Optimization

CPU time = IC x CPI x CC

- A chained if-then-else versus switch/case (use of Jumptable)
- Impact of procedure calls
  - Tail recursion
  - Inline
  - Hardware vs software
- Pipelining increases throughput (and equivalently reduces CPI or clock cycle)
  - Software resolution: Rescheduling
  - Register allocation
- Loops
  - unrolling: reduce loop overhead, enable rescheduling
  - Reduce inefficiency
  - re-ordering (take advantage of locality)

Handling multiple cases

use a series of chained conditional branches: if-then-else

if x = 1 then …
else if x = 2 then …
else if x = 3 then …
….

Convert to assembly code

```
        @ assume x is stored in r0
        mov     r1, #1          @ r1 = 1
        cmp     r0, r1          @ r0
        bne     Else1
        …                       @ code for when x = 1
Else1:  mov     r1, #2
        cmp     r0, r1
        bne     Else2
        …                       @ code for when x = 2
Else2:  mov     r1, #3
        cmp     r0, r1
        bne     Else3
        …
```

Drawbacks: Given x, its value has to be compared multiple times until a match is found, this can be costly when the chain is long and the match happens to be located towards the end of the chain.

use switch/case

```
int switchexample (int x) {
   int y;
   switch(x) {
      case 1: y = x; break;
      case 2: y = 2 * x;              /* fall through */
      case 3: y = 3 * x; break;
                                      /* missing case 4 */
      case 5: y = 5 * x; break;
      default: y = x; break;
   }
   return y;
}
```

Implementations in assembly using a jumptable (see next slide)

Recall that this jumptable technique is also used in object oriented programming for handling methods for objects (Vtable).

```
Switchexample:
        ldr     r2, =Jtable       @ address of Jtable is in r2
        mov     r1, #0
        cmp     r0, r1            @ check if x (which is in r0) is smaller than 0
        blt     Default
        mov     r1, #5
        cmp     r0, r1            @ check if x is greater than 5
        bgt     Default
        add     r2, r2, r0, LSL #2  @ x multiply by 4 is used as the offset to r2
                                  @  r2 points to the correct element in JTable
Default:
        ldr     r2, [r2]          @ load corresponding Jtable element, which
                                              @ itself is an address of the case label.
        mov     pc, r2
Case1:  mov     r7, r0
        b       done
Case2:  mov     r7, r0, lsl #1    @ fall through to case 3
Case3:  mov     r3, #3
        mul     r7, r3, r0
        b       done
Case4:  b       Case0             @ undefined case is treated as default
Case5:  mov     r3, #5
        mul     r7, r3, r0
        b       done
Case0:  mov     r7, r0            @default case
        b       done
done:   mov     pc, lr

.data
JTable: .word Case0, Case1, Case2, Case3, Case4, Case5
```

For a chained **if-then-else** construct to find a case out of N cases, what is the time complexity?

A. O(1)
B. O(log N)
C. O(N)
D. O(N^2)

For a **switch/case** construct to find a case out of N cases, what is the time complexity?

A. O(1)
B. O(log N)
C. O(N)
D. $O(N^2)$

Drawbacks: When the range is large and cases are sparse, the jumptable becomes not space economical. A chained if-then-else is a better option for such situations.

e.g.,

```
int switchexample (int x) {
  int y;
  switch(x) {
    case 1: y = x; break;
    case 20: y = 20 * x;              /* fall through */
    case 330: y = 330 * x; break;
    case 5000: y = 5000 * x; break;
    default: y = x; break;
  }
  return y;
}
```

Note: the switch/case is introduced as a distinct construct in the C language so that a compiler can “know” a jumptable implementation is preferred to a chained if-then-else implementation.

Procedure calls

Optimization to reduce the overhead (maintaining the stack)
-Inlining (cut & paste)
  Examples:

```
inline int max (int a, int b)
{
  if (a > b)
    return a;
  else
    return b;
}

a = max (x, y);

/*
  This is now equivalent to

  if (x > y)
    a = x;
  else
    a = y;
*/
```

pros: reduce run-time cost; allow for more transformations conducive to optimization.

cons: increase code size, thus less desirable in embedded systems where memory is limited.

