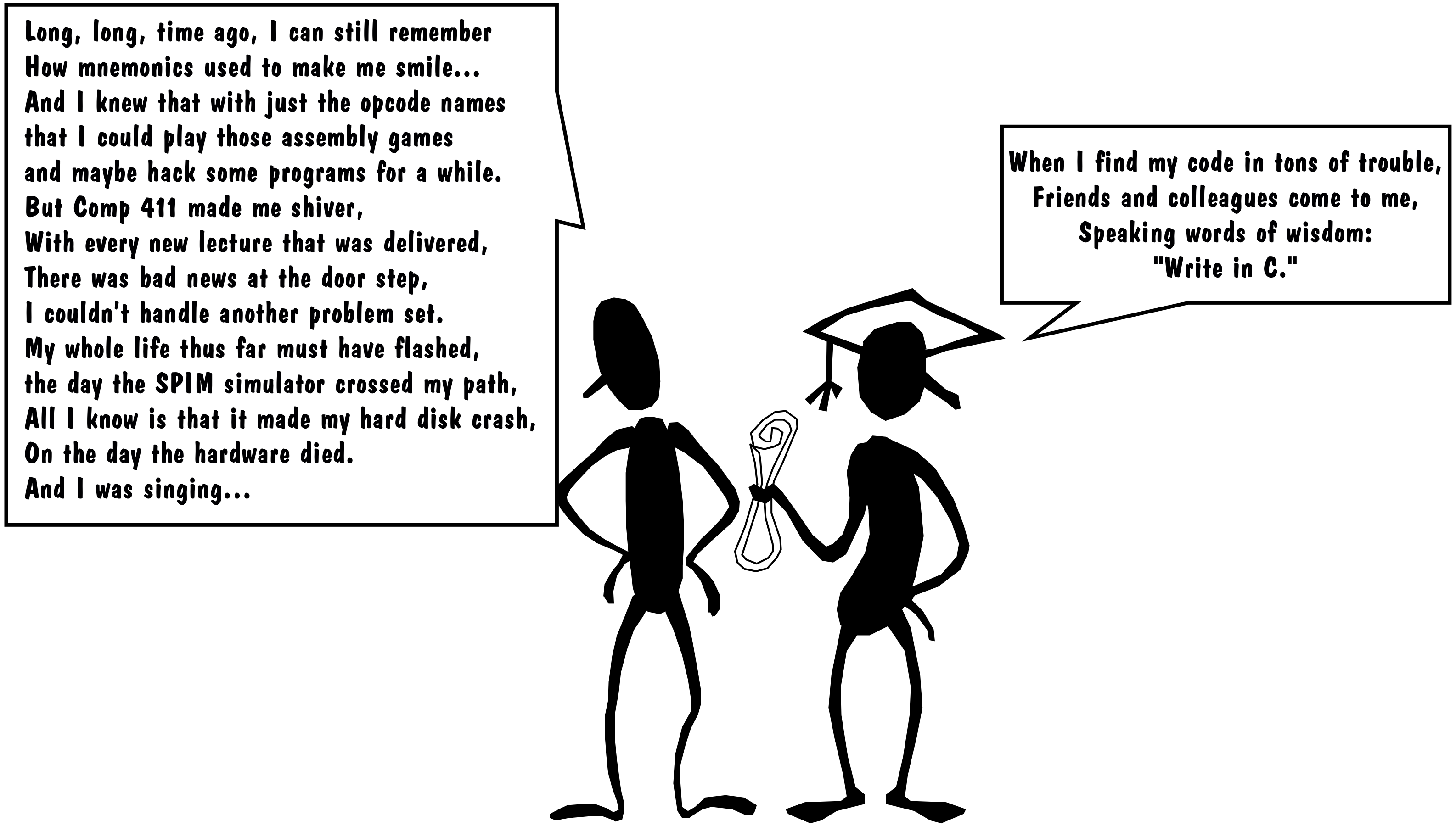
