

Matters of efficiency

Lecture B16



Lecture notes section B1 6

2002-02-08

CSE1 303 Part B lecture notes

1

Last time

- Accessing function parameters
- Returning from functions
- Recursion

2002-02-08

CSE1 303 Part B lecture notes

2

In this lecture

- Analysis of translation
- Writing efficient C

2002-02-08

CSE1 303 Part B lecture notes

3

Efficiency

- Often several ways to write the same program
- Want to choose the most efficient implementation
- Space efficiency
 - using small amount of memory
- Time efficiency
 - running in small amount of time
- These goals are sometimes at odds
 - need to make compromise between space and time

2002-02-08

CSE1 303 Part B lecture notes

4

Space efficiency

- To achieve space efficiency in data
 - allocate all variables to be of most appropriate size
 - use char or short instead of int where applicable, especially in arrays
 - free unwanted dynamic data
 - share data where possible
 - use pointers, which can point to same data
 - e.g., don't duplicate string, but point several pointers to same string
 - need to be careful of aliasing bugs
 - use space-efficient algorithms
 - e.g., quicksort (in-place) instead of mergesort (needs temporary array)

2002-02-08

CSE1 303 Part B lecture notes

5

Space efficiency

- To achieve space efficiency in code
 - re-use code
 - write functions wherever possible
 - avoid global variables
 - local variables can be accessed with one machine instruction
 - globals require two machine instructions to construct address of variable
 - MIPS features \$gp (global pointer) register to make some global variables more efficient

2002-02-08

CSE1 303 Part B lecture notes

6

Time efficiency

- To achieve time efficiency
 - avoid using multiply and divide instructions
 - in MIPS, multiply and divide halt processor until result is available, up to 50 clock cycles later
 - use shift (and perhaps add) instructions if multiplying by a constant
 - when writing C, compiler will usually do this for you

2002-02-08

CSEI 303 Part B lecture notes

7

Time efficiency

- To achieve time efficiency
 - make program shorter
 - each instruction takes same time to run, so fewer instructions means shorter execution
 - some space efficiency arguments also improve time efficiency
 - e.g., avoid globals, share data
 - make program take fewer steps
 - economize jump/branch instructions, remove redundant ones

2002-02-08

CSEI 303 Part B lecture notes

8

Loop conditions

- In MIPS, ordered comparisons (`blt`, `ble`, `bgt`, `bge`) are pseudoinstructions
 - expand to more than one machine instruction
- Save one instruction per loop by using unordered comparisons (`beq`, `bne`)
 - for (`i = 0; i < 10; i++`) ⇔ for (`i = 0; i != 10; i++`)
 - can be unsafe to do this if loop is complex

2002-02-08

CSEI 303 Part B lecture notes

9

Loop conditions

- Pre-tested loop (`while`)
- Post-tested loop (`do-while`)

```

# Do it 10 times.
sw $0, x
loop: lw $t0, x
      beq $t0, 10, end
      # body of loop: n steps
      lw $t0, x
      add $t0, $t0, 1
      sw $t0, x
      j loop
end:  rest of program ...
n+6 steps per iteration
    
```

```

# Do it 10 times.
sw $0, x
loop: # body of loop: n steps
      lw $t0, x
      add $t0, $t0, 1
      sw $t0, x
      lw $t0, x
      bne $t0, 10, loop
      # rest of program ...
end:  rest of program ...
n+5 steps per iteration
    
```

2002-02-08

CSEI 303 Part B lecture notes

10

Loop conditions

- Post-tested loops marginally more efficient (space and time) than pre-tested
 - use `do-while` loops if no harm in body running once
- Compilers can turn `while` loop into post-tested equivalent
 - by performing jump to test before loop starts

