Computer Architecture

Instructions: the Language of Computer

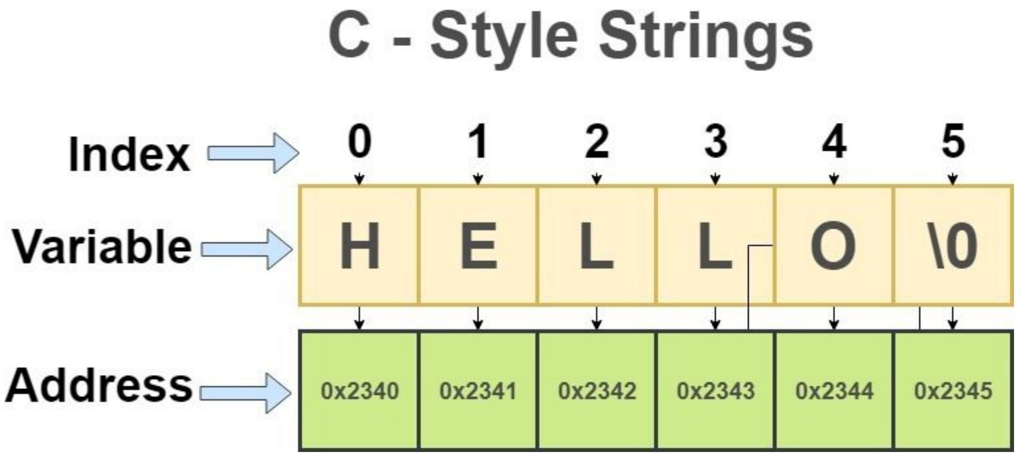
The MIPS instruction set, Part 3

Agenda

- Instructions for character data
- Big constant
- Addressing modes
- Memory map
- Translation and startup
- Arrays vs. Pointers
- Fallacies and pitfalls

Character Data

- A character is represented by a binary
    - ASCII: each character is represented by a 8-bit binary number
    - Unicode: characters are represented by binaries in different length
- So, a string is an array of binary numbers



Byte/Halfword Operations

- String processing needs instructions to **work on bytes**, not the word.
- MPIS provides instructions to move bytes, for example,
    - `lb rt, offset(rs)   #load byte from source`
    - `lh rt, offset(rs)   #load halfword from destination`
    - `lbu rt, offset(rs) #load byte unsigned`
    - `lhu rt, offset(rs) #load halfword unsigned`
    - `sb rt, offset(rs)   #store byte`
    - `sh rt, offset(rs)   #store halfword`

String Copy Example

C code: (null-terminated string, i.e., it ends with `'\0'`)

```
void strcpy (char x[], char y[]) //copy string y to string x
{
    int i;
    i = 0;
    while ((x[i]=y[i] !='\0')
        i += 1;
}
```

We assume

- addresses of `x, y` in `$a0, $a1`
- `i` in `$s0`

String Copy Example

- MIPS code:

```
strcpy:
     addi $sp, $sp, -4          # adjust stack for 1 item
     sw   $s0, 0($sp)           # save $s0
     add  $s0, $zero, $zero     # i = 0
L1:  add  $t1, $s0, $a1         # addr of y[i] in $t1
     lbu  $t2, 0($t1)           # $t2 = y[i]
     add  $t3, $s0, $a0         # addr of x[i] in $t3
     sb   $t2, 0($t3)           # x[i] = y[i]
     beq  $t2, $zero, L2        # exit loop if y[i] == 0
     addi $s0, $s0, 1           # i = i + 1
     j    L1                    # next iteration of loop
L2:  lw   $s0, 0($sp)           # restore saved $s0
     addi $sp, $sp, 4           # pop 1 item from stack
     jr   $ra                   # and return
```

preserve the value of `$s0` in the caller.

Increment the address by 1 not 4 since a string is an array of chars.

restore the value of `$s0`, so it will be the value before the `strcpy` being called.

