W05-Arch-Practical-Report

CS2002 170011474 February 25th

Overview

This practical requires students to accomplish three parts: commenting function itos_recur, print out all values in the current stack frame of print_recur(), analyse the printed result, and analyse the optimization for given assembly codes.

Implementation & Design (Part 2)

A while loop is used. During every iteration, the offset will be decremented by 8 (bytes). The most critical part is to disassemble each 8 bytes into two 4 bytes and display them separately. Two doublewords are printed firstly and the quadword comes last.

Reflection of Part 1

1. Discussion about idiv and cqto:
   a) idiv does a 128/64 bit division, so the value of *%rdx*. In this case, *%rax* holds lower 64 bits from the dividend <u>long val</u> (which is just long val itself), and *%rdx* holds the upper 64 bits. However, *%rdx* must not contain a random value, so the current zero extension is wrong. It needs the sign extension.
   b) cqto (convert quadword to octoword) does a sign extension for *%rdx*: sign extend *%rax* to (*%rdx*: *%rax)*. In this way, the division operation is correct and safe.
2. Conversion between different size types (typically <u>*movslq*</u>):

   movslq means doing a sign extension when copying 32-bit value into a 64-bit register. This specifies the extension style and avoids inconvenience and unsafety for signed values.

Analysis of Part 2 (stack frame)

1. Description of values found in the stack frames.

```
==============================================================
@ 140722639540052 | offset: 4 | (doubleword) value is: 32764
@ 140722639540048 | offset: 0 | (doubleword) value is: -1963913312
@ 140722639540048 | offset: 0 | (quadword) value is: 140722639540128
-------------------------
@ 140722639540044 | offset: -4 | (doubleword) value is: -1
@ 140722639540040 | offset: -8 | (doubleword) value is: -1
@ 140722639540040 | offset: -8 | (quadword) value is: -1
-------------------------
@ 140722639540036 | offset: -12 | (doubleword) value is: 2
@ 140722639540032 | offset: -16 | (doubleword) value is: 0
@ 140722639540032 | offset: -16 | (quadword) value is: 8589934592
-------------------------
@ 140722639540028 | offset: -20 | (doubleword) value is: 32764
@ 140722639540024 | offset: -24 | (doubleword) value is: -1963913208
@ 140722639540024 | offset: -24 | (quadword) value is: 140722639540232
-------------------------
@ 140722639540020 | offset: -28 | (doubleword) value is: 0
@ 140722639540016 | offset: -32 | (doubleword) value is: 0
@ 140722639540016 | offset: -32 | (quadword) value is: 0
-------------------------
@ 140722639540012 | offset: -36 | (doubleword) value is: 0
@ 140722639540008 | offset: -40 | (doubleword) value is: 1
@ 140722639540008 | offset: -40 | (quadword) value is: 1
-------------------------
@ 140722639540004 | offset: -44 | (doubleword) value is: 0
@ 140722639540000 | offset: -48 | (doubleword) value is: 8229472
@ 140722639540000 | offset: -48 | (quadword) value is: 8229472
-------------------------
@ 140722639539996 | offset: -52 | (doubleword) value is: 5
@ 140722639539992 | offset: -56 | (doubleword) value is: -350628112
@ 140722639539992 | offset: -56 | (quadword) value is: 25419175664
-------------------------
@ 140722639539988 | offset: -60 | (doubleword) value is: 32764
@ 140722639539984 | offset: -64 | (doubleword) value is: -1963913088
@ 140722639539984 | offset: -64 | (quadword) value is: 140722639540352
-------------------------
@ 140722639539980 | offset: -68 | (doubleword) value is: 0
@ 140722639539976 | offset: -72 | (doubleword) value is: 4199168
@ 140722639539976 | offset: -72 | (quadword) value is: 4199168
-------------------------
==============================================================
```

offset	value	Purpose
8	return address (8 bytes)	return to the code segment to the next instruction in the calling method.
0	the address of the previous %rbp (8 bytes)	the control flow will jump back to the previous function after terminating.
-8	long val (8 bytes)	the first argument
-12	int depth (4 bytes)	the second argument
-16	0	16-byte alignment
-24	char**end (8 bytes)	the third argument; the address of a pointer to a char array.
-32	long quotient (8 bytes)	
-40	long mod (8 bytes)	
-48	return value in %rax (8 bytes)	
-52	int len (4 bytes)	length of the string
-56, -64	0	16-byte alignment
-72	return address (8 bytes) as above	

2. **Tests**: Results are correct and correspond to my analysis above. Human readers can access values from the “dump” by those straightforward doublewords and quadwords.

3. Utility of the stack frames and explanation

   A stack frame under this situation consists of **64 bytes (**or **80** regardless of CFA), since ***subq $64, %rsp*** indicates 64 bytes reserved. Between two frames there are **16** bytes holding *return address* and the pushed *previous %rbp*. This convention is defined by *CFA (Canonical Frame Address)*: it is the value of the stack pointer at the call site in the previous frame.

   There are 16 bytes not used, 4 bytes following *int depth* and 12 bytes following %rax. **The reason is that *CFA convention* has set the offset 16 for alignment**. Every block of 16 bytes in stack frame is seen as a unit to store values, starting from *%rbp*. For example, *long val* occupies the first 8 bytes, *int depth* occupies the next 4 bytes. Since there is only 4 bytes left in the current 16-byte unit, which means the following *char **end* cannot be aligned in the same unit. For maximum loading efficiency, *char **end* is stored in the beginning of the next unit (-17 to -32), **leaving those 4 bytes empty 0**.