2002-02-08

CSEI 303 Part B lecture notes

11

Loop unrolling

- Loop as written
- With two iterations written in one loop

```

la $t0, source
la $t1, dest
loop: lbu $t2, 0($t0) # Copy
      sb $t2, 0($t1) # char.
      beq $t2, 0, end # Ended?
      add $t0, $t0, 1 # Next
      add $t1, $t1, 1 # char.
      j loop
end:  # rest of program ...
6 instructions per copy (12
instructions for 2 chars)
    
```

```

la $t0, source
la $t1, dest
loop: lbu $t2, 0($t0) # Copy
      sb $t2, 0($t1) # char.
      beq $t2, 0, end # Ended?
      lbu $t2, 1($t0) # Next
      sb $t2, 1($t1) # char.
      beq $t2, 0, end # Ended?
      add $t0, $t0, 2 # Jump
      add $t1, $t1, 2 # by 2.
      j loop
end:  # rest of program ...
4.5 instructions per copy (9
instructions for 2 chars)
    
```

2002-02-08

CSEI 303 Part B lecture notes

12

Loop unrolling

- Loop unrolling significantly improves time efficiency
 - at significant expense of space efficiency
- Compilers can be directed to unroll loops
 - depending on size of loop body and saved time

2002-02-08

CSE1303 Part B lecture notes

13

Switch statement

```
switch (n)
{
  case 0: /* code for zero */
    break;
  case 1: /* code for one */
    break;
  case 4: /* code for four */
    break;
  case 5: /* code for five */
    break;
  default: /* default code */
    break;
}

lw t0, n
beq $t0, 0, zero
beq $t0, 1, one
beq $t0, 4, four
beq $t0, 5, five
j default
zero: # code for zero
j end
one: # code for one
j end
four: # code for four
j end
five: # code for five
j end
default: # default code
j end
end: # rest of code...
```

switch statement coded as cascaded if-else

one comparison needed for each case

2002-02-08

CSE1303 Part B lecture notes

14

Switch statement

```
switch (n)
{
  case 0: /* code for zero */
    break;
  case 1: /* code for one */
    break;
  case 4: /* code for four */
    break;
  case 5: /* code for five */
    break;
  default: /* default code */
    break;
}

.jtable: .data
.word zero, one, default
.word default, four, five
.text
lw t0, n
blt $t0, 0, default
sll $t0, $t0, 2
lw $t0, .jtable($t0)
jr $t0
zero: # code for zero
j end
one: # code for one
j end
four: # code for four
j end
five: # code for five
j end
default: # default code
j end
end: # rest of code...
```

switch statement coded with jump table (array of addresses of target labels)

5 instructions to handle any case

2002-02-08

CSE1303 Part B lecture notes

15

Switch statement

- When switch statement coded as cascaded if-else
 - put common cases first
 - fewer failed comparisons
 - default case should be uncommon
 - since all comparisons must fail
- When switch statement coded with jump table
 - need cases to be numerically dense
- Compilers can be directed to implement switch statement either way
 - depending on properties of cases

2002-02-08

CSE1303 Part B lecture notes

16

Variables in registers

```
# This version keeps a variable # This version keeps a variable
# on stack. # in a register ($s0).
main: move $fp, $sp          main: move $fp, $sp
      subu $sp, $sp, 4
      li $v0, 5
      syscall
      sw $v0, -4($fp)
loop: lw $t0, -4($fp)
      blt $t0, $zero, end
      # Do something: n steps.
      lw $t0, -4($fp)
      sub $t0, $t0, 1
      sw $t0, -4($fp)
      j loop
end:  li $v0, 10
      syscall

      # Do something: n steps.
      sub $s0, $s0, 1
      j loop
end:  li $v0, 10
      syscall
```

n+6 steps per iteration

n+3 steps per iteration

2002-02-08

CSE1303 Part B lecture notes

17

Variables in registers

- Improved time and space efficiency by storing local variables in registers
 - in MIPS, general-purpose registers \$s0 to \$s7 are available for variables
 - if functions use these, must save old value on stack (along with \$ra and \$sp)
 - not done in CSE1303, because adds more complexity to function calling convention
- Compiler can be directed to put some local variables in registers
 - C has register keyword to provide hint to compiler (ignored by many compilers)

