KIẾN TRÚC MÁY TÍNH
Khoa Khoa Học và Kỹ Thuật Máy Tính
Đại học Bách Khoa – Tp.HCM
08-2022

Bài tập/Thực hành 4

CHƯƠNG 2 KIẾN TRÚC TẬP LỆNH MIPS: GỌI HÀM (LẬP TRÌNH CẤU TRÚC), THỜI GIAN THỰC THI

Mục tiêu

- Chuyển từ ngôn ngữ cấp cao (C) sang hợp ngữ MIPS.
- Sử dụng lệnh điều khiển (nhảy, rẽ nhánh) để lập trình cấu trúc.
- Biết nguyên lý gọi hàm. Sử dụng các lệnh gọi hàm jal, jr.
- Tính toán thời gian thực thi của chương trình.

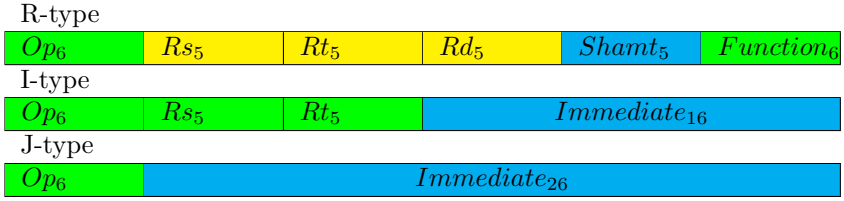
Yêu cầu

- Xem cách dùng các lệnh (set, branch, jump, load, store, **jal, jr**) trong slide và trong file tham khảo.
- Nộp các file code hợp ngữ đặt tên theo format «lab4.asm » (ví dụ **lab4_1a.asm, lab4_1b.asm**) và chứa trong folder **lab4_MSSV**.

Tập lệnh [tham khảo nhanh]

Cú pháp	Ảnh hưởng	Mô tả
slt Rd, Rs, Rt	Rd = (Rs < Rt) ? 1 : 0	[Có dấu]Rd = 1 khi Rs < Rt, ngược lại Rd = 0
sltu Rd, Rs, Rt	Rd = (Rs < Rt) ? 1 : 0	[Không dấu] Rd = 1 khi Rs < Rt, ngược lại Rd = 0
Lệnh nhảy, rẽ nhánh		
beq Rs, Rt, label	if (Rs == Rt) PC ← label	Rẽ nhánh đến label nếu Rs == Rt
bne Rs, Rt, label	if (Rs != Rt) PC ← label	Rẽ nhánh đến label nếu Rs != Rt
bltz Rs, label	if (Rs < 0) PC ← label	Rẽ nhánh đến label nếu Rs < 0
blez Rs, label	if (Rs <= 0) PC ← label	Rẽ nhánh đến label nếu Rs <= 0
bgtz Rs, label	if (Rs > 0) PC ← label	Rẽ nhánh đến label nếu Rs > 0
bgez Rs, label	if (Rs >= 0) PC ← label	Rẽ nhánh đến label nếu Rs >= 0
j label	PC ← label	Nhảy không điều kiện đến label
Gọi hàm		
jr Rs	PC ← Rs	Trở về vị trí thanh ghi Rs trỏ đến
jal label	$ra ← PC+4, PC ← label	Gọi hàm label, khi đó $ra nắm vị trí lệnh tiếp theo
jalr Rs	$ra ← PC+4, PC ← Rs	Gọi hàm Rs đang trỏ đến, khi đó $ra nắm vị trí lệnh tiếp theo

Review: MIPS instruction types



- Op (opcode) Mã lệnh, dùng để xác định lệnh thực thi (đối với kiểu R, Op = 0).

- Rs, Rt, Rd (register): Trường xác định thanh ghi (5-bit). vd: Rs = 4 có nghĩa là Rs đang dùng thanh ghi a0 hay thanh ghi 4.
- Shamt (shift amount): Xác định số bits dịch trong các lệnh dịch bit.
- Function: Xác định toán tử (operator hay còn gọi là lệnh) trong kiểu lệnh R.
- Immediate: Đại diện cho con số trực tiếp, địa chỉ, offset.

Bài tập và Thực hành

Sinh viên chuyển chương trình C bên dưới qua hợp ngữ MIPS tương ứng.