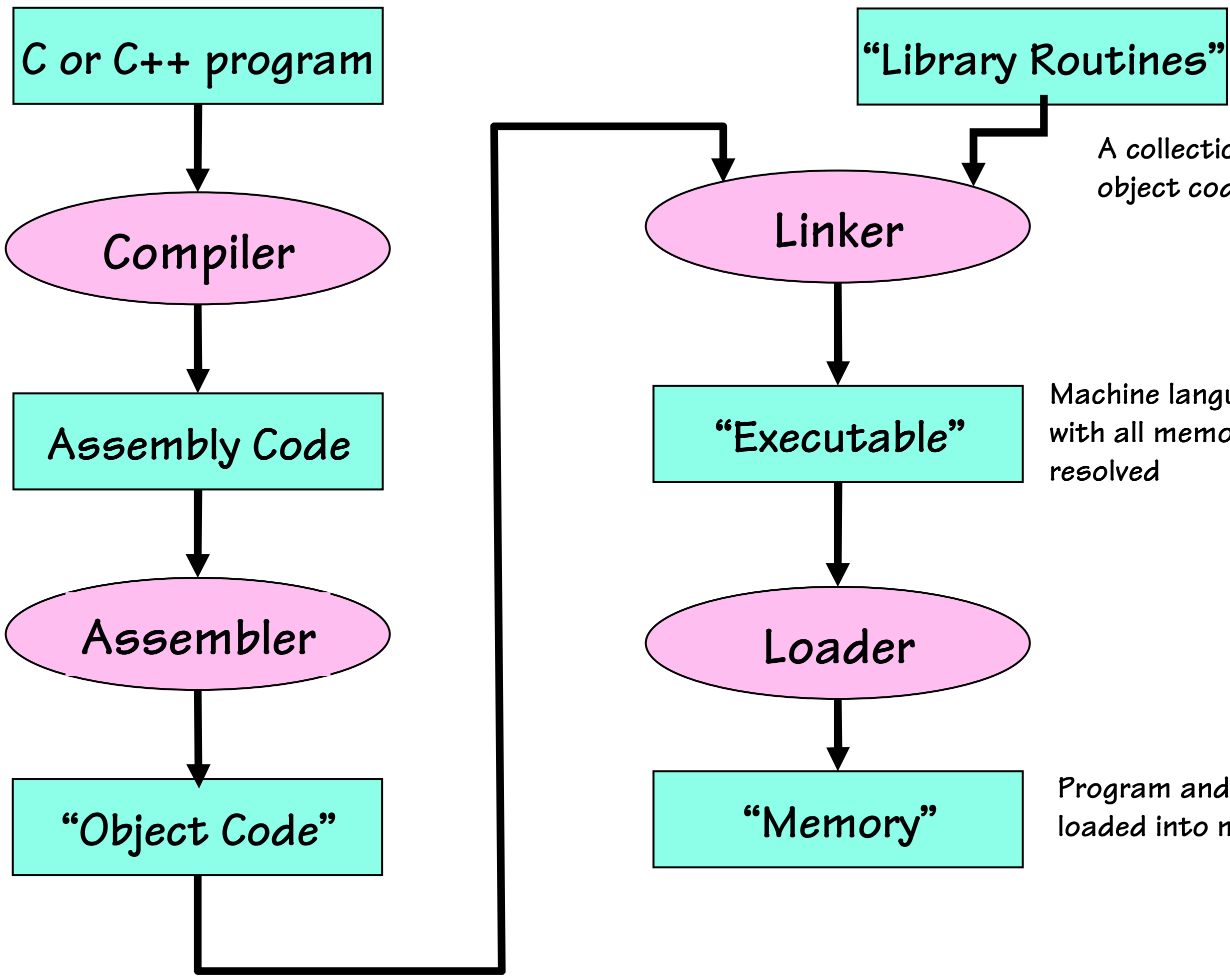
Assemblers and Compilers



