

Optimizing ARM Assembly

Computer Organization and Assembly Languages

Yung-Yu Chuang

2007/12/1

with slides by Peng-Sheng Chen

Optimization



- Compilers do perform optimization, but they have blind sites. There are some optimization tools that you can't explicitly use by writing C, for example.
 - Instruction scheduling
 - Register allocation
 - Conditional execution

You have to use hand-written assembly to optimize critical routines.

- Use ARM9TDMI as the example, but the rules apply to all ARM cores.
- Note that the codes are sometimes in *armasm* format, not *gas*.

ARM optimization

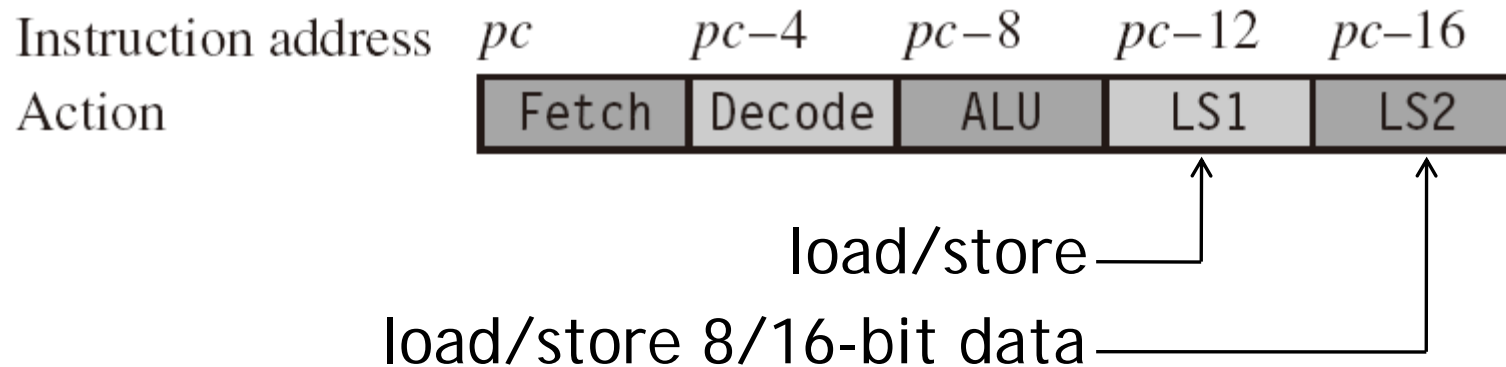


- Utilize ARM ISA's features
 - Conditional execution
 - Multiple register load/store
 - Scaled register operand
 - Addressing modes

Instruction scheduling

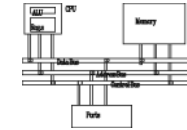


- ARM9 pipeline



- Hazard/Interlock: If the required data is the unavailable result from the previous instruction, then the process stalls.

Instruction scheduling



- No hazard, 2 cycles

```
ADD r0, r0, r1
ADD r0, r0, r2
```

- One-cycle interlock

```
LDR r1, [r2, #4]
ADD r0, r0, r1
```

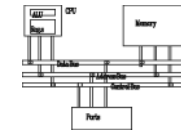
Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	ADD	LDR	...
Cycle 3		...	ADD	—	LDR

stall

bubble



Instruction scheduling

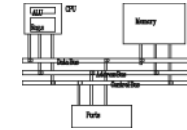


- On-cycle interlock, 4 cycles

```
LDRB  r1, [r2, #1]
ADD   r0, r0, r2    ; no effect on performance
EOR   r0, r0, r1
```

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	EOR	ADD	LDRB	...	
Cycle 2	...	EOR	ADD	LDRB	...
Cycle 3		...	<i>EOR</i>	ADD	LDRB
Cycle 4		...	EOR	—	ADD

Instruction scheduling



- Branch takes 3 cycles due to stalls

```

MOV    r1, #1
B      case1
AND    r0, r0, r1
EOR    r2, r2, r3
...
case1
SUB    r0, r0, r1
    
```

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	AND	B	MOV	...	
Cycle 2	EOR	AND	B	MOV	...
Cycle 3	SUB	—	—	B	MOV
Cycle 4	...	SUB	—	—	B
Cycle 5		...	SUB	—	—