2002-02-08

CSE1303 Part B lecture notes

18

Pointers and arrays

```

/* String copy using arrays. */
char *strcpy(char s[], char t[])
{
    int i = 0;
    while ((s[i] = t[i]) != '\0')
    {
        i++;
    }
    return s;
}

# strcpy function in MIPS.
# variables kept in registers:
# i in $t9, s in $t4, t in $t5.
strcpy: subu $sp, $sp, 8
        sw $ra, 4($sp)
        sw $fp, 0($sp)
        move $t9, $zero # i
        lw $t4, 8($fp) # s
        lw $t5, 12($fp) # t
        add $t0, $t5, $t9
        lbu $t1, 0($t0) # t[i]
        add $t0, $t4, $t9
        sb $t1, 0($t0) # s[i]
        beq $t1, 0, end
        add $t9, $t9, 1 # i++
        j loop
end:    lw $v0, 8($fp) # s
        lw $fp, 0($sp)
        lw $ra, 4($sp)
        addu $sp, $sp, 8
        jr $ra
    
```

7 instructions per copy
(would be 13 per copy if
everything kept on
stack)

2002-02-08

CSEI 303 Part B lecture notes

19

Pointers and arrays

```

/* String copy using pointers. */
char *strcpy(char *s, char *t)
{
    char *r = s;
    while ((*s = *t) != '\0')
    {
        s++; t++;
    }
    return r;
}

# strcpy function in MIPS.
# variables kept in registers:
# r in $t8, s in $t4, t in $t5.
strcpy: subu $sp, $sp, 8
        sw $ra, 4($sp)
        sw $fp, 0($sp)
        move $t8, $sp # r
        move $t8, $t4 # s
        lw $t4, 8($fp) # s
        lw $t5, 12($fp) # t
        lbu $t1, 0($t5) # *t
        sb $t1, 0($t4) # *s
        beq $t1, 0, end
        add $t4, $t4, 1 # s++
        add $t5, $t5, 1 # t++
        j loop
end:    move $v0, $t8 # r
        lw $fp, 0($sp)
        lw $ra, 4($sp)
        addu $sp, $sp, 8
        jr $ra
    
```

6 instructions per copy
(would be 12 per copy if
everything kept on
stack)

2002-02-08

CSEI 303 Part B lecture notes

20

Pointers and arrays

- Pointers often produce more efficient code than arrays
 - especially if arrays are walked along entire length
- Compilers usually not smart enough to generate more efficient code
 - to get benefit, you have to write different C code

2002-02-08

CSEI 303 Part B lecture notes

21

Inline functions

```

# Calling a function ...
main:    .text
        move $fp, $sp
        subu $sp, $sp, 4 # local
        # call func(local, 42)
        subu $sp, $sp, 8
        li $t0, -4($fp) # local
        sw $t0, 0($sp) # arg 1
        li $t0, 42
        sw $t0, 4($sp) # arg 2
        jal func
        addu $sp, $sp, 8
        # more code ...

# A function ...
func:    subu $sp, $sp, 8
        sw $ra, 4($sp)
        sw $fp, 0($sp)
        move $fp, $sp
        # code that uses
        # params at 8($fp)
        # and 12($fp)
        lw $fp, 0($sp)
        lw $ra, 4($sp)
        addu $sp, $sp, 8
        jr $ra
    
```

7 instructions needed
to call function

8 instructions needed to
enter/leave function

2002-02-08

CSEI 303 Part B lecture notes

22

Inline functions

```

# Calling a function ...
main:    .text
        move $fp, $sp
        subu $sp, $sp, 4 # local
        # Pretend to call
        # func(local, 42).
        # Code that uses
        # value at -4($fp)
        # and constant 42.
        # more code ...

# A function ...
func:    subu $sp, $sp, 8
        sw $ra, 4($sp)
        sw $fp, 0($sp)
        move $fp, $sp
        # code that uses
        # params at 8($fp)
        # and 12($fp)
        lw $fp, 0($sp)
        lw $ra, 4($sp)
        addu $sp, $sp, 8
        jr $ra
    
```

saved 15 instructions by
putting body of func
straight into main

no longer even
need this
function's code

2002-02-08

CSEI 303 Part B lecture notes

23

Inline functions

- Inlining functions gives large improvement in time efficiency
 - at vast cost of space efficiency
 - only feasible for very short functions or for functions called very few times
- Compilers can be directed to inline selected functions
 - automatically, or hinted through use of C inline keyword