Study sections 2.10-2.15

Path from Programs to Bits

- Traditional Compilation



High-level, portable (architecture independent) program description

Architecture dependent mnemonic program description with symbolic memory references

Machine language with symbolic memory references

A collection of precompiled object code modules

Machine language with all memory references resolved

Program and data bits loaded into memory

How an Assembler Works

Three major components of assembly

1) Allocating and initialing data storage
2) Conversion of mnemonics to binary instructions
3) Resolving addresses

```
.data
array:    .space 40
total:    .word 0
```

```
.text
.globl main
main:     la       $t1,array
          move     $t2,$0
          move     $t3,$0
          beq      $0,$0,test
loop:     sll      $t0,$t3,2
          add      $t0,$t1,$t0
          sw       $t3,($t0)
          add      $t2,$t2,$t3
          addi     $t3,$t3,1
test:     slti     $t0,$t3,10
          bne      $t0,$0,loop
          sw       $t2,total
          j        $ra
```

```
lui    $9, arrayhi
ori    $9,$9,arraylo
```

```
0x3c09????
0x3529????
```

Resolving Addresses- 1st Pass

- "Old-style" 2-pass assembler approach

Pass 1

Segment offset	Code	Instruction
0 4	0x3c090000 0x35290000	la $t1,array
8 12	0x00005021 0x00005821	move $t2,$ move $t3,$0
16	0x10000000	beq $0,$0,test
20	0x000b4080	loop: sll $t0,$t3,2

24
28
32
36
	0x01284020
	0xad0b0000
	0x014b5020
	0x216b0001
		add $t0,$t1,$t0
		sw $t0,($t0)
		add $t0,$t1,$t0
		addi $t3,$t3,1

40	0x2968000a	test: slti $t0,$t3,10
44	0x15000000	bne $t0,$0,loop
48 52	0x3c010000 0xac2a0000	sw $t2,total
56	0x03e00008	j $ra

- In the first pass, data and instructions are encoded and assigned offsets within their segment, while the symbol table is constructed.
- Unresolved address references are set to 0

Symbol table after Pass 1

Symbol	Segment	Location pointer offset
array	data	0
total	data	40
main	text	0
loop	text	20
test	text	40

Resolving Addresses - 2nd Pass

- "Old-style" 2-pass assembler approach

Pass 2

Segment offset	Code	Instruction
0 4	0x3c091001 0x35290000	la $t1,array
8 12	0x00005021 0x00005821	move $t2,$ move $t3,$0
16	0x10000006	beq $0,$0,test
20	0x000b4080	loop: sll $t0,$t3,2

24
28
32
36
	0x01284020
	0xad0b0000
	0x014b5020
	0x216b0001
		add $t0,$t1,$t0
		sw $t0,($t0)
		add $t0,$t1,$t0
		addi $t3,$t3,1

40	0x2968000a	test: slti $t0,$t3,10
44	0x1500fffa	bne $t0,$0,loop
48 52	0x3c011001 0xac2a0028	sw $t2,total
56	0x03e00008	j $ra

– In the second pass, the appropriate fields of those instructions that reference memory are filled in with the correct values if possible.

Symbol table after Pass 1

Symbol	Segment	Location pointer offset
array	data	0
total	data	40
main	text	0
loop	text	20
test	text	40

Modern Way – 1-Pass Assemblers

Modern assemblers keep more information in their symbol table which allows them to resolve addresses in a single pass.

- Known addresses (backward references) are immediately resolved.
- Unknown addresses (forward references) are "back-filled" once they are resolved.

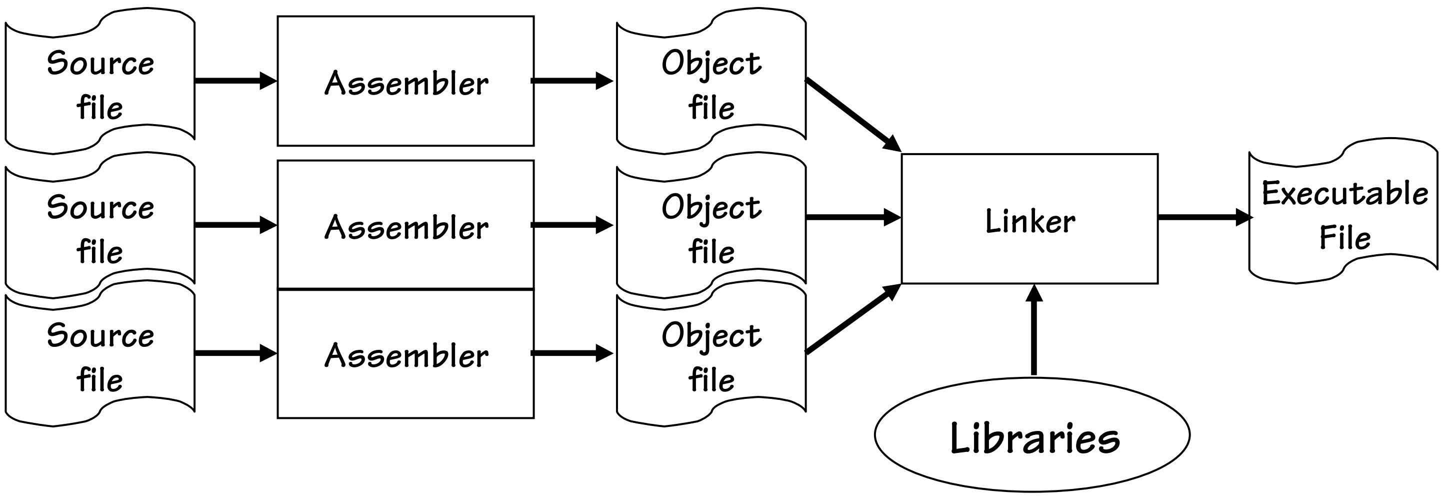
SYMBOL	SEGMENT	Location pointer offset	Resolved ?	Reference list
array	data	0	y	null
total	data	40	y	null
main	text	0	y	null
loop	text	16	y	null
test	text	?	n	16