Scheduling of load instructions



- Load occurs frequently in the compiled code, taking approximately 1/3 of all instructions. Careful scheduling of loads can avoid stalls.

```
void str_tolower(char *out, char *in)
{
    unsigned int c;

    do
    {
        c = *(in++);
        if (c>='A' && c<='Z')
        {
            c = c + ('a' - 'A');
        }
        *(out++) = (char)c;
    } while (c);
}
```


Scheduling of load instructions



```
str_tolower
```

```
LDRB    r2, [r1], #1    ; c = *(in++)
SUB     r3, r2, #0x41   ; r3 = c - 'A'
CMP     r3, #0x19      ; if (c <= 'Z' - 'A')
ADDLS   r2, r2, #0x20   ;   c += 'a' - 'A'
STRB    r2, [r0], #1   ; *(out++) = (char)c
CMP     r2, #0          ; if (c != 0)
BNE     str_tolower    ;   goto str_tolower
MOV     pc, r14         ; return
```

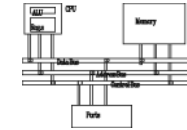
2-cycle stall. Total 11 cycles for a character.
It can be avoided by preloading and unrolling.
The key is to do some work when awaiting data.

Load scheduling by preloading



- Preloading: loads the data required for the loop at the end of the previous loop, rather than at the beginning of the current loop.
- Since loop i is loading data for loop $i+1$, there is always a problem with the first and last loops. For the first loop, insert an extra load outside the loop. For the last loop, be careful not to read any data. This can be effectively done by conditional execution.

Load scheduling by preloading



```
out    RN 0    ; pointer to output string
in     RN 1    ; pointer to input string
c      RN 2    ; character loaded
t      RN 3    ; scratch register

; void str_tolower_preload(char *out, char *in)
str_tolower_preload
LDRB   c, [in], #1      ; c = *(in++)

loop
SUB    t, c, #'A'      ; t = c-'A'
CMP    t, #'Z'-'A'     ; if (t <= 'Z'-'A')
ADDLS  c, c, #'a'-'A'  ; c += 'a'-'A';
STRB   c, [out], #1    ; *(out++) = (char)c;
TEQ    c, #0           ; test if c==0
LDRNEB c, [in], #1    ; if (c!=0) { c=*in++;
BNE    loop           ;          goto loop; }
MOV    pc, lr         ; return
```

9 cycles.

11/9~1.22

Load scheduling by unrolling



- Unroll and interleave the body of the loop. For example, we can perform three loops together. When the result of an operation from loop i is not ready, we can perform an operation from loop $i+1$ that avoids waiting for the loop i result.

Load scheduling by unrolling



```
out    RN 0    ; pointer to output string
in     RN 1    ; pointer to input string
ca0    RN 2    ; character 0
t      RN 3    ; scratch register
ca1    RN 12   ; character 1
ca2    RN 14   ; character 2
      ; void str_tolower_unrolled(char *out, char *in)
str_tolower_unrolled
STMFD  sp!, {lr}          ; function entry
```

Load scheduling by unrolling



loop_next3

```
LDRB    ca0, [in], #1    ; ca0 = *in++;
LDRB    ca1, [in], #1    ; ca1 = *in++;
LDRB    ca2, [in], #1    ; ca2 = *in++;
SUB      t, ca0, #'A'     ; convert ca0 to lower case
CMP      t, #'Z'-'A'
ADDLS   ca0, ca0, #'a'-'A'
SUB      t, ca1, #'A'     ; convert ca1 to lower case
CMP      t, #'Z'-'A'
ADDLS   ca1, ca1, #'a'-'A'
SUB      t, ca2, #'A'     ; convert ca2 to lower case
CMP      t, #'Z'-'A'
ADDLS   ca2, ca2, #'a'-'A'
```

Load scheduling by unrolling



```
STRB    ca0, [out], #1    ; *out++ = ca0;
TEQ     ca0, #0           ; if (ca0!=0)
STRNEB  ca1, [out], #1    ;   *out++ = ca1;
TEQNE   ca1, #0           ; if (ca0!=0 && ca1!=0)
STRNEB  ca2, [out], #1    ;   *out++ = ca2;
TEQNE   ca2, #0           ; if (ca0!=0 && ca1!=0 && ca2!=0)
BNE     loop_next3       ;   goto loop_next3;
LDMFD   sp!, {pc}        ; return;
```

21 cycles. 7 cycle/character

11/7~1.57

More than doubling the code size

Only efficient for a large data size.