32-bit Constants

- In some cases, we need a 32-bit constant or 32-bit address.
  - But MIPS instructions are only 32 bits long.
- MIPS offers instruction for handling them.
  - `lui rt, constant #load upper immediate`
    - Copies the upper (i.e., the mostleft) 16-bit of the constant to left 16 bits of `rt`
    - Clears right 16 bits of `rt` to 0
  - `ori rt, rs, constant #bitwise OR`

The machine language version of `lui $t0, 255` `# $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255:`

0000 0000 1111 1111	0000 0000 0000 0000

32-bit Constant Example

- Integer: 612304, which is > 2^{16}-1;
- Its 32-bit binary: 0000 0000 0011 1101 0000 1001 0000 0000;
- We want to load it to `$s0`

```
lui $s0, 61
```
#61_{10} = $0000\ 0000\ 0011\ 1101_2$

- `$s0` afterwards is,

```
0000 0000 0011 1101 0000 0000 0000 0000
```

- Next, insert the lower 16 bit to `$s0`,

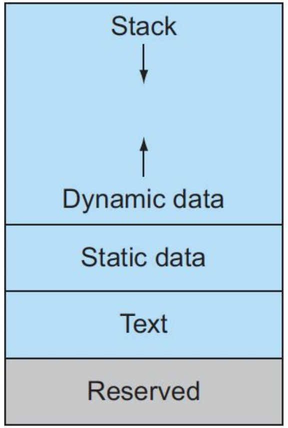
```
ori $s0, $s0, 2304
```
#2340_{10} = $0000\ 1001\ 0000\ 0000_2$

- The final value in `$s0` is,

```
0000 0000 0011 1101 0000 1001 0000 0000
```

Addressing Modes

- A program is a set of instructions stored in memory.
- To run a program, the machine needs to know where to get the instructions.
- MIPS provides 5 addressing modes (two types).
  - Modes for reading and writing operands: register addressing, immediate addressing, base addressing
  - Modes for writing program counter: PC-relative addressing, and pseudo-direct addressing



Modes for reading and writing operands

- **Register addressing**: it uses registers for all source and destination operands.
    - All R-type instructions use this mode
- **Immediate addressing**: it uses the 16-bit immediate along with the register as operands
    - Some I-type instructions use it, e.g., `addi`, and `lui`
- **Base addressing**: adding the base address in register to the 16-bit offset.
    - Memory access instructions use this mode, e.g., `lw`, and `sw`.

PC-relative Addressing

- Branch instruction size problem: the address field for branching is 16 bits only, which could be a limit on the program size.

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- **PC-relative addressing**: the branching target address (BTA) is calculate as the following:

```
BTA =(PC+4)+offset*4
```

  - PC: the address in program counter register; (PC+4) is the address of the next instruction
  - offset: the number of instructions (each takes 4 bytes) between the branch instruction and the target address.

PC-relative addressing

- PC-relative addressing:
  - In practice, most branch targets are near the PC
  - The branch address ranges by PC $\pm 2^{15}$ **words** (offsets are signed integers)
  - The offset is the number of words, not bytes.
- MIPS (and most computers) uses PC-relative addressing for **all conditional branches**

Example: PC-relative addressing

MIPS code:

```
0xA4          beq  $t0, $0, else
0xA8          addi $v0, $0, 1
0xAC          addi $sp, $sp, 8
0xB0          jr   $ra
0xB4    else: addi $a0, $a0, -1
0xB8          jal  factorial
```

Field Values

op	rs	rt	imm
4	8	0	3
6 bits	5 bits	5 bits	16 bits

3 is the number of instructions between `beq` and `else`.
If the condition is true, it will change the address in PC to `0xA4 + 4 + 3*4 = 0xB4`

Pseudo-Direct Addressing

- PC-relative addressing works based on the fact that conditional branches (e.g., `bne`, `beq`) is not very far from the PC. But for jump (`j` and `jal`), the jump target address (JTA) could be anywhere in text segment.

op	address
6 bits	26 bits

- **Pseudo-direct address:**

  `JTA = (PC+4)[31:28]:(offset*4)`

  - (PC+4)[31:28]: the four most significant bits of PC+4
  - offset: it is counted in word, so it should be multiplied by 4.
  - JTA is the concatenation of the 4 most significant bits in PC and the offset.
- This ensures the jump remains within 256MB region of the memory.