2002-02-08

CSEI 303 Part B lecture notes

24

Parameter passing

- In C, all parameters are passed by value
 - copies are made of all arguments before function begins
 - function works on these copies
 - function cannot affect caller's environment

2002-02-08

CSEI 303 Part B lecture notes

25

Parameter passing

- Sometimes, want function to modify variable belonging to caller
 - pass address of variable as parameter
 - function treats parameter as pointer to variable
 - function must dereference pointer to access variable
- This is how to implement pass by reference in C (and MIPS) functions

2002-02-08

CSEI 303 Part B lecture notes

26

Passing by reference

```
void remquo(int num, int den,
            int *remPtr, int *quoPtr);

int main()
{
    int a, b, c, d;
    scanf("%d%d", &a, &b);
    remquo(a, b, &c, &d);
    printf("%d %d\n", c, d);
}

void remquo(int num, int den,
            int *remPtr, int *quoPtr)
{
    *quoPtr = num / den;
    *remPtr = num % den;
}
```

\$sp =
0x7FFF3134 → a
b
c
d
\$fp =
0x7FFF3144 →

		0x7FFF311C
		0x7FFF3120
		0x7FFF3124
		0x7FFF3128
		0x7FFF312C
		0x7FFF3130
a	64	0x7FFF3134
b	15	0x7FFF3138
c	???	0x7FFF313C
d	???	0x7FFF3140
		0x7FFF3144

2002-02-08

CSEI 303 Part B lecture notes

27

Passing by reference

these parameters are passed by reference
remPtr points to c
quoPtr points to d

\$sp = fp =
0x7FFF3144 → saved \$fp
saved \$ra
num
den
remPtr
quoPtr
a
b
c
d

		0x7FFF311C
		0x7FFF3120
		0x7FFF3124
		0x7FFF3128
		0x7FFF312C
		0x7FFF3130
		0x7FFF3134
		0x7FFF3138
		0x7FFF313C
		0x7FFF3140
		0x7FFF3144

2002-02-08

CSEI 303 Part B lecture notes

28

Passing by reference

```
main:
    .text
    move $fp, $sp
    subu $sp, $sp, 16
    # Read a and b.
    li $v0, 5
    syscall
    sw $v0, -16($fp) # a
    li $v0, 5
    syscall
    sw $v0, -12($fp) # b
    # Call remquo.
    subu $sp, $sp, 16
    lw $t0, -16($fp) # a
    sw $t0, 0($sp) # arg 1
    lw $t0, -12($fp) # b
    sw $t0, 4($sp) # arg 2
    la $t0, -8($fp) # &c
    sw $t0, 8($sp) # arg 3
    la $t0, -4($fp) # &d
    sw $t0, 12($sp) # arg 4
    jal remquo
    addu $sp, $sp, 16
```

```
# Print out c and d.
li $v0, 1
lw $a0, -8($fp) # c
syscall
li $v0, 11 # print char
li $a0, ' '
syscall
li $v0, 1
lw $a0, -4($fp) # d
syscall
li $v0, 10 # exit
syscall
```