Register allocation



- ATPCS requires called to save R4~R11 and to keep the stack 8-byte aligned.

```
routine_name
```

```
    STMFD sp!,      {r4-r12, lr}
    ; body of routine
    ; the fourteen registers r0-r12 and lr
    LDMFD sp!,      {r4-r12, pc}
```

Do not use sp(R13) and pc(R15)

Total 14 general-purpose registers.

- We stack R12 only for making the stack 8-byte aligned.

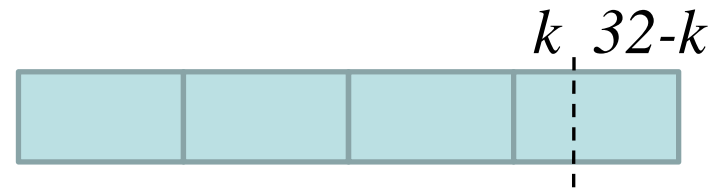
Register allocation



```
unsigned int shift_bits(unsigned int *out, unsigned int *in,
                        unsigned int N, unsigned int k)
{
    unsigned int carry=0, x;
    do
    {
        x = *in++;
        *out++ = (x<<k) | carry;
        carry = x>>(32-k);
        N -= 32;
    } while (N);

    return carry;
}
```

Assume that $K \leq 32$ and N is large and a multiple of 256



Register allocation



Unroll the loop to handle 8 words at a time and to use multiple load/store

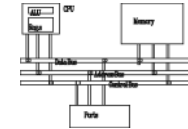
shift_bits

```
    STMFD    sp!, {r4-r11, lr}        ; save registers
    RSB     kr, k, #32                ; kr = 32-k;
    MOV     carry, #0
```

loop

```
    LDMIA   in!, {x_0-x_7}           ; load 8 words
    ORR     y_0, carry, x_0, LSL k    ; shift the 8 words
    MOV     carry, x_0, LSR kr
    ORR     y_1, carry, x_1, LSL k
    MOV     carry, x_1, LSR kr
    ORR     y_2, carry, x_2, LSL k
    MOV     carry, x_2, LSR kr
    ORR     y_3, carry, x_3, LSL k
    MOV     carry, x_3, LSR kr
```

Register allocation



```
ORR    y_4, carry, x_4, LSL k
MOV    carry, x_4, LSR kr
ORR    y_5, carry, x_5, LSL k
MOV    carry, x_5, LSR kr
ORR    y_6, carry, x_6, LSL k
MOV    carry, x_6, LSR kr
ORR    y_7, carry, x_7, LSL k
MOV    carry, x_7, LSR kr
STMIA  out!, {y_0-y_7}           ; store 8 words
SUBS   N, N, #256                ; N -= (8 words * 32 bits)
BNE    loop                       ; if (N!=0) goto loop;
MOV    r0, carry                 ; return carry;
LDMFD  sp!, {r4-r11, pc}
```

Register allocation

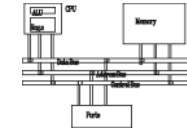


- What variables do we have?

arguments		read-in		overlap	
out	RN 0	x_0	RN 5	y_0	RN 4
in	RN 1	x_1	RN 6	y_1	RN x_0
N	RN 2	x_2	RN 7	y_2	RN x_1
k	RN 3	x_3	RN 8	y_3	RN x_2
		x_4	RN 9	y_4	RN x_3
		x_5	RN 10	y_5	RN x_4
		x_6	RN 11	y_6	RN x_5
		x_7	RN 12	y_7	RN x_6

- We still need to assign **carry** and **kr**, but we have used 13 registers and only one remains.
 - Work on 4 words instead
 - Use stack to save least-used variable, here N
 - Alter the code

Register allocation



- We notice that **carry** does not need to stay in the same register. Thus, we can use y_i for it.