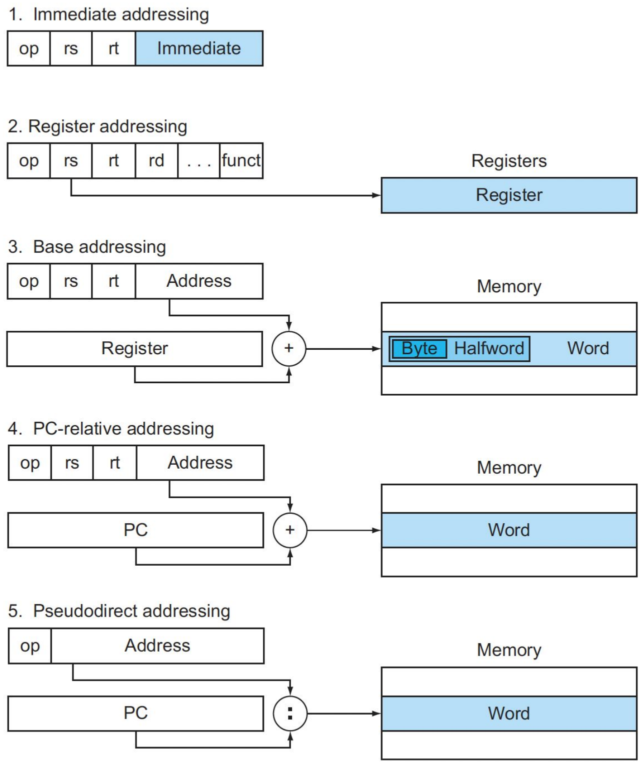
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example: assume `L1` is too far to encode with 16-bit offset, we can rewrite it using `bne` and branch to `L2`, which is originally the instructions following the `beq`, then use unconditional jump to go to `L1`.

```
    beq $s0, $s1, L1
```

```
    bne $s0, $s1, L2
    j L1
L2: …
```

Addressing Modes Summary



Immediate addressing: operand is 16 bits of the instruction itself. (e.g., `addi $sp $sp -4`)

Register addressing: operand is a register (e.g., `add $t3, $s0, $a0`)

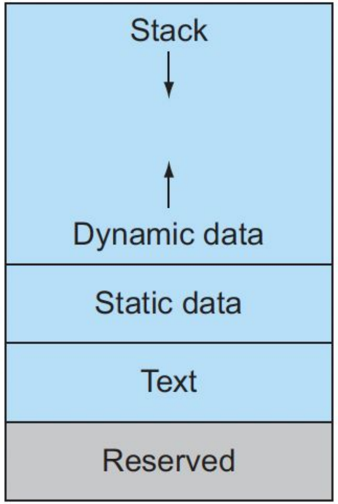
Base addressing: operand is in memory (e.g., `lbu $t2, 0($t1)`)

PC-relative addressing: address instructions in memory; adding a 16-bit address shifted left 2 bits to the PC. (e.g., `beq $t2, $zero, L1`)

Pseudo-direct addressing: concatenating a 26-bit address shifted left 2 bits with 4 upper bits of the PC. (e.g., `j L1`)