Procedure calls

Optimization to reduce the overhead (maintaining the stack)

-Tail recursion:

- when no further computation follows a recursive call;
- can be easily converted to iteration (automatically by the compiler);
- no stack overhead is needed any more.

```
gcd(a, b) {
   while (a!=b) {
      if(a>b) a = a – b;
      else  b = b – a;
   }
   return a;
}
-------
gcd:

While:      cmp     r0, r1
            beq     Return

If:         ble     Else
            sub     r0, r0, r1
            b       While
Else:       sub     r1, r1, r0
            b       While
Return:
            mov     pc, lr
```

```
gcd(a, b) {
   if(a==b)          return a;
   else if (a>b)   return gcd(a-b, b);
   else              return gcd(a, b-a);
}
-----------
gcd:
            sub     sp, sp, 4          @ push
            str     lr, [sp, #0]       @push
            cmp     r0, r1
            beq     Return

If:         ble     Else
            sub     r0, r0, r1
            b       Rec
Else:       sub     r1, r1, r0
Rec:        bl      gcd                @ recursive call

@ Note that here is the return point after the recursive call,
@ but there is no more instructions to execute here other than fall
@ though to the end of the whole procedure !
@ Therefore, no need to save this return addr onto the stack.
@ So, remove the code for push and pop, and change bl to b.

Return:     ldr     lr, [sp, #0]       @ pop
            add     sp, sp, #4         @ pop

            mov     pc, lr
```

```
gcd(a, b) {
  while (a!=b) {
    if(a>b) a = a – b;
    else  b = b – a;
  }
  return a;
}
-------
gcd:


While:   cmp     r0, r1
         beq     Return

If:      ble     Else
         sub     r0, r0, r1
         b       While
Else:    sub     r1, r1, r0
         b       While
Return:
         mov     pc, lr
```

```
gcd(a, b) {
  if(a==b)       return a;
  else if (a>b)  return gcd(a-b, b);
  else           return gcd(a, b-a);
}
-----------
gcd:
         sub     sp, sp, 4              @ push
         str     lr, [sp, #0]           @push
         cmp     r0, r1
         beq     Return

If:      ble     Else
         sub     r0, r0, r1
         b       Rec
Else:    sub     r1, r1, r0
Rec:     b       gcd                    @iterative loop

@ Note that here is the return point after the recursive call,
@ but there is no more instructions to execute here other than fall
@ though to the end of the whole procedure !
@ Therefore, no need to save this return addr onto the stack.
@ So, remove the code for push and pop, and change bl to b.

Return:  ldr     lr, [sp, #0]           @ pop
         add     sp, sp, #4             @ pop

         mov     pc, lr
```

fact(n)
  if n = 1 return 1
  else return n * fact(n-1)

"tailize" →

fact1(n, p)
  if n = 1 return p
  else return fact1(n-1, n*p)

```
@assume n is in r0
fact:
        sub     sp, sp, #8
        str     lr, [sp, #4]
        str     r0, [sp, #0]
        mov     r2, #1
        cmp     r0, r2
        bne     Else
        mov     r7, #1
        b       Return

Else:   sub     r0, r0, #1   @ n = (n-1)
        bl      fact

        ldr     r0, [sp, #0]
        mul     r7, r0, r7

Return:
        ldr     lr, [sp, #4]
        add     sp, sp, #8
        mov     pc, lr
```

```
@ assume n is in r0, and p is in r1
fact1:

        mov     r2, #1
        cmp     r0, r2
        bne     Else
        mov     r7, r1
        b       Return

Else:   mul     r1, r1, r0
        sub     r0, r0, #1
        b       fact1

Return:
        mov     pc, lr
```