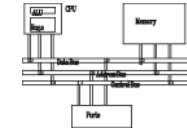
```
kr RN lr
shift_bits
```

```
STMFD sp!, {r4-r11, lr} ; save registers
RSB kr, k, #32 ; kr = 32-k;
MOV y_0, #0 ; initial carry
```

```
loop
```

```
LDMIA in!, {x_0-x_7} ; load 8 words
ORR y_0, y_0, x_0, LSL k ; shift the 8 words
MOV y_1, x_0, LSR kr ; recall x_0 = y_1
ORR y_1, y_1, x_1, LSL k
MOV y_2, x_1, LSR kr
ORR y_2, y_2, x_2, LSL k
MOV y_3, x_2, LSR kr
```

Register allocation



```
ORR    y_3, y_3, x_3, LSL k
MOV    y_4, x_3, LSR kr
ORR    y_4, y_4, x_4, LSL k
MOV    y_5, x_4, LSR kr
ORR    y_5, y_5, x_5, LSL k
MOV    y_6, x_5, LSR kr
ORR    y_6, y_6, x_6, LSL k
MOV    y_7, x_6, LSR kr
ORR    y_7, y_7, x_7, LSL k
STMIA  out!, {y_0-y_7}           ; store 8 words
MOV    y_0, x_7, LSR kr
SUBS   N, N, #256                ; N -= (8 words * 32 bits)
BNE    loop                      ; if (N!=0) goto loop;
MOV    r0, y_0                   ; return carry;
LDMFD  sp!, {r4-r11, pc}
```

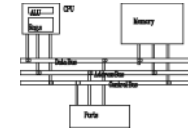
This is often an iterative process until all variables are assigned to registers.

More than 14 local variables



- If you need more than 14 local variables, then you store some on the stack.
- Work outwards from the inner loops since they have more performance impact.

More than 14 local variables



```
nested_loops
```

```
    STMFD    sp!, {r4-r11, lr}  
    ; set up loop 1
```

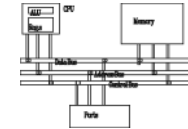
```
loop1
```

```
    STMFD    sp!, {loop1 registers}  
    ; set up loop 2
```

```
loop2
```

```
    STMFD    sp!, {loop2 registers}  
    ; set up loop 3
```


More than 14 local variables



loop3

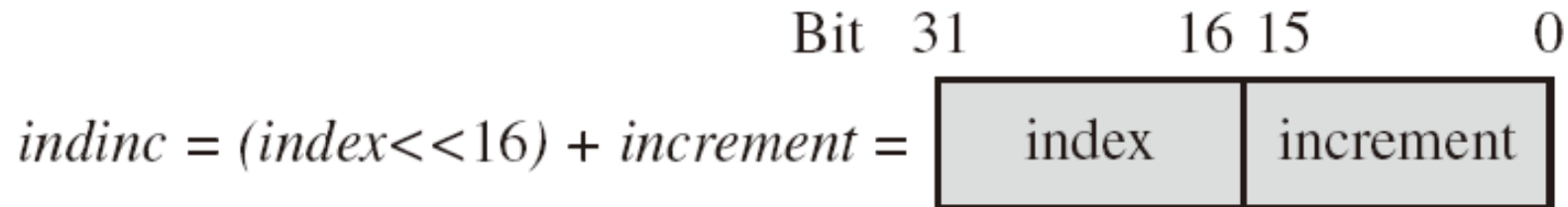
```
    ; body of loop 3
B{cond} loop3
LDMFD  sp!, {loop2 registers}
    ; body of loop 2
B{cond} loop2
LDMFD  sp!, {loop1 registers}
    ; body of loop 1
B{cond} loop1
LDMFD  sp!, {r4-r11, pc}
```

Packing

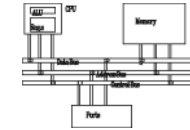


- Pack multiple (sub-32bit) variables into a single register.

```
sample = table[index];  
index += increment;
```



```
LDRB    sample, [table, indinc, LSR#16]    ; table[index]  
ADD     indinc, indinc, indinc, LSL#16     ; index+=increment
```



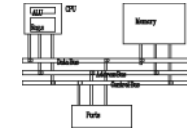
Packing

- When shifting by a register amount, ARM uses bits 0~7 and ignores others.
- Shift an array of 40 entries by shift bits.

$$cntshf = (count \ll 8) + shift =$$

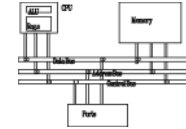
Bit 31	8 7	0
count		shift

Packing



```
out      RN 0    ; address of the output array
in       RN 1    ; address of the input array
cntshf   RN 2    ; count and shift right amount
x        RN 3    ; scratch variable
          ; void shift_right(int *out, int *in, unsigned shift);
shift_right
          ADD     cntshf, cntshf, #39<<8 ; count = 39
shift_loop
          LDR     x, [in], #4
          SUBS    cntshf, cntshf, #1<<8  ; decrement count
          MOV     x, x, ASR cntshf       ; shift by shift
          STR     x, [out], #4
          BGE     shift_loop             ; continue if count>=0
          MOV     pc, lr
```

