Five classic components

I am like a control tower
I am like a pack of file folders
I am like a conveyor belt + service stations
I exchange information with outside world

CS/COE0447: Computer Organization and Assembly Language

Chapter 2

Sangyeun Cho
Dept. of Computer Science
University of Pittsburgh

MIPS operations and operands

- Operation specifies what function to perform by the instruction
- Operand specifies with what a specific function is to be performed by the instruction

- MIPS operations
  - Arithmetic (integer/floating-point)
  - Logical
  - Shift
  - Compare
  - Load/store
  - Branch/jump
  - System control and coprocessor

- MIPS operands
  - Registers
  - Fixed registers (e.g., HI/LO)
  - Memory location
  - Immediate value

MIPS arithmetic

- \( <\text{op}> <r_{\text{target}} > <r_{\text{source1}} > <r_{\text{source2}} > \)

- All arithmetic instructions have 3 operands
  - Operand order in notation is fixed; target first
  - Two source registers and one target or destination register

- Examples
  - \( \text{add } s1, s2, s3 \) \# \( s1 \leftarrow s2 + s3 \)
  - \( \text{sub } s4, s5, s6 \) \# \( s4 \leftarrow s5 - s6 \)
### MIPS registers

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$at</td>
<td>$a0</td>
</tr>
<tr>
<td>$v0</td>
<td>$v1</td>
<td>$v2</td>
</tr>
<tr>
<td>$a0</td>
<td>$a1</td>
<td>$a2</td>
</tr>
<tr>
<td>$a3</td>
<td>$t0</td>
<td>$t1</td>
</tr>
<tr>
<td>$t2</td>
<td>$t3</td>
<td>$t4</td>
</tr>
<tr>
<td>$t5</td>
<td>$t6</td>
<td>$t7</td>
</tr>
<tr>
<td>$r15</td>
<td>$r16</td>
<td>$s0</td>
</tr>
<tr>
<td>$r17</td>
<td>$r18</td>
<td>$s1</td>
</tr>
<tr>
<td>$r19</td>
<td>$r20</td>
<td>$s2</td>
</tr>
<tr>
<td>$r21</td>
<td>$r22</td>
<td>$s3</td>
</tr>
<tr>
<td>$r23</td>
<td>$r24</td>
<td>$s4</td>
</tr>
<tr>
<td>$r25</td>
<td>$r26</td>
<td>$s5</td>
</tr>
<tr>
<td>$r27</td>
<td>$r28</td>
<td>$s6</td>
</tr>
<tr>
<td>$r29</td>
<td>$r30</td>
<td>$s7</td>
</tr>
<tr>
<td>$r31</td>
<td>$s8</td>
<td></td>
</tr>
</tbody>
</table>

### General-purpose registers (GPRs)

- The name GPR implies that all these registers can be used as operands in instructions
- Still, conventions and limitations exist to keep GPRs from being used arbitrarily
  - $0, termed $zero, always has a value of “0”
  - $31, termed $ra (return address), is reserved for storing the return address for subroutine call/return
  - Register usage and related software conventions are typically summarized in “application binary interface” (ABI) – important when writing system software such as an assembler or a compiler
- **32 GPRs in MIPS**
  - Are they sufficient?

### Special-purpose registers

- HI/LO registers are used for storing result from multiplication operations
- PC (program counter)
  - Always keeps the pointer to the current program execution point; instruction fetching occurs at the address in PC
  - Not directly visible and manipulated by programmers in MIPS
- Other architectures
  - May not have HI/LO; use GPRs to store the result of multiplication
  - May allow storing to PC to make a jump

### Instruction encoding

- Instructions are encoded in binary numbers
  - Assembler translates assembly programs into binary numbers
  - Machine (processor) decodes binary numbers to figure out what the original instruction is
  - MIPS has a fixed, 32-bit instruction encoding
- Encoding should be done in a way that decoding is easy
- **MIPS instruction formats**
  - R-format: arithmetic instructions
  - I-format: data transfer/arithmetic/jump instructions
  - J-format: jump instruction format
  - (FI-/FR-format: floating-point instruction format)
### MIPS instruction formats

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Size</td>
<td>6 bits 5 bits 5 bits 5 bits 5 bits 6 bits</td>
<td>All MIPS instructions 32 bits</td>
</tr>
<tr>
<td>R-format</td>
<td>op rs rt rd shamt funct</td>
<td>Arithmetic/logic instruction format</td>
</tr>
<tr>
<td>I-format</td>
<td>op rs rt address/immediate</td>
<td>Data transfer, branch, immediate format</td>
</tr>
<tr>
<td>J-format</td>
<td>op target address</td>
<td>Jump instruction format</td>
</tr>
</tbody>
</table>