1. Leaf function (hàm lá)
   Chuyển thủ tục "reverse" (đảo thứ tự chuỗi) từ ngôn ngữ C sang hợp ngữ MIPS. Thủ tục reverse được gọi khi thực thi lệnh **`jal reverse`** từ vùng .text. cArray, cArray_size được gắn vào các thanh ghi thanh ghi a_0$, a_1$. Giá trị trả về (nếu có) chứa vào v_0$. Xuất chuỗi ra console.

   ```
   char[] cArray = "Computer Architecture 2022"
   int cArray_size = 26;
   void reverse(char[] cArray, int cArray_size)
   {
      int i;
      char temp;
      for (i =0 ; i <cArray_size/2; i ++)
      {
         temp = cArray[i];
         cArray[i] = cArray[cArray_size -1 -i];
         cArray[cArray_size -1 -i] = temp;
      }
   }
   ```

   Lưu ý: Dùng `"jal reverse"` để gọi thủ tục "reverse" và dùng `"jr $ra"` trở về vị trí thanh ghi $ra đánh dấu.

2. Non-leaf function (là hàm/thủ tục gọi một hàm/thủ tục bên trong).
   Chuyển thủ tục range từ C sang hợp ngữ MIPS tương đương.

   ```
   int iArray[10];
   int iArray_size = 10;
   int range(iArray, iArray_size)
   {
      int temp1 = max(iArray, iArray_size);
      int temp2 = min(iArray, iArray_size);
      int range = temp1 - temp2;
      return range;
   }
   ```

   Chương trình bắt đầu từ vùng .text, sau đó nó gọi hàm range. Trong hàm range lại gọi 2 hàm con là max và min. Giả sử địa chỉ và kích thước iarray được gắn lần lượt vào các thanh ghi a_0$, a_1$. Xuất giá trị range ra ngoài console.
   Lưu ý: Khi gọi các hàm/thủ tục thanh ghi $ra sẽ tự đánh dấu lệnh tiếp theo như là vị trí trở về. Do đó trước khi gọi hàm con trong hàm range thì sinh viên cần lưu lại giá trị thanh ghi $ra trong stack. Sau khi thực thi xong, sinh viên cần phục hồi lại giá trị cho thanh ghi $ra từ stack. Dùng `"jal range"`, `"jal max"`, `"jal min"` để gọi thủ tục range, max, min. Dùng "jr $ra" để trở về vị trí lệnh mà thanh ghi $ra đã đánh dấu.
   Để lưu (push) giá trị r_a$ vào stack, ta có thể dùng các lệnh sau:

   ```
   addi $sp, $sp, -4   # adjust stack for 1 item
   sw   $ra, 0($sp)    # save return address
   ```

   Để phục hồi (pop) r_a$ từ stack, ta có thể dùng các lệnh sau:

   ```
   lw   $ra, 0($sp)    # restore return address
   addi $sp, $sp, 4    # pop 1 item from stack
   ```

3. Cho đoạn code hợp ngữ MIPS bên dưới

```
  addi $a0, $zero, 100    // upper threshold
  addi $a1, $zero, 0      // count variable
  add  $a2, $zero, $zero  // sum initialization
loop:
  beq  $a0, $a1, exit
  add  $a2, $a2, $a1
  addi $a1, $a1, 1
  j loop
exit:
```

(a) Xác định giá trị của thanh ghi $a2 sau khi thực thi đoạn code trên.

(b) Xác định tổng số chu kỳ thực thi khi thực thi đoạn chương trình trên. Giả sử CPI của các lệnh là 1.

(c) Giả sử vùng .text (text segment - vùng để chứa các lệnh thực thi) bắt đầu từ địa chỉ 0x10080000. Xác định mã máy của lệnh `"j loop"` ở dạng HEX.

MIPS32® Instruction Set Quick Reference

R_D	—	Destination register
R_S, R_T	—	Source operand registers
R_A	—	Return address register (R31)
PC	—	Program counter
Acc	—	64-bit accumulator
Lo, Hi	—	Accumulator low ($Acc_{31:0}$) and high ($Acc_{63:32}$) parts
±	—	Signed operand or sign extension
∅	—	Unsigned operand or zero extension
::	—	Concatenation of bit fields
R2	—	MIPS32 Release 2 instruction
DOTTED	—	Assembler pseudo-instruction

Please refer to *"MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set"* for complete instruction set information.