Packing

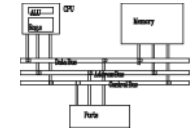


- Simulate SIMD (single instruction multiple data).
- Assume that we want to merge two images X and Y to produce Z by

$$z_n = (ax_n + (256 - a)y_n)/256$$

$$0 \leq a \leq 256$$

Example



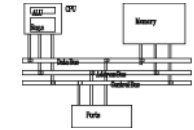
X



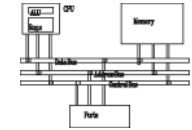
Y

$$X^* \alpha + Y^* (1 - \alpha)$$

$\alpha = 0.75$



$\alpha=0.5$



Packing



- Load 4 bytes at a time

$$[x_3, x_2, x_1, x_0] = x_3 2^{24} + x_2 2^{16} + x_1 2^8 + x_0 =$$

Bit 24	16	8	0
x_3	x_2	x_1	x_0

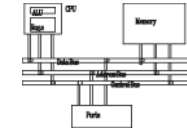
- Unpack it and promote to 16-bit data

$$[x_2, x_0] = x_2 2^{16} + x_0 =$$

Bit 31	16	15	0
x_2	x_0		

- Work on 176x144 images

Packing

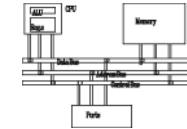


```
IMAGE_WIDTH      EQU 176      ; QCIF width
IMAGE_HEIGHT     EQU 144      ; QCIF height

pz      RN 0      ; pointer to destination image (word aligned)
px      RN 1      ; pointer to first source image (word aligned)
py      RN 2      ; pointer to second source image (word aligned)
a       RN 3      ; 8-bit scaling factor (0-256)

xx      RN 4      ; holds four x pixels [x3, x2, x1, x0]
yy      RN 5      ; holds four y pixels [y3, y2, y1, y0]
x       RN 6      ; holds two expanded x pixels [x2, x0]
y       RN 7      ; holds two expanded y pixels [y2, y0]
z       RN 8      ; holds four z pixels [z3, z2, z1, z0]
count  RN 12     ; number of pixels remaining
mask   RN 14     ; constant mask with value 0x00ff00ff
```

Packing



```
    ; void merge_images(char *pz, char *px, char *py, int a)
```

```
merge_images
```

```
    STMFD    sp!, {r4-r8, lr}
```

```
    MOV     count, #IMAGE_WIDTH*IMAGE_HEIGHT
```

```
    LDR     mask, =0x00FF00FF    ; [ 0, 0xFF, 0, 0xFF ]
```

```
merge_loop
```

```
    LDR     xx, [px], #4        ; [ x3, x2, x1, x0 ]
```

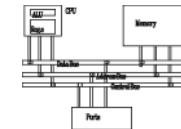
```
    LDR     yy, [py], #4        ; [ y3, y2, y1, y0 ]
```

```
    AND     x, mask, xx        ; [ 0, x2, 0, x0 ]
```

```
    AND     y, mask, yy        ; [ 0, y2, 0, y0 ]
```

```
    SUB     x, x, y            ; [ (x2-y2), (x0-y0) ]
```

Packing



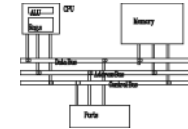
```
MUL    x, a, x           ; [ a*(x2-y2), a*(x0-y0) ]
ADD    x, x, y, LSL#8    ; [          w2,          w0 ]
AND    z, mask, x, LSR#8 ; [ 0, z2, 0, z0 ]
AND    x, mask, xx, LSR#8 ; [ 0, x3, 0, x1 ]
AND    y, mask, yy, LSR#8 ; [ 0, y3, 0, y1 ]
SUB    x, x, y           ; [ (x3-y3), (x1-y1) ]
MUL    x, a, x           ; [ a*(x3-y3), a*(x1-y1) ]
ADD    x, x, y, LSL#8    ; [          w3,          w1 ]
AND    x, mask, x, LSR#8 ; [ 0, z3, 0, z1 ]
ORR    z, z, x, LSL#8    ; [ z3, z2, z1, z0 ]
STR    z, [pz], #4      ; store four z pixels
SUBS   count, count, #4
BGT    merge_loop
LDMFD  sp!, {r4-r8, pc}
```

Conditional execution



- By combining conditional execution and conditional setting of the flags, you can implement simple if statements without any need of branches.
- This improves efficiency since branches can take many cycles and also reduces code size.