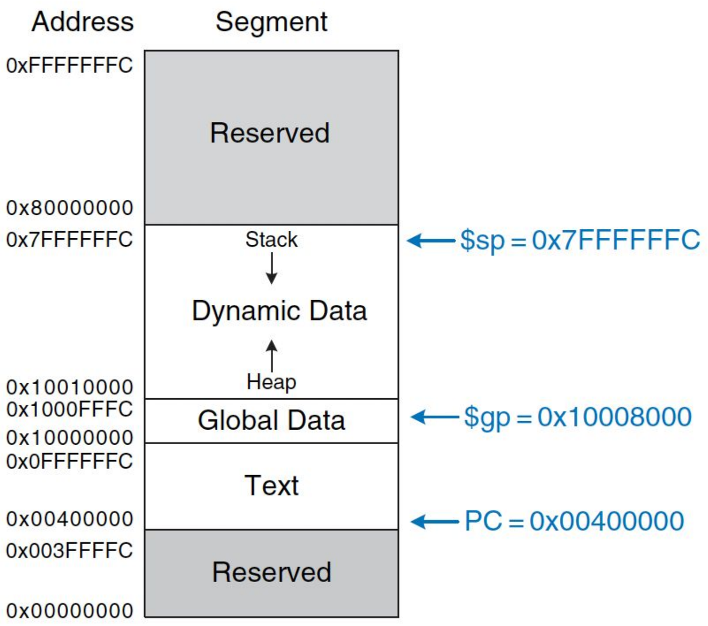
Memory Map

- MIPS address space spans 2^{32} bytes = 4 GB.
- Each MIPS program can have maximum 4 GB memory space. (The size of a MIPS program ≤ 4GB.)
- The address space is divided into four segments:
  - **Text segment**: stores the machine language program. (≤ 256 MB)
  - **Global data segment**: stores global variables that can be seen by all functions in the program. (≤ 64 KB)
  - **Dynamic data segment**: holds stack and heap. (~2GB)
  - **Reserved segments**: used by the operating system and cannot directly be used by the program.



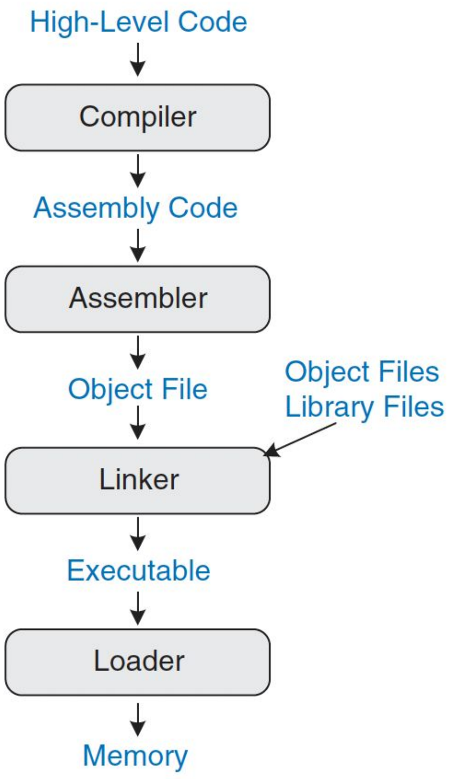
Memory Map

- Each MIPS program will be loaded into memory following this layout.
- The address starts from `0` and ends at `0xFFFFFFFC`.
- The stack grows downward from `0x7FFFFFC`. Usually, the size of the stack is determined by the OS. (e.g., for linux/Unix/Mac OS, it is 8 MB)
- `$gp` is set to be `0x10008000`, allowing ±offset into the global data segment.



Translation and Startup

- A high-level language program is first compiled into an assembly language program, and then,
- it is assembled into an object module in machine language.
- The linker combines multiple modules with library routines to resolve all references.
- The loader then places the machine code into the proper memory locations for execution by the processor.



Compilation

A compiler translates high-level code into assembly language.

High-Level Code

```
int f, g, y; // global variables

int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

MIPS Assembly Code

```
.data
f:
g:
y:

.text
main:
  addi $sp, $sp, −4   # make stack frame
  sw   $ra, 0($sp)    # store $ra on stack
  addi $a0, $0, 2     # $a0 = 2
  sw   $a0, f         # f = 2
  addi $a1, $0, 3     # $a1 = 3
  sw   $a1, g         # g = 3
  jal  sum            # call sum function
  sw   $v0, y         # y = sum(f, g)
  lw   $ra, 0($sp)    # restore $ra from stack
  addi $sp, $sp, 4    # restore stack pointer
  jr   $ra            # return to operating system

sum:
  add  $v0, $a0, $a1 # $v0 = a + b
  jr   $ra            # return to caller
```

Assembling

The assembler turns the assembly code into an object file containing machine code.

It makes two passes through the assembly code:

- 1st pass: the assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names.
- 2nd pass: it produces the machine language code.

The first pass in assembling

After the first pass, the names and addresses of the symbols are kept in a symbol table.

- Global variables are assigned storage locations in the global data segment, starting at address `0x10000000`.

MIPS Assembly Code

```
.data
f:
g:
y:

.text
main:
  addi $sp, $sp, −4   # make stack frame
  sw   $ra, 0($sp)    # store $ra on stack
  addi $a0, $0, 2     # $a0 = 2
  sw   $a0, f         # f = 2
  addi $a1, $0, 3     # $a1 = 3
  sw   $a1, g         # g = 3
  jal  sum            # call sum function
  sw   $v0, y         # y = sum(f, g)
  lw   $ra, 0($sp)    # restore $ra from stack
  addi $sp, $sp, 4    # restore stack pointer
  jr   $ra            # return to operating system

sum:
  add  $v0, $a0, $a1 # $v0 = a + b
  jr   $ra            # return to caller
```

```
0x00400000 main: addi $sp, $sp, −4
0x00400004       sw   $ra, 0($sp)
0x00400008       addi $a0, $0, 2
0x0040000C       sw   $a0, f
0x00400010       addi $a1, $0, 3
0x00400014       sw   $a1, g
0x00400018       jal  sum
0x0040001C       sw   $v0, y
0x00400020       lw   $ra, 0($sp)
0x00400024       addi $sp, $sp, 4
0x00400028       jr   $ra
0x0040002C sum:  add  $v0, $a0, $a1
0x00400030       jr   $ra
```

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

The second pass in assembling

The assembler converts each assembly instruction into its corresponding machine code using the symbol table.

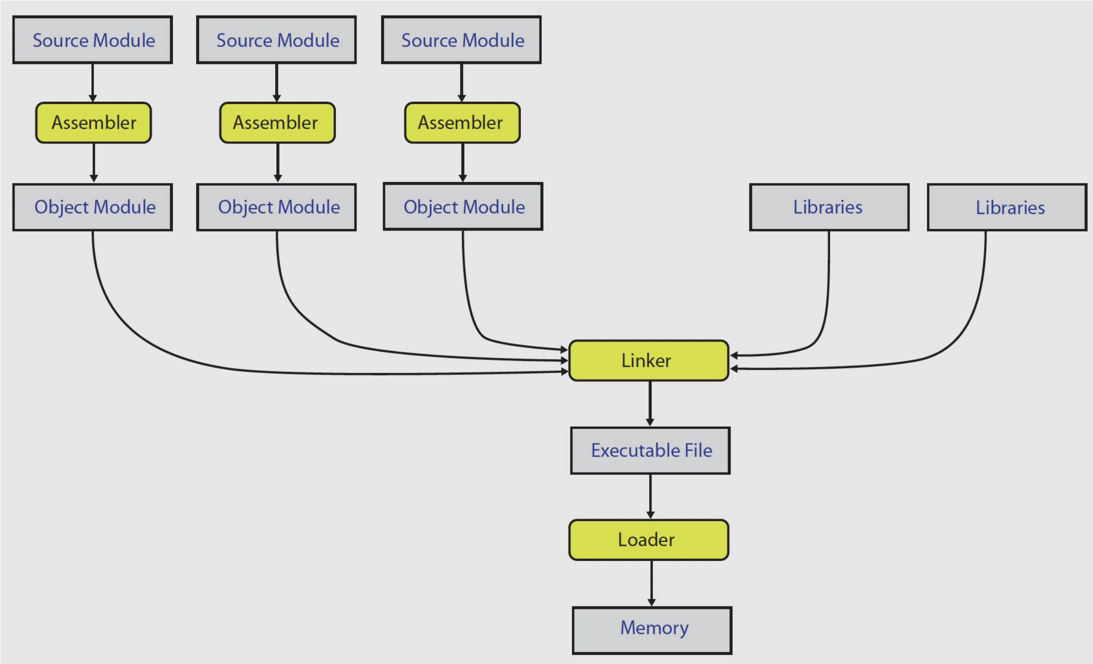
It substitutes labels and global variables with the addresses in the symbol table.

The machine code and symbol table are stored in the object file.

Text segment	Address	Instruction	
	0x00400000	0x23BDFFFC	addi $sp, $sp, –4
	0x00400004	0xAFBF0000	sw $ra, 0($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008($gp)
	0x00400020	0x8FBF0000	lw $ra, 0($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, –4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E00008	jr $ra

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

Linking



Large programs contain more than one file. Each file is translated into object file.

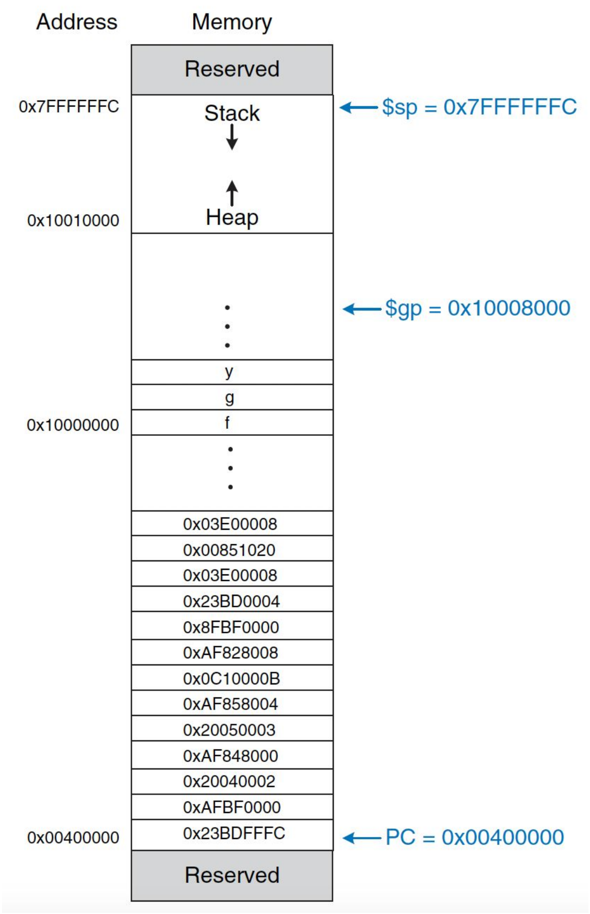
The linker combines all of the object files into one machine code file called executable.

So, it only needs to recompile the file that is changed by the programer, not all files.

Loading

The OS loads a program by reading the text segment of the executable file from a storage device (e.g., the hard drive) into text segment memory.

The OS sets `$gp` to `0x10008000` (i.e., the middle of the global segment) and `$sp` to `0x7FFFFFFC` (the top of the dynamic data segment), then, put `0x00400000` into PC to jump to the beginning of the program.



A C sort example to put it all together