Arithmetic Operations			
ADD	R_D, R_S, R_T	$R_D = R_S + R_T$	(overflow trap)
ADDI	R_D, R_S, const16	$R_D = R_S + const16^{\pm}$	(overflow trap)
ADDIU	R_D, R_S, const16	$R_D = R_S + const16^{\pm}$	
ADDU	R_D, R_S, R_T	$R_D = R_S + R_T$	
CLO	R_D, R_S	R_D = CountLeadingOnes(R_S)	
CLZ	R_D, R_S	R_D = CountLeadingZeros(R_S)	
LA	R_D, label	R_D = Address(label)	
LI	R_D, imm32	R_D = imm32	
LUI	R_D, const16	R_D = const16 << 16	
MOVE	R_D, R_S	$R_D = R_S$	
NEGU	R_D, R_S	$R_D = -R_S$	
SEB[R2]	R_D, R_S	$R_D = R_{S7:0}^{\pm}$	
SEH[R2]	R_D, R_S	$R_D = R_{S15:0}^{\pm}$	
SUB	R_D, R_S, R_T	$R_D = R_S - R_T$	(overflow trap)
SUBU	R_D, R_S, R_T	$R_D = R_S - R_T$	

Shift and Rotate Operations		
ROTR[R2]	R_D, R_S, bits5	$R_D = R_{SBITS5-1:0} :: R_{S31:BITS5}$
ROTRV[R2]	R_D, R_S, R_T	$R_D = R_{SRT4:0-1:0} :: R_{S31:RT4:0}$
SLL	R_D, R_S, shift5	$R_D = R_S$ << shift5
SLLV	R_D, R_S, R_T	$R_D = R_S << R_{T4:0}$
SRA	R_D, R_S, shift5	$R_D = R_S^{\pm} >>$ shift5
SRAV	R_D, R_S, R_T	$R_D = R_S^{\pm} >> R_{T4:0}$
SRL	R_D, R_S, shift5	$R_D = R_S^{\varnothing} >>$ shift5
SRLV	R_D, R_S, R_T	$R_D = R_S^{\varnothing} >> R_{T4:0}$

Logical and Bit-Field Operations		
AND	R_D, R_S, R_T	$R_D = R_S$ & R_T
ANDI	R_D, R_S, const16	$R_D = R_S$ & $const16^{\varnothing}$
EXT[R2]	R_D, R_S, P, S	$R_S = R_{SP+S-1:P}^{\varnothing}$
INS[R2]	R_D, R_S, P, S	$R_{DP+S-1:P} = R_{S_{S-1:0}}$
NOP		No-op
NOR	R_D, R_S, R_T	$R_D = \sim(R_S \mid R_T)$
NOT	R_D, R_S	$R_D = \sim R_S$
OR	R_D, R_S, R_T	$R_D = R_S \mid R_T$
ORI	R_D, R_S, const16	$R_D = R_S \mid const16^{\varnothing}$
WSBH[R2]	R_D, R_S	$R_D = R_{S23:16} :: R_{S31:24} :: R_{S7:0} :: R_{S15:8}$
XOR	R_D, R_S, R_T	$R_D = R_S \oplus R_T$
XORI	R_D, R_S, const16	$R_D = R_S \oplus const16^{\varnothing}$

Condition Testing and Conditional Move Operations		
MOVN	R_D, R_S, R_T	if $R_T \neq 0$, $R_D = R_S$
MOVZ	R_D, R_S, R_T	if $R_T = 0$, $R_D = R_S$
SLT	R_D, R_S, R_T	$R_D = (R_S^{\pm} < R_T^{\pm})$? 1 : 0
SLTI	R_D, R_S, const16	$R_D = (R_S^{\pm} < const16^{\pm})$? 1 : 0
SLTIU	R_D, R_S, const16	$R_D = (R_S^{\varnothing} < const16^{\varnothing})$? 1 : 0
SLTU	R_D, R_S, R_T	$R_D = (R_S^{\varnothing} < R_T^{\varnothing})$? 1 : 0

Multiply and Divide Operations		
DIV	R_S, R_T	$Lo = R_S^{\pm} / R_T^{\pm}$; $Hi = R_S^{\pm}$ mod $R_T^{\pm}$
DIVU	R_S, R_T	$Lo = R_S^{\varnothing} / R_T^{\varnothing}$; $Hi = R_S^{\varnothing}$ mod $R_T^{\varnothing}$
MADD	R_S, R_T	$Acc \mathrel{+}= R_S^{\pm} \times R_T^{\pm}$
MADDU	R_S, R_T	$Acc \mathrel{+}= R_S^{\varnothing} \times R_T^{\varnothing}$
MSUB	R_S, R_T	$Acc \mathrel{-}= R_S^{\pm} \times R_T^{\pm}$
MSUBU	R_S, R_T	$Acc \mathrel{-}= R_S^{\varnothing} \times R_T^{\varnothing}$
MUL	R_D, R_S, R_T	$R_D = R_S^{\pm} \times R_T^{\pm}$
MULT	R_S, R_T	$Acc = R_S^{\pm} \times R_T^{\pm}$
MULTU	R_S, R_T	$Acc = R_S^{\varnothing} \times R_T^{\varnothing}$