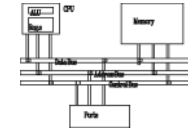
Conditional execution



```
if (i<10)
{
    c = i + '0';
}
else
{
    c = i + 'A'-10;
}
```

```
CMP        i, #10
ADDLO     c, i, #'0'
ADDHS     c, i, #'A'-10
```

Conditional execution



```
if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
{
    vowel++;
}
```

```
TEQ    c, #'a'
TEQNE  c, #'e'
TEQNE  c, #'i'
TEQNE  c, #'o'
TEQNE  c, #'u'
ADDEQ  vowel, vowel, #1
```

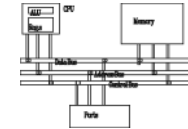
Conditional execution



```
if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
{
    letter++;
}
```

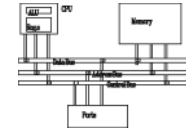
```
    SUB    temp, c, #'A'
    CMP    temp, #'Z'-'A'
    SUBHI  temp, c, #'a'
    CMPHI  temp, #'z'-'a'
    ADDLS  letter, letter, #1
```


Block copy example



```
void bcopy(char *to, char *from, int n)
{
    while (n-- > 0)
        *to++ = *from++;
}
```

Block copy example



```
@ arguments: R0: to, R1: from, R2: n
bcopy:    TEQ    R2, #0
          BEQ    end
loop:     SUB    R2, R2, #1
          LDRB  R3, [R1], #1
          STRB  R3, [R0], #1
          B     bcopy
end:      MOV    PC, LR
```

Block copy example



@ arguments: R0: to, R1: from, R2: n

@ rewrite "n--" as "--n>=0"

```
bcopy:  SUBS    R2, R2, #1
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        BPL   bcopy
        MOV   PC, LR
```

Block copy example



@ arguments: R0: to, R1: from, R2: n

@ assume n is a multiple of 4; loop unrolling

```
bcopy:  SUBS    R2, R2, #4
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        LDRPLB R3, [R1], #1
        STRPLB R3, [R0], #1
        BPL    bcopy
        MOV    PC, LR
```

Block copy example



@ arguments: R0: to, R1: from, R2: n

@ n is a multiple of 16;

```
bcopy:  SUBS    R2, R2, #16
        LDRPL  R3, [R1], #4
        STRPL  R3, [R0], #4
        LDRPL  R3, [R1], #4
        STRPL  R3, [R0], #4
        LDRPL  R3, [R1], #4
        STRPL  R3, [R0], #4
        LDRPL  R3, [R1], #4
        STRPL  R3, [R0], #4
        BPL   bcopy
        MOV   PC, LR
```

Block copy example



@ arguments: R0: to, R1: from, R2: n

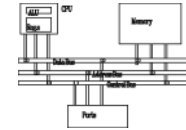
@ n is a multiple of 16;

```
bcopy:  SUBS    R2, R2, #16
        LDMPL  R1!, {R3-R6}
        STMPL  R0!, {R3-R6}
        BPL    bcopy
        MOV    PC, LR
```

@ could be extend to copy 40 byte at a time

@ if not multiple of 40, add a copy_rest loop

Search example



```
int main(void)
{
    int a[10]={7,6,4,5,5,1,3,2,9,8};
    int i;
    int s=4;

    for (i=0; i<10; i++)
        if (s==a[i]) break;
    if (i>=10) return -1;
    else return i;
}
```