### Instruction encoding example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>000000</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>00000</td>
<td>100000</td>
<td>NA</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>000000</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>00000</td>
<td>100010</td>
<td>NA</td>
</tr>
<tr>
<td>addi</td>
<td>I</td>
<td>001000</td>
<td>reg</td>
<td>reg</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>constant</td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>100011</td>
<td>reg</td>
<td>reg</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>address offset</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>101011</td>
<td>reg</td>
<td>reg</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>address offset</td>
</tr>
</tbody>
</table>

### Logic instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>op rs rt rd shamt funct</td>
<td>Logic instruction format</td>
</tr>
</tbody>
</table>

- Bit-wise logic operations
  - `<op> <r_target> <r_source1> <r_source2>`

- Examples
  - `and $s3, $s2, $s1  # $s3 = $s2 & $s1`
  - `or $t3, $t2, $t1  # $t3 = $t2 | $t1`
  - `nor $s3, $t2, $s1  # $s3 = ~(($t2 | $s1))`
  - `xor $s3, $s2, $s1  # $s3 = $s2 ^ $s1`

### Dealing with immediate

- Many operations involve small “immediate” value
  - `a = a + 1`
  - `b = b - 4`
  - `c = d | 0x04`

- Some frequently used arithmetic/logic instruction shaves their “immediate” version
  - `addi $s3, $s2, 16  # $s3 = $s2 + 16`
  - `andi $s5, $s4, 0xfffe  # $s5 = $s4 & 1111111111111110b`
  - `ori $t4, $t4, 4  # $t4 = $t4 | 0000000000000100b`
  - `xor $t7, $s6, 16  # $t7 = $s6 ^ 000000000000010000b`

- There is no “subi”; why?
Handling long immediate number

- Sometimes we need a long immediate value, e.g., 32 bits
  - Do we need an instruction to load a 32-bit constant value to a register?
- MIPS requires that we use two instructions
  - lui $s3, 1010101001010101b
    
    ![Binary Representation of lui instruction]

    $$1010101001010101$$
    $$0000000000000000$$
  
  Then we fill the low-order 16 bits
  - ori $s3, $s3, 1100110000110011b
    
    ![Binary Representation of ori instruction]

    $$1010101001010101$$
    $$1100110000110011$$

Memory transfer instructions

- Also called memory access instructions
- Only two types of instructions
  - Load: move data from memory to register
    - e.g., lw $s5, 4($t6) # $s5 ← memory[$t6 + 4]
  - Store: move data from register to memory
    - e.g., sw $s7, 16($t3) # memory[$t3 + 16] ← $s7
- In MIPS (32-bit architecture) there are memory transfer instructions for
  - 32-bit word: “int” type in C
  - 16-bit half-word: “short” type in C
  - 8-bit byte: “char” type in C

Address calculation

- Memory address is specified with a (register, constant) pair
  - Register to keep the base address
  - Constant field to keep the “offset” from the base address
  - Address is, then, (register + offset)
  - The offset can be positive or negative
- MIPS uses this simple address calculation method; other architectures such as PowerPC and x86 support different methods

Machine code example

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

```assembly
swap:
    sll $t0, $a1, 2
    add $t1, $a0, $t0
    lw $t3, 0($t1)
    lw $t4, 4($t1)
    sw $t4, 0($t1)
    sw $t3, 4($t1)
    jr $ra
```
Memory view

- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A memory address is an index into the array

<table>
<thead>
<tr>
<th></th>
<th>BYTE #0</th>
<th>BYTE #1</th>
<th>BYTE #2</th>
<th>BYTE #3</th>
<th>BYTE #4</th>
<th>BYTE #5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit byte address
- $2^{32}$ bytes with byte addresses from 0 to $2^{32} - 1$
- $2^{10}$ words with byte addresses 0, 4, 8, ..., $2^{32} - 4$

Words are aligned
- 2 least significant bits (LSBs) of an address are 0s

Half words are aligned
- LSB of an address is 0

Addressing within a word
- Which byte appears first and which byte the last?
- Big-endian vs. little-endian

More on alignment

- A misaligned access
  - lw $s4, 3($t0)

How do we define a word at address?
- Data in byte 0, 1, 2, 3
  - If you meant this, use the address 0, not 3
- Data in byte 3, 4, 5, 6
  - If you meant this, it is indeed misaligned!
  - Certain hardware implementation may support this; usually not
  - If you still want to obtain a word starting from the address 3 – get a byte from address 3, a word from address 4 and manipulate the two data to get what you want