Accumulator Access Operations		
MFHI	R_D	$R_D = Hi$
MFLO	R_D	$R_D = Lo$
MTHI	R_S	$Hi = R_S$
MTLO	R_S	$Lo = R_S$

Jumps And Branches (Note: One Delay Slot)		
B	off18	PC += $off18^{\pm}$
BAL	off18	R_A = PC + 8, PC += $off18^{\pm}$
BEQ	R_S, R_T, off18	if $R_S = R_T$, PC += $off18^{\pm}$
BEQZ	R_S, off18	if $R_S = 0$, PC += $off18^{\pm}$
BGEZ	R_S, off18	if $R_S \geq 0$, PC += $off18^{\pm}$
BGEZAL	R_S, off18	R_A = PC + 8; if $R_S \geq 0$, PC += $off18^{\pm}$
BGTZ	R_S, off18	if $R_S > 0$, PC += $off18^{\pm}$
BLEZ	R_S, off18	if $R_S \leq 0$, PC += $off18^{\pm}$
BLTZ	R_S, off18	if $R_S < 0$, PC += $off18^{\pm}$
BLTZAL	R_S, off18	R_A = PC + 8; if $R_S < 0$, PC += $off18^{\pm}$
BNE	R_S, R_T, off18	if $R_S \neq R_T$, PC += $off18^{\pm}$
BNEZ	R_S, off18	if $R_S \neq 0$, PC += $off18^{\pm}$
J	addr28	PC = $PC_{31:28}$ :: $addr28^{\varnothing}$
JAL	addr28	R_A = PC + 8; PC = $PC_{31:28}$ :: $addr28^{\varnothing}$
JALR	R_D, R_S	R_D = PC + 8; PC = R_S
JR	R_S	PC = R_S

Load and Store Operations		
LB	R_D, off16(R_S)	R_D = $mem8(R_S + off16^{\pm})^{\pm}$
LBU	R_D, off16(R_S)	R_D = $mem8(R_S + off16^{\pm})^{\varnothing}$
LH	R_D, off16(R_S)	R_D = $mem16(R_S + off16^{\pm})^{\pm}$
LHU	R_D, off16(R_S)	R_D = $mem16(R_S + off16^{\pm})^{\varnothing}$
LW	R_D, off16(R_S)	R_D = $mem32(R_S + off16^{\pm})$
LWL	R_D, off16(R_S)	R_D = LoadWordLeft(R_S + $off16^{\pm}$)
LWR	R_D, off16(R_S)	R_D = LoadWordRight(R_S + $off16^{\pm}$)
SB	R_S, off16(R_T)	$mem8(R_T + off16^{\pm}) = R_{S7:0}$
SH	R_S, off16(R_T)	$mem16(R_T + off16^{\pm}) = R_{S15:0}$
SW	R_S, off16(R_T)	$mem32(R_T + off16^{\pm}) = R_S$
SWL	R_S, off16(R_T)	StoreWordLeft(R_T + $off16^{\pm}$, R_S)
SWR	R_S, off16(R_T)	StoreWordRight(R_T + $off16^{\pm}$, R_S)
ULW	R_D, off16(R_S)	R_D = unaligned_$mem32(R_S + off16^{\pm})$
USW	R_S, off16(R_T)	unaligned_$mem32(R_T + off16^{\pm}) = R_S$

Atomic Read-Modify-Write Operations		
LL	R_D, off16(R_S)	R_D = $mem32(R_S + off16^{\pm})$; link
SC	R_D, off16(R_S)	if Atomic, $mem32(R_S + off16^{\pm}) = R_D$; R_D = Atomic ? 1 : 0

Registers

0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

Default C Calling Convention (O32)

Stack Management

- The stack grows down.
  - Subtract from $sp to allocate local storage space.
  - Restore $sp by adding the same amount at function exit.
- The stack must be 8-byte aligned.
  - Modify $sp only in multiples of eight.