The difference can be observed via the semantics at the high level programming language

```
  fact(6)
= 6* (fact(5))          /* multiplication is held off because fact(5) is not known */
= 6* (5 * (fact(4)))
= 6* (5 * (4* (fact(3))))
= 6* (5 * (4* (3 * (fact(2)))))
= 6* (5 * (4* (3 * (2 * (fact(1))))))
= 6* (5 * (4* (3 * (2 * 1)))))
= 6* (5 * (4* (3 * 2 ))))
= 6* (5 * (4* 6)))
= 6* (5 * 24))
= 6* 120
= 720
```

→ Correspond to the growth and shrink of the stack in memory

```
  fact1(6, 1)
= fact1(5, 6)
= fact1(4, 30)
= fact1(3, 120)
= fact1(2, 360)
= fact1(1, 720)
```

```
fact1(n, p)
  if n = 1  return p
  else return fact1(n-1, n*p)
```

Reduce loop inefficiency

<u>Loop invariant</u>: instructions whose result does not change from iteration to iteration, and therefore can be moved outside the loop without affecting the semantics of the program.

For example,

```
for(i = 0; i < n; i++) {
        z = x + y;
        a[i] = 4*i + z*z;
}
```

Loop-invariant: z = x + y, and z*z

```
z = x + y;
w = z * z;
for(i = 0; i < n; i++) {
        a[i] = 4*i + w;
}
```

Strength reduction: a costly operation is replaced with an equivalent by less expensive operation

For example, multiplication and division by a power of 2 can be achieved with shifting