Search

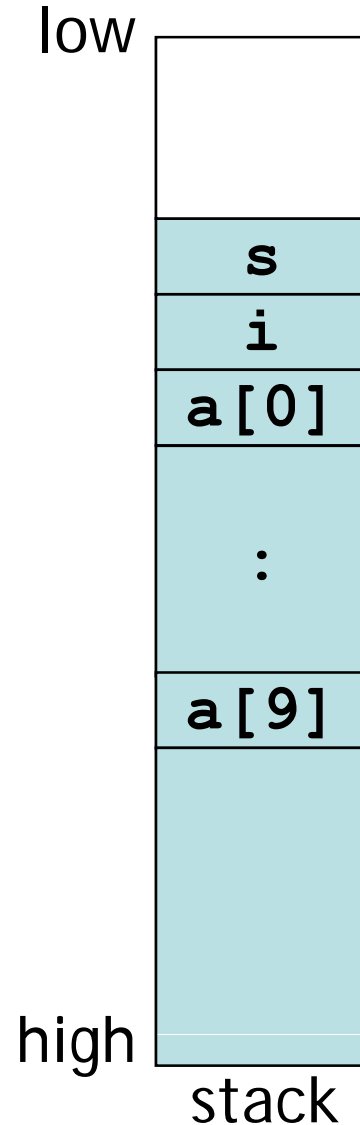


```
        .section    .rodata
.LC0:
        .word      7
        .word      6
        .word      4
        .word      5
        .word      5
        .word      1
        .word      3
        .word      2
        .word      9
        .word      8
```

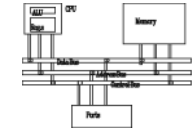

Search



```
.text
.global main
.type main, %function
main: sub sp, sp, #48
      adr r4, L9 @ =.LC0
      add r5, sp, #8
      ldmia r4!, {r0, r1, r2, r3}
      stmia r5!, {r0, r1, r2, r3}
      ldmia r4!, {r0, r1, r2, r3}
      stmia r5!, {r0, r1, r2, r3}
      ldmia r4!, {r0, r1}
      stmia r5!, {r0, r1}
```



Search



```
mov r3, #4
str r3, [sp, #0] @ s=4
mov r3, #0
str r3, [sp, #4] @ i=0

loop: ldr r0, [sp, #4] @ r0=i
      cmp r0, #10 @ i<10?
      bge end
      ldr r1, [sp, #0] @ r1=s
      mov r2, #4
      mul r3, r0, r2
      add r3, r3, #8
      ldr r4, [sp, r3] @ r4=a[i]
```

low

s
i
a[0]
:
a[9]

high

stack

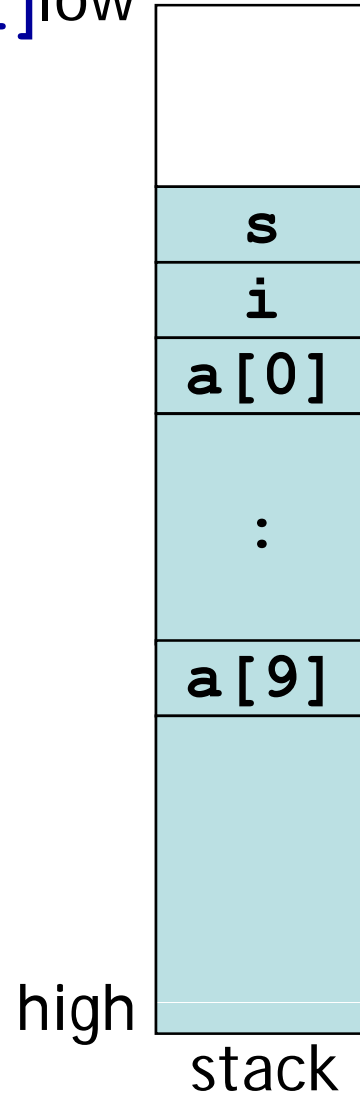
Search



```
teq  r1, r4  @ test if s==a[i]low
beq  end

add  r0, r0, #1  @ i++
str  r0, [sp, #4] @ update i
b    loop

end: str  r0, [sp, #4]
     cmp  r0, #10
     movge r0, #-1
     add  sp, sp, #48
     mov  pc, lr
```