Function Parameters

- Every parameter smaller than 32 bits is promoted to 32 bits.
- First four parameters are passed in registers $a0–$a3.
  - 64-bit parameters are passed in register pairs:
    - Little-endian mode: $a1:$a0 or $a3:$a2.
    - Big-endian mode: $a0:$a1 or $a2:$a3.
- Every subsequent parameter is passed through the stack.
  - First 16 bytes on the stack are not used.
  - Assuming $sp was not modified at function entry:
    - The 1st stack parameter is located at 16($sp).
    - The 2nd stack parameter is located at 20($sp), etc.
  - 64-bit parameters are 8-byte aligned.

Return Values

- 32-bit and smaller values are returned in register $v0.
- 64-bit values are returned in registers $v0 and $v1:
  - Little-endian mode: $v1:$v0.
  - Big-endian mode: $v0:$v1.

MIPS32 Virtual Address Space

kseg3	0xE000.0000	0xFFFF.FFFF	Mapped	Cached
ksseg	0xC000.0000	0xDFFF.FFFF	Mapped	Cached
kseg1	0xA000.0000	0xBFFF.FFFF	Unmapped	Uncached
kseg0	0x8000.0000	0x9FFF.FFFF	Unmapped	Cached
useg	0x0000.0000	0x7FFF.FFFF	Mapped	Cached

Reading The Cycle Count Register From C

```
unsigned mips_cycle_counter_read()
{
    unsigned cc;
    asm volatile("mfc0 %0, $9" : "=r" (cc));
    return (cc << 1);
}
```

Assembly-Language Function Example

```
# int asm_max(int a, int b)
# {
#   int r = (a < b) ? b : a;
#   return r;
# }

    .text
    .set     nomacro
    .set     noreorder

    .global  asm_max
    .ent     asm_max

asm_max:
    move     $v0, $a0         # r = a
    slt      $t0, $a0, $a1    # a < b ?
    jr       $ra              # return
    movn     $v0, $a1, $t0    # if yes, r = b

    .end     asm_max
```

C / Assembly-Language Function Interface

```
#include <stdio.h>

int asm_max(int a, int b);

int main()
{
    int x = asm_max(10, 100);
    int y = asm_max(200, 20);
    printf("%d %d\n", x, y);
}
```

Invoking MULT And MADD Instructions From C

```
int dp(int a[], int b[], int n)
{
    int i;
    long long acc = (long long) a[0] * b[0];
    for (i = 1; i < n; i++)
        acc += (long long) a[i] * b[i];
    return (acc >> 31);
}
```

Atomic Read-Modify-Write Example

```
atomic_inc:
    ll       $t0, 0($a0)      # load linked
    addiu    $t1, $t0, 1      # increment
    sc       $t1, 0($a0)      # store cond'l
    beqz     $t1, atomic_inc  # loop if failed
    nop
```

Accessing Unaligned Data

note: ULW and USW automatically generate appropriate code

Little-Endian Mode		Big-Endian Mode	
LWR	RD, OFF16(RS)	LWL	RD, OFF16(RS)
LWL	RD, OFF16+3(RS)	LWR	RD, OFF16+3(RS)
SWR	RD, OFF16(RS)	SWL	RD, OFF16(RS)
SWL	RD, OFF16+3(RS)	SWR	RD, OFF16+3(RS)

Accessing Unaligned Data From C

```
typedef struct
{
    int u;
} __attribute__((packed)) unaligned;


int unaligned_load(void *ptr)
{
    unaligned *uptr = (unaligned *)ptr;
    return uptr->u;
}
```

MIPS SDE-GCC Compiler Defines

__mips	MIPS ISA (= 32 for MIPS32)
__mips_isa_rev	MIPS ISA Revision (= 2 for MIPS32 R2)
__mips_dsp	DSP ASE extensions enabled
_MIPSEB	Big-endian target CPU
_MIPSEL	Little-endian target CPU
_MIPS_ARCH_*CPU*	Target CPU specified by -march=*CPU*
_MIPS_TUNE_*CPU*	Pipeline tuning selected by -mtune=*CPU*

Notes

- Many assembler pseudo-instructions and some rarely used machine instructions are omitted.
- The C calling convention is simplified. Additional rules apply when passing complex data structures as function parameters.
- The examples illustrate syntax used by GCC compilers.
- Most MIPS processors increment the cycle counter every other cycle. Please check your processor documentation.