```c
void swap(int v[], int k){
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

void sort(int v[], int n){
    int i, j;
    for(i=0; i<n; i+=1){
        for(j=i-1; j>=0 && v[j]>v[j+1]; j-=1){
            swap(v, j);
        }
    }
}
```

- Two procedures: one to swap array elements and one to sort them

The Procedure Swap (leaf)

- assuming `v` in `$a0`, `k` in `$a1`, `temp` in `$t0`

```c
void swap(int v[], int k){
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

The procedure body

- assuming `v` in `$a0`, `k` in `$a1`, `i` in `$s0`, `j` in `$s1`

```c
void sort(int v[], int n){
    int i, j;
    for(i=0; i<n; i+=1){
        for(j=i-1; j>=0 && v[j]>v[j+1]; j-=1){
            swap(v, j);
        }
    }
}
```

	move $s2, $a0	# save $a0 into $s2	Move params
	move $s3, $a1	# save $a1 into $s3	
	move $s0, $zero	# i = 0	Outer loop
for1tst:	slt $t0, $s0, $s3	# $t0 = 0 if $s0 ≥ $s3 (i ≥ n)	
	beq $t0, $zero, exit1	# go to exit1 if $s0 ≥ $s3 (i ≥ n)	Inner loop
	addi $s1, $s0, –1	# j = i – 1	
for2tst:	slti $t0, $s1, 0	# $t0 = 1 if $s1 < 0 (j < 0)	
	bne $t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)	
	sll $t1, $s1, 2	# $t1 = j * 4	
	add $t2, $s2, $t1	# $t2 = v + (j * 4)	
	lw $t3, 0($t2)	# $t3 = v[j]	
	lw $t4, 4($t2)	# $t4 = v[j + 1]	
	slt $t0, $t4, $t3	# $t0 = 0 if $t4 ≥ $t3	
	beq $t0, $zero, exit2	# go to exit2 if $t4 ≥ $t3	
	move $a0, $s2	# 1st param of swap is v (old $a0)	Pass params & call
	move $a1, $s1	# 2nd param of swap is j	
	jal swap	# call swap procedure	
	addi $s1, $s1, –1	# j –= 1	Inner loop
	j for2tst	# jump to test of inner loop	
exit2:	addi $s0, $s0, 1	# i += 1	Outer loop
	j for1tst	# jump to test of outer loop	