The Role of a Linker

Some aspects of address resolution cannot be handled by the assembler alone.

1) References to data or routines in other object modules
2) The layout of all segments in memory
3) Support for REUSABLE code modules
4) Support for RELOCATABLE code modules

This final step of resolution is the job of a LINKER



Static and Dynamic Libraries

- LIBRARIES are commonly used routines stored as a concatenation of "Object files". A global symbol table is maintained for the entire library with entry points for each routine.
- The routines in LIBRARIES can be referenced by assembly modules. The appropriate routines entry points are resolved by LINKER, and the appropriate code is added to the executable. This sort of linking is called STATIC.
- Many programs use common libraries. It is wasteful of both memory and disk space to include the same code in multiple executables. The modern alternative to STATIC linking is to allow a LOADER to resolve the addresses of some libraries routines. This form of lining is called DYNAMIC linking (e.x. .dll).

Dynamically Linked Libraries

- C call to library function:

  `printf(“sqr[%d] = %d\n”, x, y);`

- Assembly code

```
addi   $a0,$0,1
la     $a1,ctrlstring
lw     $a2,x
lw     $a3,y
call   fprintf
```

- Maps to:

```
addi   $a0,$0,1
lui    $a1,ctrlstringHi
ori    $a1,ctrlstringLo
lui    $at,xhi
lw     $a2,xlo($at)
lw     $a3,ylo($at)
lui    $at,fprintfHi
ori    $at,fprintfLo
jar     $at
```

Dynamically Linked Libraries

- Lazy address resolution:

```
sysload:        addui $sp,$sp,16
                .
                .
                # check if stdio module
                # is loaded, if not load it
                .
                .
                # backpatch jump table
                la      $t1,stdio
                la      $t0,$dfopen
                sw      $t0,($t1)
                la      $t0,$dfclose
                sw      $t0,4($t1)
                la      $t0,$dfputc
                sw      $t0,8($t1)
                la      $t0,$dfgetc
                sw      $t0,12($t1)
                la      $t0,$dfprintf
                sw      $t0,16($t1)
```

- Before any call is made to a procedure in “stdio.dll”

```
.globl stdio:
stdio:
fopen:   sysload
fclose:  sysload
fgetc:   sysload
fputc:   sysload
fprintf: sysload
```

- After first call is made to any procedure in “stdio.dll”

```
.globl stdio:
stdio:
fopen:   dfopen
fclose:  dclose
fgetc:   dfgetc
fputc:   dfputc
fprintf: dprintf
```