2002-02-08

CSEI 303 Part B lecture notes

29

Passing by reference

```
remquo: subu $sp, $sp, 8
        sw $ra, 4($sp)
        sw $fp, 0($sp)
        subu $sp, $sp, 4
        # num / den
        lw $t0, 8($fp) # num
        lw $t1, 12($fp) # den
        div $t0, $t0, $t1
        # *quoPtr = ...
        lw $t1, 16($fp) # quoPtr
        sw $t0, 0($t1) # *quoPtr
```

```
# num % den
lw $t0, 8($fp) # num
lw $t1, 12($fp) # den
rem $t0, $t0, $t1
# *remPtr = ...
lw $t1, 20($fp) # remPtr
sw $t0, 0($t1) # *remPtr
lw $sp, 0($sp)
lw $ra, 4($sp)
addu $sp, $sp, 8
jr $ra
```

2002-02-08

CSEI 303 Part B lecture notes

30

Passing large parameters

- Arrays always passed by reference
 - including strings (arrays of char)
 - this is why you can write `char x[]` or `char *x` in function parameter lists
- Structures can be passed by value in C, but probably should always be passed by reference instead
 - see next example for why

2002-02-08

CSE1 303 Part B lecture notes

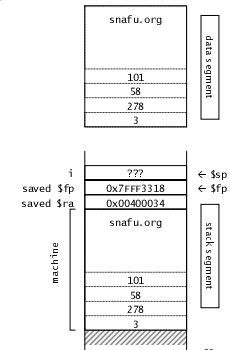
31

Passing by value

```
struct Host {
    char name[20];
    int ip[4];
} mycomputer = {
    "snafu.org",
    { 101, 58, 278, 3 }
};

int printIP(const
    struct Host machine)
{
    int i;
    for (i = 0; i < 4; i++)
        printf("%d ",
            machine.ip[i]);
}

int main()
{
    printIP(mycomputer);
}
```



2002-02-08

CSE1 303 Part B lecture notes

32

Passing by value

```
struct Host {
    char name[20];
    int ip[4];
} mycomputer = {
    "snafu.org",
    { 101, 58, 278, 3 }
};

int printIP(const
    struct Host machine)
{
    int i;
    for (i = 0; i < 4; i++)
        printf("%d ",
            machine.ip[i]);
}

int main()
{
    printIP(mycomputer);
}
```

```
.data
mycomputer:
    .asciiz "snafu.org"
    .space 10
    .word 101, 58, 278, 3
```

2002-02-08

CSE1 303 Part B lecture notes

33

Passing by value

```
.text
printIP:
    subu $sp, $sp, 8
    sw $fp, 4($sp)
    sw $ra, 0($sp)
    move $fp, $sp

    subu $sp, $sp, 4
    sw $zero, -4($fp) # i

loop:   lw $t2, -4($fp) # i
        bge $t2, 4, end

        li $v0, 1
        la $t0, 28($fp) # ip
        lw $t2, -4($fp) # i
        sll $t1, $t2, 2
        add $t0, $t0, $t1
        lw $a0, 0($t0)
        syscall
```

```
li $v0, 11 # print char
li $a0, ' '
syscall

lw $t0, -4($fp) # i
add $t0, $t0, 1
sw $t0, -4($fp) # i
j loop

end:   addu $sp, $sp, 4
        lw $ra, 4($sp)
        lw $fp, 0($sp)
        addu $sp, $sp, 8
        jr $ra
```

pass-by-value printIP
takes 75 instructions to
run

2002-02-08

CSE1 303 Part B lecture notes

34

Passing by value

```
main:   move $fp, $sp

# Copy whole struct!
# 20*1 + 4*4 = 52 bytes
subu $sp, $sp, 32
lw $t0, mycomputer+0
sw $t0, 0($sp)
lw $t0, mycomputer+4
sw $t0, 4($sp)
lw $t0, mycomputer+8
sw $t0, 8($sp)
lw $t0, mycomputer+12
sw $t0, 12($sp)
lw $t0, mycomputer+16
sw $t0, 16($sp)
lw $t0, mycomputer+20
sw $t0, 20($sp)
lw $t0, mycomputer+24
sw $t0, 24($sp)
lw $t0, mycomputer+28
sw $t0, 28($sp)
```

```
jal printIP
addi $sp, $sp, 32

li $v0, 10
syscall
```