4. x86 adopts **little-endian** to store values. If we focus on the offset –8 with value 123 stored, from –1 to –4 there is a 0, and from –5 to –8 there is a 123. Thus, x86 stores less significant values in less significant bits.
5. The effect brought by added function print_stack() on format of stack frames

   This is no dramatic change in the layout of stack frames. Although recursively calling print_recur() pushes new frames onto the stack, positions of values stored in the current frame are not changed.

Analysis of Part 3 (Division by invariant integers using multiplication)

1. Error Analysis of Formula (2) and Formula (3):

   The formula (2) does not work for division. For $n = 2^{63} - 1$, $2^{65} \equiv 2 \pmod{10}$, hence $\lceil 2^{65}/10 \rceil - 2^{65}/10 = 8/10$. Moreover, $(n/10 - \lfloor n/10 \rfloor)$ can be up to 9/10. The upper bound of the error is $8/10 * (2^{63} - 1)/2^{65} + 9/10 \approx 1.1 \geq 1$. This shows formula (2) has the same problem as formula (1).

   The formula (3) gives us desired result. $2^{66} \equiv 4 \pmod{10}$, hence $\lceil 2^{66}/10 \rceil - 2^{66}/10 = 6/10$. The upper bound of the error is $6/10 * (2^{63} - 1)/2^{66} + 9/10 \approx 0.975 < 1$. Since the error is less than 1, it can be safely ignored.

2. Implementation of 2^{66} in the assembly file (5 lines) for non-negative n:
   a) movq %rdi, %r15
   b) movabsq $7378697629483820647, %rcx
   c) movq %r15, %rax
   d) imulq %rcx
   e) sarq $2, %rdx

   The argument long val was copied into *%rax* after (a), (c). We can notice that 7378697629483820647 is equal to $2^{66}/10 = \lceil 2^{66}/10 \rceil$ in the formula. (b) is to copy this number into *%rcx*. In (d), the upper 64 bits of the 128-bit product is put into *%rdx* while the lower 64 bits is put into *%rax.* In (e), the upper 64 bits right shift by 2 bits, which indicates now the remaining number in *%rdx* is ($n/2^{66}$). This is what we are looking for.

3. Handling negative n:

   There are 2 fundamental operations: obtain quotients and mods.

   a) For <u>long quot,</u> we can obtain a quot after the multiplication and right shift. However, under signed division and quotient rounded towards 0, if n < 0 while the divisor is positive then the current quotient should be 1 less than the true quotient (Granlund and Montgomery, 1994). To recover the correct value, the quotient is incremented by 1. What has to be mentioned is that this 1 can come from the sign bit of long val. If n ≥ 0, then the quotient remains the same; If n < 0, then the sign bit is 1 which will be added to the quotient.
   b) To obtain an absolute mod, the mod is negated by using negq.

Extension

1. Further optimisations in itos1.s
   a) When a conditional statement comes, if (quot != 0), it uses 19 compared to val + 9 to see if <u>val</u> is greater than 10, which can be used to determine the quotient.
   b) leaq is a very quick way to calculate a result directly stored into the destination register. In itos1.s, leaq is adopted in many arithmetic calculations, which can reduce the compile time.
   c) Before defining the <u>int len,</u> load the sign bit from <u>long val</u> into *%rax*. If <u>long val</u> is negative, then the sign bit is 1, otherwise the sign bit is 0. Afterwards, <u>int len</u> is incremented by 2. This avoids a comparison between <u>val</u> and 0 and reduces the number of registers to be used.
   d) When a comparison involves 0 or setting a register with 0, *testq* and *xorl* can achieve them in a logical bitwise way. This is quick and safe within just 1 instruction.
2. Comparison among MIPS, X86, ARM

Differences	MIPS	ARM	x86
CISC / RISC	RISC	RISC	CISC
Destination registers	Usually the first operand	Usually the first operand	Usually the last operand
Locate the specific value	Offset and $sp	offset and sp, r11(fp)	Offset and rbp

   a) CISC processors tend to use a single instruction with multiple tasks finished. For example, when operating an expression like (x + 4*y), it can use leaq (*%reg1, %reg2*, 4), or when doing a comparison, the flags have also been set. However, RISC processors are likely to implement one task by one instruction.
   b) When a branch instruction appears, MIPS will use "nop" to fill the branch delay slot of jumps or branches in the pipeline. Since if the pipeline is too "deep", a branch will lead to many wasteful instructions following the branch, and they should not be used after branching to another place. "nop" can take up 5 stages

in pipeline and it can reduce the cost in "branching time". X86 so far in itos1.s has no similar method.

c) **Calling conventions:** MIPS usually calls functions with the help of jal and jr $ra explicitly. ARM has similar techniques to *"jump and link"* and *jump back* at the function epilogue. This difference from x86 leads to extra operations to push and pop these registers. Additionally, MIPS and ARM have 2 general registers to hold returning values, while x86 only has *%rax.* Moreover, MIPS contains 2 special registers called *$hi* and *$lo*. They hold values from multiplication and division, while x86 uses *%rdx* and *%rax* but which can also be used to store other values.

Evaluation & Conclusion

My work satisfies requirements and my report has answered all questions raised in the specification. Additionally, in order to understand the architecture of stack frames extensively, I also searched many articles to consolidate my foundation.

In conclusion, I have acquired the deeper insights into the architecture of stack frames and assembly codes of X86. I still need to learn more from MIPS.

Reference list

Granlund, T. and Montgomery, P. (1994). Division by invariant integers using multiplication. *ACM SIGPLAN Notices,* [online] 29(6), pp.61-72. Available at: https://gmplib.org/~tege/divcnst-pldi94.pdf [Accessed 24 Feb. 2019].