW(ads)* means write value from source register into memory address "ads".

L	L opcode	operand	operand	operand
M-M	M opcode	operand	M opcode	operand
M	M opcode	operand	-	-
M-S-S	M opcode	operand	S opcode	S opcode
M-S	M opcode	operand	S opcode	-
S-S-S-S	S opcode	S opcode	S opcode	S opcode
S-S-S	S opcode	S opcode	S opcode	-
S-S	S opcode	S opcode	-	-
S	S opcode	-	-	-
S-S-M	S opcode	S opcode	M opcode	operand
S-M-S	S opcode	M opcode	operand	S opcode
S-M	S opcode	M opcode	operand	-

Figure 4. Pattern of packed instructions

The concurrent execution is denotes by writing them in same line and separated each event from others by comma ",". These events are concurrent, the writing order of each event in the same line is irrelevant.

The short hand notation for SP and PC can denotes as follows:

```
        SP+1 is alu(SP + 1) -> SP
        SP-1 is alu (SP – 1) -> SP
        PC+1 is PC+1 -> PC
        PC+arg is PC+arg -> PC
```

Here are the control steps for call and return instruction.

```
<Call>

SP++
TS->mW(SP), PC++
PC->TS
arg[23:2]->tbus->nx->pc,
  arg[1:0]->bp, mR[tbus]->ir
ir[x:y]->arg, ir[y+1:y+2]->bp
alu(SP+arg)->tbus, FP->mW(tbus)
alu(SP+arg)->tbus, tbus->SP->FP/
decide_state

<Return>

sp->ff
alu(fp=ff), ifFalse /retv
ts->pc
alu(fp-arg)->sp
mR(sp)->tssp-1
mR(fp)->fp, 0->bp /fetch

<Returnv>
(return with return value on stack)

alu(fp+1)->tbus, mR(tbus)->ff
ff->pc
alu(fp-arg)->sp
mR(fp)->fp, 0->bp /fetch
```

4. PERFORMANCE

In the experiment to evaluate the performance of the proposed processor, we use a set of benchmark programs. Table 1 shows the number of cycles for each program of the reference processor compared with the packed instruction processor.

"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. "quick" is a quick sort program with a similar input to "bubble". "hanoi" is a program to solve Tower of Hanoi problem with 7 disks. "matmul" is a matrix multiplication program; the input is two matrices of the size 4 × 4.

Table 1 compares the number of cycles of the two processors. Table 2 compares the size (in byte) of the program of the reference processor and the packed instruction of the proposed processor.

In average, the number of cycles of the proposed processor is 13.7% more than the reference processor. Comparing the size of program between the reference processor and the packed-instruction processor, the packed-instruction is 29.52% smaller.

The augmented processor is 13.7% slower due to the increase in the instruction decoding time. The memory access time in the FPGA board we used for the experiment is very fast so the gain in reducing the instruction fetch is offset by the instruction decoding.

However, the number of instruction fetch is obviously reduced by 33.83% as show in Table 3.

5. RELATED WORK

The current interest in embedded systems spurs a lot of research activities in instruction packing. The work in [3, 4] proposed integrating an instruction register file to decrease code size and improve performance. The software and hardware extension to the instruction register file for supporting multiple instruction register windows allows a greater number of relevant instructions to be available for packing in each function. Others that related to the stack-based processor are [5, 6]. Our earlier work in instruction packing [7] proposed a similar idea presented in this paper but it supported only the word boundary jump.

6. CONCLUSION

The packed-instruction achieved around 30% reduction in code size. Although the performance of the proposed processor is lacking due to the delay in instruction decoding, it is possible to improve the implementation of the decoding state.

In terms of resources used, the proposed processor consumes a little more resource than the reference processor when synthesis on the FPGA devices. The equivalent gate count for this design is 20,699 gates while the reference processor consumes 19,853 gates.

Table 1. Comparing the number of cycles.

Program	Ref. Processor	Packed Inst. Processor	Increased (%)
Bubble	59763	69061	15.58
Quick	21103	22373	6.02
Hanoi	52739	63040	19.52
Matmul	72260	91220	20.78

Table 2. Comparing the code size (in byte).

Program	Ref. Processor	Packed Inst. Processor	Reduced (%)
Bubble	316	224	29.11
Quick	540	344	36.30
Hanoi	424	324	23.58
MatMul	784	556	29.08

Table 3. Comparing the number of instruction fetch.

Program	Ref. Processor	Packed Inst. Processor	Reduced (%)
Bubble	12549	8414	32.95
Quick	4542	3134	31.00
Hanoi	11005	8121	26.21
MatMul	15612	8563	45.15

7. REFERENCES

[1] http://www.cp.eng.chula.ac.th/~piak/ teaching/embed/chip/sx-chip.htm

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

[3] S. Hines, G. Tyson, and D. Whalley, "Reducing Instruction Fetch Cost by Packing Instructions into Register Windows", Publication Journal, Computer Science Department, Florida State University, 2005.

[4] S. Hines, J. Green, G. Tyson and D. Whalley, "Improving Program Efficiency by Packing Instructions into Registers", Publication Journal, Computer Science Department, Florida State University, 2005

[5] J. Sharkey, D. Ponomarev, K. Ghose and O. Ergin, "Instruction Packing: Reducing Power and Delay of the Dynamic Scheduling Logic", Department of Computer Science, State University of New York, 2005.

[6] D. Yuyuan, "Design of a 16-bit real time stack processor in FPGA", No.127, Northeastern University, Shenyang 110004, P.R.China, 2005.