Alignment issue does not exist for byte access

```c
int main()
{
    int A[100];
    int i;
    int *ptr;
    for (i = 0; i < 100; i++) A[i] = i;
    printf("address of A[0] = %8x\n", &A[1]);
    printf("address of A[50] = %8x\n", &A[50]);
    ptr = &A[50];
    printf("address in ptr = %8x\n", ptr);
    printf("value pointed by ptr = %d\n", *ptr);
    ptr = (int*)((unsigned int)ptr + 1);
}
```

address of A[0] = bffff2b4
address of A[50] = bffff378
value pointed by ptr = 50
Segmentation fault
### Shift instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>op</td>
<td>NOT USED</td>
</tr>
</tbody>
</table>

- Bits change their positions inside a word
- `<op> <r_target> <r_source> <shift_amount>`

Examples
- `sll $s3, $s4, 4` # `$s3 <= $s4 << 4`
- `srl $s6, $s5, 6` # `$s6 <= $s5 >> 6`

- Shift amount can be in a register (in that case “shamt” is not used)
- Shirt right arithmetic (sra) keeps the sign of a number
  - `sra $s7, $s5, 4`

### Control

- Instruction that potentially changes the flow of program execution
- MIPS conditional branch instructions
  - `bne $s4, $s3, LABEL`
  - `beq $s4, $s3, LABEL`
- Example
  ```
  if (i == h) h = i+j;
  bne $s5, $s6, LABEL
  add $s6, $s6, $s4
  LABEL: ...
  ```

- MIPS unconditional branch instruction (i.e., jump)
  - `j LABEL`
- Example
  ```
  if (i == h) f=g+h;
  else f=g–h;
  bne $t6, $s7, ELSE
  add $s5, $s6, $s7
  j EXIT
  ELSE: sub $s5, $s6, $s7
  EXIT: ...
  ```

- Loops
  - “While” loops
    - Example on page 74
  - “For” loops
Control

- We have `beq` and `bne`; what about branch-if-less-than?
  - We have `slt`
    
    ```
    if (s1<s2) t0=1;
    else t0=0;
    ```
    ```
    slt $t0, $s1, $s2
    ```

- Can you make a “pseudo” instruction “`blt $s1, $s2, LABEL`”?

- Assembler needs a temporary register to do this
  - `at` is reserved for this purpose

Address in I-format

- Immediate address in an instruction is not a 32-bit value – it’s only 16-bit
  - The 16-bit immediate value is in signed, 2’s complement form

- Addressing in branch instructions
  - The 16-bit number in the instruction specifies the number of “instructions” to be skipped
  - Memory address is obtained by adding this number to the PC
  - Next address = PC + 4 + sign_extend(16-bit immediate << 2)

  - Example
    - `beq $s1, $s2, 100`

<table>
<thead>
<tr>
<th>Branch/Immediate</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16-bit immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>17</td>
<td>18</td>
<td>25</td>
</tr>
</tbody>
</table>

J-format

- The address of next instruction is obtained from PC and the immediate value
  - Next address = {PC[31:28],IMM[25:0],00}
  - Address boundaries of 256MB

  - Example
    - `j 10000`

<table>
<thead>
<tr>
<th>Jump</th>
<th>op</th>
<th>26-bit immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2500</td>
<td></td>
</tr>
</tbody>
</table>

MIPS addressing modes

- Immediate addressing (I-format)

  - Example
    - `beq $s1, $s2, 100`

- Register addressing (R-/I-format)

- Base addressing (load/store) – [register + offset]

- PC-relative addressing (beq/bne) [PC + 4 + offset]

- Pseudo-direct addressing (j) [concatenation w/ PC]
### MIPS instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>MIPS operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$t0-t15, $a0, $v0-$v1, $s0-$s7, $t0-$t7, $a0-$a3, $t0-$t9, $k0-$k1, $gp, $sp, $fp, $ra</td>
<td>32 locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $sat is reserved for the assembler to handle large constants.</td>
<td></td>
</tr>
<tr>
<td>2nd memory</td>
<td>Memory[4]...</td>
<td>Accessible only by data transfer instructions. MIPS uses 32-bit addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and saved registers, such as those saved on procedure calls.</td>
<td></td>
</tr>
</tbody>
</table>

#### MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $t0, $t1, $t2</td>
<td>$t0 = $t0 + $t1</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $t0, $t1, $t2</td>
<td>$t0 = $t1 - $t2</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>add $t0, $t1, 100</td>
<td>$t0 = $t0 + 100</td>
<td>Add to memory</td>
</tr>
<tr>
<td></td>
<td>load word</td>
<td>lw $t0, 0($s0)</td>
<td>$t0 = Memory[0] + $s0</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $t0, 0($s0)</td>
<td>Memory[0] = $t0 + $s0</td>
<td>Store from memory to register</td>
</tr>
<tr>
<td></td>
<td>load upper</td>
<td>lw $t0, 100</td>
<td>$t0 = Memory[100]</td>
<td>Load transient value</td>
</tr>
<tr>
<td></td>
<td>branch on equal</td>
<td>beq $t0, $t1, 25</td>
<td>$t0 == $t1 go to PC + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $t0, $t1, 25</td>
<td>$t0 != $t1 go to PC + 100</td>
<td>Not equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>bgt $t0, $t1, 63</td>
<td>$t0 &gt; $t1; set $t1 = 0</td>
<td>Compare greater than; set $t1 = 0</td>
</tr>
<tr>
<td></td>
<td>set less than</td>
<td>blt $t0, $t1, 63</td>
<td>$t0 &lt; $t1; set $t1 = 0</td>
<td>Compare less than; set $t1 = 0</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>jump register</td>
<td>j $t0</td>
<td>$t0 go to $t0</td>
<td>For unconditional jump</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal $t0</td>
<td>$t0 go to $t0 + 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>

### Register usage convention

<table>
<thead>
<tr>
<th>NAME</th>
<th>REG. NUMBER</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler usage</td>
</tr>
<tr>
<td>$v0~$v1</td>
<td>2~3</td>
<td>values for results and expres. eval.</td>
</tr>
<tr>
<td>$a0~$a3</td>
<td>4~7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0~$t7</td>
<td>8~15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0~$s7</td>
<td>16~23</td>
<td>temporaries, saved</td>
</tr>
<tr>
<td>$t8~$t9</td>
<td>24~25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$k0~$k1</td>
<td>26~27</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

### Stack and frame pointers

- **Stack pointer ($sp)**
  - Keeps the address to the top of the stack
  - $29 is reserved for this purpose
  - Stack grows from high address to low
  - Typical stack operations are push/pop
- **Procedure frame**
  - Contains saved registers and local variables
  - “Activation record"
- **Frame pointer ($fp)**
  - Points to the first word of a frame
  - Offers a stable reference pointer
  - $30 is reserved for this
  - Some compilers don’t use $fp

### “C” program down to numbers

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

```assembly
void swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```
Producing a binary

Assembler

- Expands macros
  - Macro is a sequence of operations conveniently defined by a user
  - A single macro can expand to many instructions
- Determines addresses and translates source into binary numbers
  - Start from a pre-defined address
  - Record in “symbol table” addresses of labels
  - Resolve branch targets and complete branch instructions’ encoding
  - Record instructions that need be fixed after linkage
- Packs everything in an object file
- “Two-pass assembler”
  - To handle forward references

Object file

- Header
  - Size and position of other pieces of the file
- Text segment
  - Machine codes
- Data segment
  - Binary representation of the data in the source
- Relocation information
  - Identifies instructions and data words that depend on absolute addresses
- Symbol table
  - Keeps addresses of global labels
  - Lists unresolved references
- Debugging information
  - Contains a concise description of the way in which the program was compiled

Important assembler directives

- Assembler directives guide the assembler to properly handle source code with certain considerations
- .text
  - Tells assembler that a “code” segment follows
- .data
  - Tells assembler that a “data” segment follows
- .align
  - Directs aligning following items
- .global
  - Tells to treat following symbols as global
- .asciiz
  - Tells to handle following input as a “string”
Code example

```
.text
.align 2
.globl main
main:
  subu $sp, $sp, 32

loop:
  lw $t6, 28($sp)
  la $a0, str
  lw $a1, 24($sp)
  jal printf
  jr $ra

.str:
  .asciiz "The sum from 0 … 100 is %d\n"
```

Macro example

```
.data
.int str:
.asciiz "%d"
.text
.macro print_int($arg)
  la $a0, int_str
  mov $a1, $arg
  jal printf
.end_macro