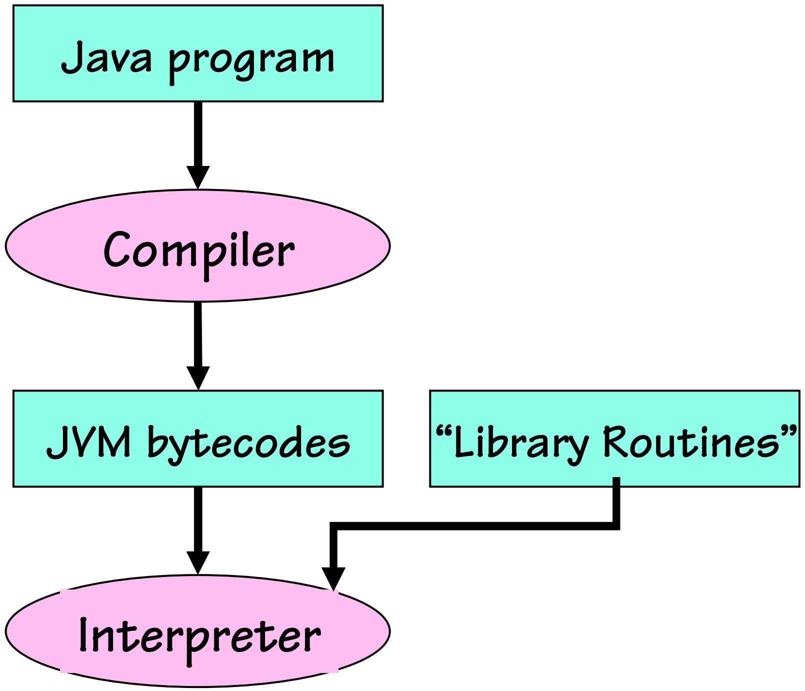
Modern Languages

- Intermediate "object code language"

High-level, portable (architecture independent) program description

PORTABLE mnemonic program description with symbolic memory references

An application that EMULATES a virtual machine. Can be written for any Instruction Set Architecture. In the end, machine language instructions must be executed for each JVM bytecode



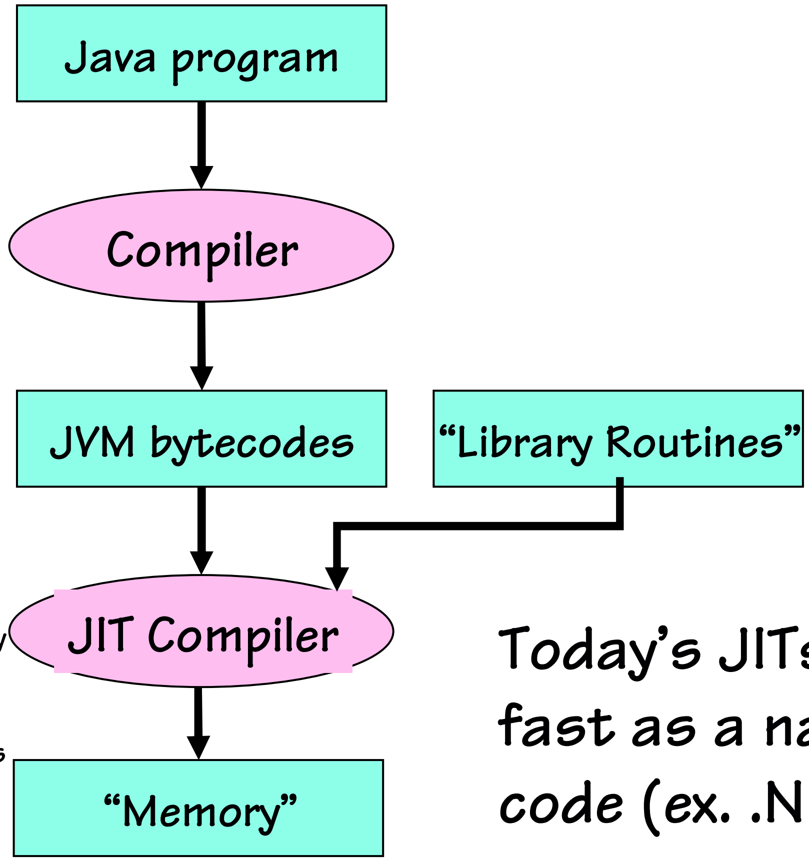
Modern Languages

- Intermediate "object code language"

High-level, portable (architecture independent) program description

PORTABLE mnemonic program description with symbolic memory references

While interpreting on the first pass it keeps a copy of the machine language instructions used. Future references access machine language code, avoiding further interpretation



Today's JITs are nearly as fast as a native compiled code (ex. .NET).