[7] P. Nanthanavoot, A. Burutarchanai, and P. Chongstitvatana, "Instruction Packing for a 32-bit Resource Efficiency Processor", National Science and Technology Development Agency (NSTDA) Annual Conference, Thailand, 27-30 March 2005 (in Thai).

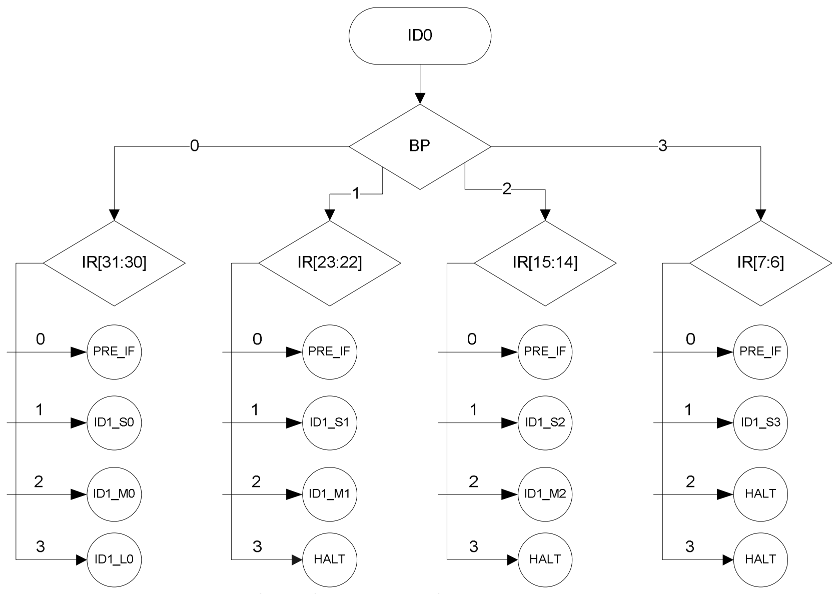

Figure 5.The flow of decode state for packed-instructions

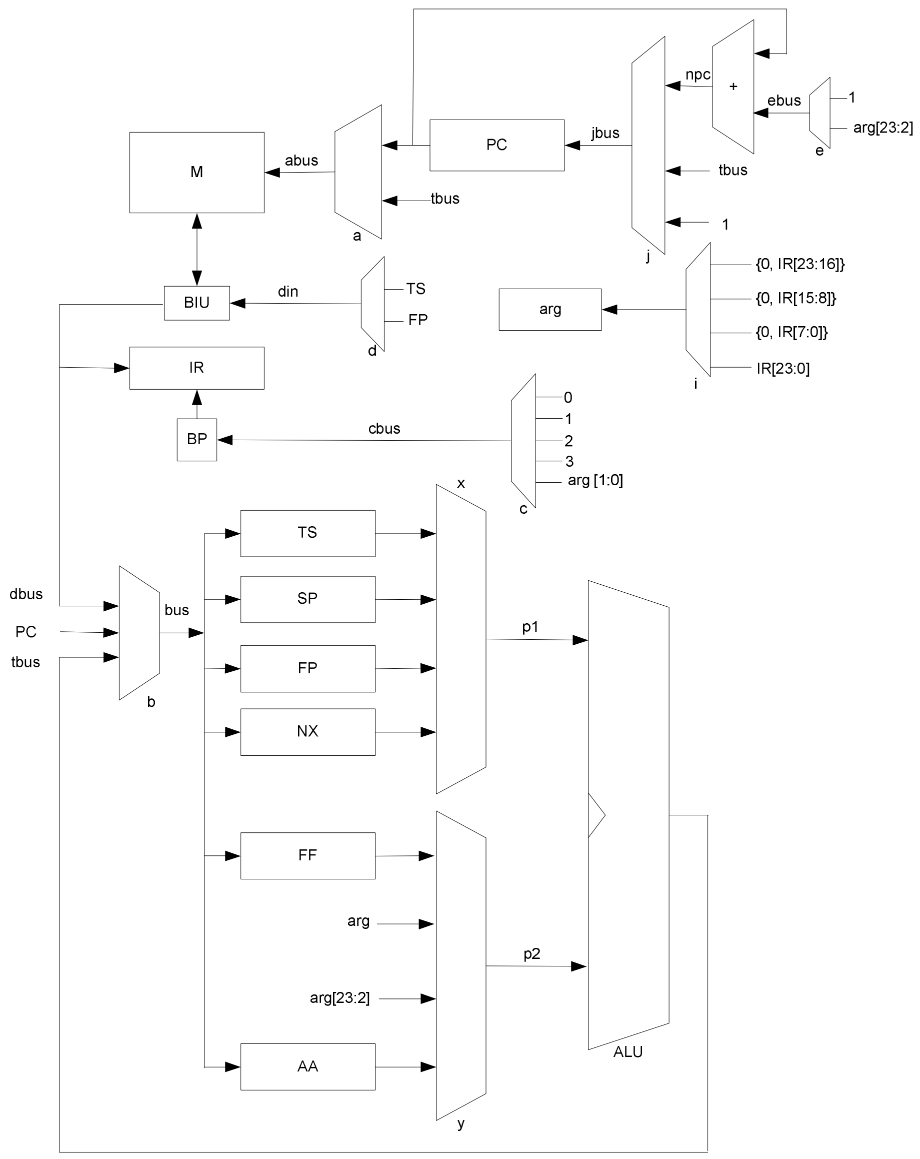

Figure 6. The complete data path