print_int($7)
```

Linker

- Argument passing
  - First 4 arguments are passed through $a0~$a3
  - More arguments are passed through stack
- Result passing
  - First 2 results are passed through $v0~$v1
  - More results can be passed through stack
- Stack manipulations can be tricky and error-prone
  - Application binary interface (ABI) has formal treatment of procedure calls and stack manipulations
- More will be discussed in labs.
High-level language

<table>
<thead>
<tr>
<th></th>
<th>High-level language</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>C, Fortran, Java, ...</td>
<td>-</td>
</tr>
<tr>
<td>+</td>
<td>High productivity</td>
<td>Low productivity</td>
</tr>
<tr>
<td></td>
<td>– Short description &amp; readability</td>
<td>– Long description &amp; low readability</td>
</tr>
<tr>
<td></td>
<td>Portability</td>
<td>Not portable</td>
</tr>
<tr>
<td>–</td>
<td>Limited optimization capability in certain cases</td>
<td>With proper knowledge and experiences, fully optimized codes can be written</td>
</tr>
</tbody>
</table>

Interpreter vs. compiler

<table>
<thead>
<tr>
<th></th>
<th>Interpreter</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept</td>
<td>Line-by-line translation and execution</td>
<td>Whole program translation and native execution</td>
</tr>
<tr>
<td>Example</td>
<td>Java, BASIC, ...</td>
<td>C, Fortran, ...</td>
</tr>
<tr>
<td>+</td>
<td>Interactive</td>
<td>High performance</td>
</tr>
<tr>
<td></td>
<td>Portable (e.g., Java Virtual Machine)</td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>Low performance</td>
<td>Binary not portable to other machines or generations</td>
</tr>
</tbody>
</table>

Compiler structure

- Compiler is a very complex software with a variety of functions and algorithms implemented inside it

- Front-end
  - Deals with high-level language constructs and translates them into more relevant tree-like or list-format internal representation (IR)
  - Symbols (e.g., a[i+8*j]) are still available

- Back-end
  - Back-end IR more or less correspond to machine instructions
  - With many compiling steps, IR items becomes machine instructions

Compiler front-end

- Scanning
  - Takes the input source program and chops the programs into recognizable “tokens”
  - Also known as “lexical analysis” and “LEX” tool is used

- Parsing
  - Takes the token stream, checks the syntax, and produces abstract syntax trees
  - “YACC” or “BISON” tools are used

- Semantic analysis
  - Takes the abstract syntax trees, performs type checking, and builds a symbol table

- IR generation
  - Similar to assembly language program, but assumes unlimited registers, etc.
Compiler back-end

- Local optimization
  - Optimizations within a basic block
  - Common sub-expression elimination (CSE), copy propagation, constant propagation, dead code elimination, ...
- Global optimization
  - Optimizations that deal with multiple basic blocks
- Loop optimizations
  - Loops are so important that many compiler and architecture optimizations target them
  - Induction variable removal, loop invariant removal, strength reduction, ...
- Register allocation
  - Try to keep as many variables in registers as possible
- Machine-dependent optimizations
  - Utilize any useful instructions provided by the target machine

Code & abstract syntax tree

while (save[i] == k)
    i+=1;

Internal representation

```
# comments are written like this--source code often included
# while (save[i] == k)
loop: LI R1, save    # loads the starting address of save into R1
    LW R2,1
    MULT R3, R2, 4 #Multiply R2 by 4
    ADD R4, R3, R1
    LW R5, 0(R4)   # load save[i]
    LW R6, k
    ENE R5, R6, 0   #whileloop
    # i += 1
    LW R6, 1
    ADD R7, R6, 1   #increment
    SW R7,1
    branch loop #next iteration
endwhileloop;
```

Control flow graph

1. LI R1, save
2. LW R2,1
3. SLL R3, R2, 2
4. ADD R4, R3, R1
5. LW R5, 0(R4)
6. LW R6, k
7. BEQ R5, R6, startwhileloop
8. LW R6,1
9. ADD R7, R5, R6
10. SW R7,1
Loop optimization

Register allocation

Compiler example: gcc
- gcc = GNU open-source C Compiler
- cpp: C pre-processor
  - Macro expansion (#define)
  - File expansion (#include)
  - Handles other directives (#if, #else, #pragma)
- cc1: C compiler (front-end & back-end)
  - Compiles your C program
- gas: assembler
  - Assembles compiler-generated assembly codes
- ld: linker
  - Links all the object files and library files to generate an executable

Compiler optimization
- Optimizations are a must
  - Code quality can be much improved; 2~10 times improvement is not uncommon
- Turning on optimizations may incur a significant compile time overhead
  - For a big, complex software project, compile time is an issue
  - Not that serious in many cases as compiling is a rare event compared to program execution
  - Considering the resulting code quality, you pay this fee
  - Now, computers are fast!
Wrapping up

- **MIPS ISA**
  - Typical RISC style architecture
  - 32 GPRs
  - 32-bit instruction format
  - 32-bit address space

- **Operations & operands**

- Arithmetic, logical, memory access, control instructions

- Compiler, interpreter, …