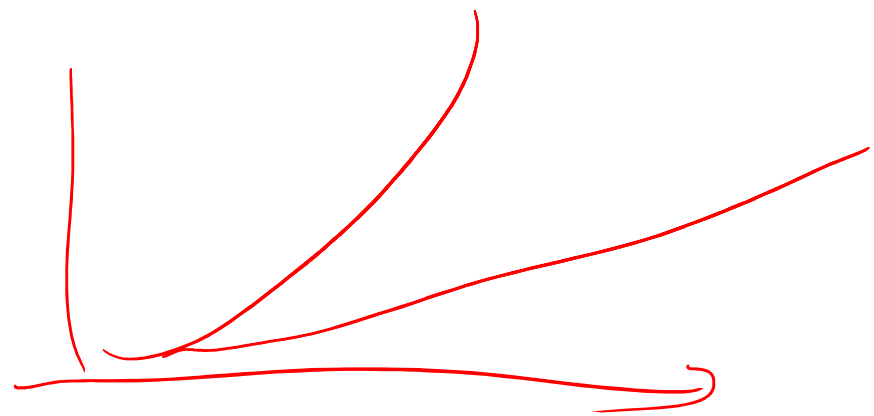
4*i by i<<2

Reduce loop inefficiency

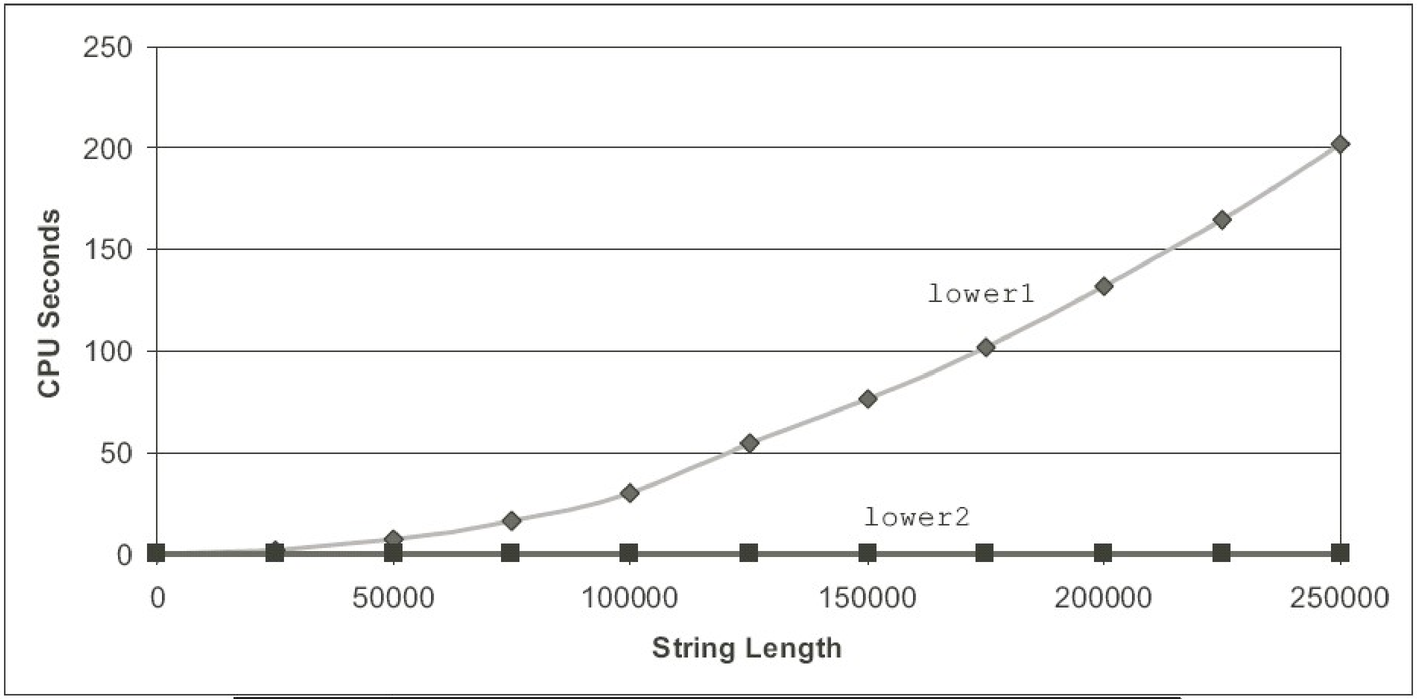
```c
/* Convert string to lower case: slow */
void lower1(char *s)
{
    int i;

    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

/* Convert string to lower case: faster */
void lower2(char *s)
{
    int i;
    int len = strlen(s);

    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

/* Implementation of library function strlen */
/* Compute length of string */
size_t strlen(const char *s)
{
    int length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

Is there any loop-invariant?

A) Yes
B) No



Credit: Bryant & O'Hallaron, CSAPP



Function	String Length					
	8,192	16,384	32,768	65,536	131,072	262,144
lower1	0.15	0.62	3.19	12.75	51.01	186.71
lower2	0.0002	0.0004	0.0008	0.0016	0.0031	0.0060

Credit: Bryant & O'Hallaron, CSAPP

Unrolling loops & rescheduling

Example: function that computes the dot product of two n-element int arrays.