Search (remove load/store)



```
mov    r1, #4
str   r3, [sp, #0] @ s=4
mov    r0, #0
str   r3, [sp, #4] @ i=0

loop:  ldr   r0, [sp, #4] @ r0=i
        cmp   r0, #10 @ i<10?
        bge  end
        ldr   r1, [sp, #0] @ r1=s
        mov   r2, #4
        mul  r3, r0, r2
        add  r3, r3, #8
        ldr  r4, [sp, r3] @ r4=a[i] high
```

low

s
i
a[0]
:
a[9]

high
stack

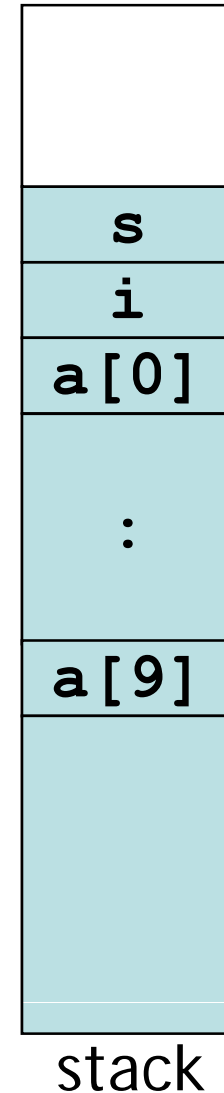
Search (remove load/store)



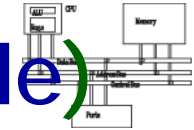
```
teq  r1, r4  @ test if s==a[i]low
beq  end

add  r0, r0, #1  @ i++
str  r0, [sp, #4] @ update i
b    loop

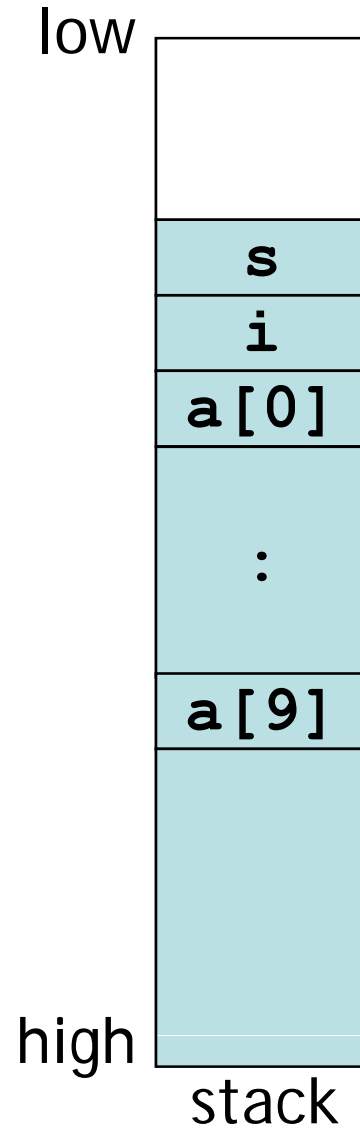
end: str  r0, [sp, #4]
     cmp   r0, #10
     movge r0, #-1
     add   sp, sp, #48
     mov   pc, lr
```



Search (loop invariant/addressing mode)



```
mov r1, #4
str r3, [sp, #0] @ s=4
mov r0, #0
str r3, [sp, #4] @ i=0
mov r2, sp, #8
loop: ldr r0, [sp, #4] @ r0=i
      cmp r0, #10 @ i<10?
      bge end
ldr r1, [sp, #0] @ r1=s
mov r2, #4
      mul r3, r0, r2
add r3, r3, #8
ldr r4, [sp, r3] @ r4=a[i]
      ldr r4, [r2, r0, LSL #2]
```



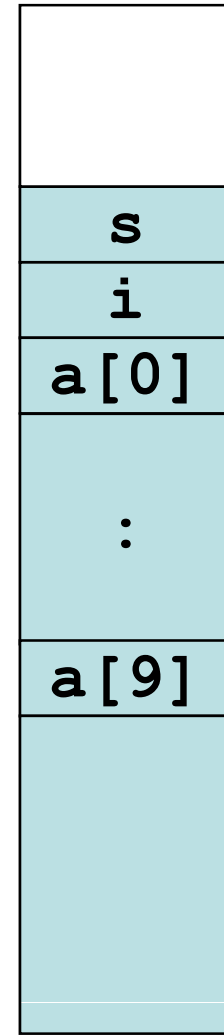
Search (conditional execution)



```
teq r1, r4 @ test if s==a[i]low
beq end

addeq r0, r0, #1 @ i++
str r0, [sp, #4] @ update i
beq loop

end: str r0, [sp, #4]
      cmp r0, #10
      movge r0, #-1
      add sp, sp, #48
      mov pc, lr
```



high
stack

Optimization



- Remove unnecessary load/store
 - Remove loop invariant
 - Use addressing mode
 - Use conditional execution
-
- From 22 words to 13 words and execution time is greatly reduced.