pass-by-value main takes
19 instructions to call
printIP

2002-02-08

CSE1 303 Part B lecture notes

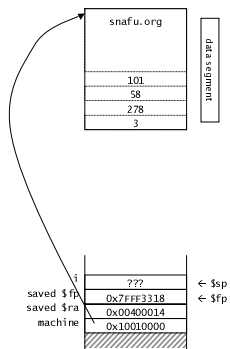
35

Passing by reference

```
struct Host {
    char name[20];
    int ip[4];
} mycomputer = {
    "snafu.org",
    { 101, 58, 278, 3 }
};

int printIP(const
    struct Host *machine)
{
    int i;
    for (i = 0; i < 4; i++)
        printf("%d ",
            machine->ip[i]);
}

int main()
{
    printIP(&mycomputer);
}
```



2002-02-08

CSE1 303 Part B lecture notes

36

Passing by reference

```

struct Host {
    char name[20];
    int ip[4];
} mycomputer = {
    "snafu.org",
    { 101, 58, 278, 3 }
};

int printIP(const
    struct Host *machine)
{
    int i;
    for (i = 0; i < 4; i++)
        printf("%d ",
            machine->ip[i]);
}

int main()
{
    printIP(&mycomputer);
}

```

```

.data
mycomputer:
    .asciiz "snafu.org"
    .space 10
    .word 101, 58, 278, 3

```

2002-02-08

CSE1303 Part B lecture notes

37

Passing by reference

```

.printIP:
    subu $sp, $sp, 8
    sw $fp, 4($sp)
    sw $ra, 0($sp)
    move $fp, $sp

    subu $sp, $sp, 4
    sw $zero, -4($fp) # i

loop:   lw $t2, -4($fp) # i
        bge $t2, 4, end

    li $v0, 1
    lw $t0, 8($fp) # machine
    lw $t2, -4($fp) # i
    sll $t1, $t2, 2
    add $t0, $t0, $t1
    lw $a0, 20($t0) # ip
    syscall

```

```

    li $v0, 11 # print char
    li $a0, ' '
    syscall

    lw $t0, -4($fp) # i
    add $t0, $t0, 1
    sw $t0, -4($fp) # i
    j loop

end:   addu $sp, $sp, 4
        lw $ra, 4($sp)
        lw $fp, 0($sp)
        addu $sp, $sp, 8
        jr $ra

```

pass-by-reference
printIP still takes 75
instructions to run

2002-02-08

CSE1303 Part B lecture notes

38

Passing by reference

```

main:   move $fp, $sp

    # Pass param as address.
    # 1 * 4 = 4 bytes
    subu $sp, $sp, 4
    la $t0, mycomputer
    sw $t0, 0($sp)
    jal printIP
    addi $sp, $sp, 4

    li $v0, 10
    syscall

```

pass-by-reference main
takes 5 instructions to
call printIP instead of
19

2002-02-08

CSE1303 Part B lecture notes

39

Parameter passing

- Pass by reference is more efficient
 - when size of parameter is larger than size of pointer
 - especially obvious for large structures
 - slight penalty in time efficiency in callee
 - usually no more than one instruction per variable reference
 - huge saving in time efficiency in caller
 - because structure does not need to be copied
- Compiler can't handle this automatically
 - need to write different C code

2002-02-08

CSE1303 Part B lecture notes

40

Covered in this lecture

- Time efficiency
- Space efficiency
- Efficiency with
 - loops
 - switch statements
 - variables in registers
 - pointers and arrays
 - inlining functions
 - passing by reference

2002-02-08

CSE1303 Part B lecture notes

41

Going further

- Compilers and optimization
 - gcc manual page (man gcc)

2002-02-08

CSE1303 Part B lecture notes

42

Next time

- Revision

2002-02-08

CSEI 303 Part B lecture notes

43

Copyright

Copyright © 2001 Deborah Pickett.
No part of this presentation may be
duplicated without permission from
the author.

2002-02-08

CSEI 303 Part B lecture notes

44