```
dp = 0;
for(i=1; i<n; i++) {
        dp += A[i] x B[i];
}
```

```
        mov     r3, r0   @ r0 = &A
        mov     r4, r1   @ r1 = &B
        mov     r5, r2   @ r2 = n
        mov     r9, #0
        b  L2
L1:     ldr     r6,[ r3, #0]
        ldr     r7, [r4, #0]
        mul     r8, r6, r7
        add     r9, r9, r8
        add     r3, r3, #4
        add     r4, r4, #4
        sub     r5, r5, #1
        mov     r10, #0
        cmp     r5, r10
L2:     bne     L1
```

stalled

Let's assume the pipelined CPU has one cycle delay for load-and-use dependency, and 3 cycle delay for read-before-write for mul. Without rescheduling, this program needs to have delayed 4 cycles per iteration. E.g., if n=200, idled cycles = 4 x 200 = 800.

If we reschedule as follows, we can remove one delayed cycle after ldr r7, two delayed cycles after mul, and leave only one delayed cycle after mul.

original

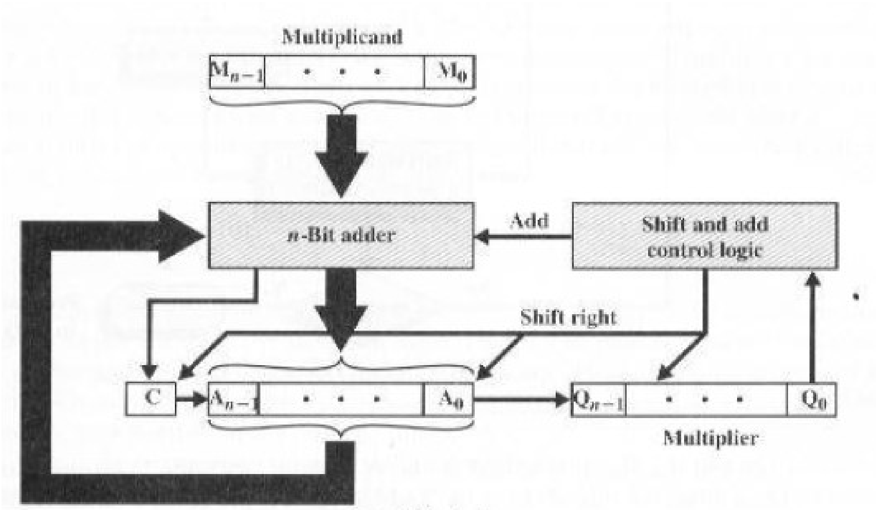
```
-------------------------------------------------------
1       mov     r3, r0   @ r0 = &A
2       mov     r4, r1   @ r1 = &B
3       mov     r5, r2   @ r2 = n
3       mov     r9, #0
5       b  L2
L1:     ldr     r6,[ r3, #0]
7       ldr     r7, [r4, #0]
8       mul     r8, r6, r7
9       add     r9, r9, r8
10      add     r3, r3, #4
11      add     r4, r4, #4
12      sub     r5, r5, #1
13      mov     r10, #0
14      cmp     r5, r10
L2:     bne     L1
```

After rescheduling

```
-------------------------------------------------------
1       mov     r3, r0   @ r0 = r&A
2       mov     r4, r1   @ r1 = &B
3       mov     r5, r2   @ r2 = n
3       mov     r9, #0
5       b  L2
L1:     ldr     r6,[ r3, #0]
7       ldr     r7, [r4, #0]
8       sub     r5, r5, #1
9       mul     r9, r6, r7
10      add     r3, r3, #4
11      add     r4, r4, #4
12      add     r9, r9, r8
13      mov     r10, #0
14      cmp     r5, r10
L2:     bne     L1
```

Hardware (e.g., multiplication)
mul r1, r2, r3

The function can be natively supported by an instruction at the machine level with hardware implementation as shown below. While this hardware solution seems to be the best -- having neither overhead run-time cost like for subroutine nor increased code size of inline function, this leads to more expensive hardware and therefore should only be resorted to for optimizing mostly common functions such as multiplication.



Another example of hardware solution to optimization.

The instruction

```
            …
            bl      sub1
            …
```

can be simulated with a couple ARM instructions as follows.

```
            …
            ldr     lr, =Return_here
            b       sub1
Return_here: …
```

Design philosophy of RISC v.s. CISC:
CISC favors hardware solution, i.e, more instructions natively supported by hardware, whereas RISC favors fewer instructions and provides pseudoinstructions supported by the assembler to ease the programming.

Reordering nested loops to increase cache hits (via locality)

```
for i := 1 to n
    for j := 1 to n
        A[i, j] := 0
```

If A is laid out in row-major order, and if each cache line contains m elements of A, then this code will suffer n^2/m cache misses. On the other hand, if A is laid out in column-major order, and if the cache is too small to hold n lines of A, then the code will suffer n^2 misses, fetching the entire array from memory m times. The difference can have an enormous impact on performance. A loop-reordering compiler can improve this code by *interchanging* the nested loops:

```
for j := 1 to n
    for i := 1 to n
        A[i, j] := 0
```

In more complicated examples, interchanging loops may improve locality of reference in one array, but worsen it in others. Consider this code to transpose a two-dimensional matrix:

```
for j := 1 to n
    for i := 1 to n
        A[i, j] := B[j, i]
```