Compiler Optimizations

- Example “C” Code:

```
int a[10];
int total;

int main( ) {
    int i;

    total = 0;
    for (i = 0; i < 10; i++) {
        a[i] = i;
        total = total + i;
    }
}
```

Unoptimized Assembly Output

- With debug flags set:

```
.globl main
.text
main:
     addu $sp,$sp,-8         # allocates space for ra and i
     sw $0,total             # total = 0
     sw $0,0($sp)            # i = 0
     lw $8,0($sp)            # copy i to $t0
     b  L.3                  # goto test
L.2:                         # for(...) {
     sll $24,$8,2            #  make i a word offset
     sw $8,array($24)        #  array[i] = i
     lw $24,total            #  total = total + i
     addu $24,$24,$8
     sw $24,total
     addi $8,$8,1            #  i = i + 1
L.3:
     sw $8,0($sp)            #  update i in memory
     la $24,10               #  loads const 10
     blt $8,$24,L.2          #} loops while i < 10
     addu $sp,$sp,8
     j $31
```

Register Allocation

- Assign local variables to registers

```
.globl main
.text
main:
     addu $sp,$sp,-4          #allocates space for ra
     sw $0,total              #total = 0
     move $8,$0               #i = 0
     b  L.3                   #goto test
L.2:                          #for(...) {
     sll $24,$8,2             #  make i a word offset
     sw $8,array($24)         #  array[i] = i
     lw $24,total             #  total = total + i
     addu $24,$24,$8
     sw $24,total
     addi $8,$8,1             #  i = i + 1
L.3:
     la $24,10                #  loads const 10
     blt $8,$24,L.2           #} loops while i < 10
     addu $sp,$sp,4
     j $31
```

Loop-Invariant Code Motion

- Assign globals to temp registers and move assignments outside of loop

```
.globl main
.text
main:
     addu $sp,$sp,-4          #allocates space for ra
     sw $0,total              #total = 0
     move $9,$0               #temp for total
     move $8,$0               #i = 0
     b  L.3                   #goto test
L.2:                          #for(...) {
     sll $24,$8,2             #  make i a word offset
     sw $8,array($24)         #  array[i] = i
     addu $9,$9,$8
     sw $9,total
     addi $8,$8,1             #  i = i + 1
L.3:
     la $24,10                #  loads const 10
     blt $8,$24,L.2           #} loops while i < 10
     addu $sp,$sp,4
     j $31
```

Remove Unnecessary Tests

- Since i is initially set to 0, we know it is less than 10

```
.globl main
.text
main:
     addu $sp,$sp,-4        #allocates space for ra
     sw $0,total            #total = 0
     move $9,$0             #temp for total
     move $8,$0             #i = 0
L.2:                        #for(...) {
     sll $24,$8,2           #  make i a word offset
     sw $8,array($24)       #  array[i] = i
     addu $9,$9,$8
     addi $8,$8,1           #  i = i + 1
     slti $24,$8,10         #  loads const 10
     bne $24,$0,L.2         #} loops while i < 10
     sw  $9,total
     addu $sp,$sp,4
     j $31
```

Remove Unnecessary Stores

- All we care about it the value of total after the loop, and simplify loop

```
.globl main
.text
main:
     addu $sp,$sp,-4          #allocates space for ra and i
     sw $0,total              #total = 0
     move $9,$0               #temp for total
     move $8,$0               #i = 0
L.2:
     sll $24,$8,2             #for(...) {
     sw $8,array($24)         #  array[i] = i
     addu $9,$9,$8
     addi $8,$8,1             #  i = i + 1
     slti $24,$8,10           #  loads const 10
     bne $24,$0,L.2           #} loops while i < 10
     sw  $9,total
     addu $sp,$sp,4
     j $31
```

Unrolling Loop

- Two copies of the inner loop reduce the branching overhead

```
.globl main
.text
main:
     addu $sp,$sp,-4          #allocates space for ra and i
     sw $0,total              #total = 0
     move $9,$0               #temp for total
     move $8,$0               #i = 0
L.2:
     sll $24,$8,2             #for(...) {
     sw $8,array($24)         #  array[i] = i
     addu $9,$9,$8
     addi $8,$8,1             #  i = i + 1
     sll $24,$8,2             #
     sw $8,array($24)         #  array[i] = i
     addu $9,$9,$8
     addi $8,$8,1             #  i = i + 1
     slti $24,$8,10           #  loads const 10
     bne $24,$0,L.2           #} loops while i < 10
     sw  $9,total
     addu $sp,$sp,4
     j $31
```