The full procedure

```
sort:     addi  $sp,$sp, -20        # make room on stack for 5 registers
          sw $ra, 16($sp)           # save $ra on stack
          sw $s3,12($sp)            # save $s3 on stack
          sw $s2, 8($sp)            # save $s2 on stack
          sw $s1, 4($sp)            # save $s1 on stack
          sw $s0, 0($sp)            # save $s0 on stack
          …                         # procedure body
          …
          exit1: lw $s0, 0($sp)     # restore $s0 from stack
          lw $s1, 4($sp)            # restore $s1 from stack
          lw $s2, 8($sp)            # restore $s2 from stack
          lw $s3,12($sp)            # restore $s3 from stack
          lw $ra,16($sp)            # restore $ra from stack
          addi  $sp,$sp, 20         # restore stack pointer
          jr $ra                    # return to calling routine
```

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing an Array

We show C and MIPS assembly of two procedures to clear a sequence of words in memory:

- One using array indices
- One using pointers

Array version of Clear

- C code

```
clear1 (int array[], int size) {
    int i;
    for (i=0; i<size; i+=1)
    {
        array[i] = 0;
    }
}
```

- `array` in `$a0`, `size` in `$a1`, `i` in `$t0`

```
       move $t0,$zero    # i = 0
loop1: sll $t1,$t0,2     # $t1 = i * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[i]
       sw $zero, 0($t2)  # array[i] = 0
       addi $t0,$t0,1    # i = i + 1
       slt $t3,$t0,$a1   # $t3 =
                         #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

Pointer version of Clear

- C code:

```
clear2(int *array, int size) {
    int *p;
    for(p=&array[0]; p<&array[size]; p=p+1)
    {
        *p=0;
    }
}
```

```
       move $t0,$a0      # p = & array[0]
       sll $t1,$a1,2     # $t1 = size * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[size]
loop2: sw $zero,0($t0)   # Memory[p] = 0
       addi $t0,$t0,4    # p = p + 4
       slt $t3,$t0,$t2   # $t3 =
                         #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

Comparison

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
       move $t0,$zero    # i = 0
loop1: sll $t1,$t0,2     # $t1 = i * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1    # i = i + 1
       slt $t3,$t0,$a1   # $t3 =
                         #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

6 instructions in loop

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       move $t0,$a0     # p = & array[0]
       sll $t1,$a1,2    # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 =
                        #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2 # $t3 =
                        #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

4 instructions in loop

Comparison of Array vs. Ptr

- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler (with optimization) can achieve same effect as manual use pointers
  - Induction variable elimination
  - Better to make program clearer and safer

ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Fallacies

- Fallacy: more powerful instructions mean higher performance
  - A powerful instruction ⇒ complete a complex task with less instructions
  - But a instruction that can complete a complex task is often hard to implement
    - These instructions often require complex circuits ⇒ Many slow down all instructions, including simple ones.
- Fallacy: write in assembly language to obtain the highest performance
  - Modern compilers are better at dealing with modern processors
  - Gap between compiler generated assembly code and assembly code produced by hand is closing fast. (actually, current compilers can often do better than human programmers in assembly language.)

Pitfalls

- Sequential words are not at sequential address
  - Increment by 4, not by 1!
- Pitfall: using a pointer to an automatic variable (i.e., those stored by `$sp`) outside its defining procedure.
  - e.g., passing pointer back via an argument; note: following the stack discipline, the memory that contains the local array will be reused as soon as the procedure returns.
  - Pointer becomes invalid when stack popped.