MIPS Architecture

- 32-bit processor, MIPS instruction size: 32 bits.
- Registers:
  1. 32 registers, notation `$0, $1, ···` $31 $0: always 0. $31: return address. $1, $26, $27, $28, $29 used by OS and assembler.
  2. Stack pointer ($sp or $29) and frame pointer ($fp or $30).
  3. 2 32-bit registers (HI and LO) that hold results of integer multiply and divide.
- Data formats:
  1. MIPS architecture addresses individual bytes ⇒ addresses of sequential words differ by 4.
  2. Alignment constraints: halfword accesses on even byte boundary, and word access aligned on byte boundary divisible by 4.
  3. Both Big Endian (SGI) and Little Endian (Dec). So be careful what you choose..
- Instruction set:
  1. Load/Store: move data between memory and general registers.
  2. Computational: Perform arithmetic, logical and shift operations on values in registers.
  3. Jump and Branch: Change control flow of a program.
  4. Others: coprocessor and special instructions.

  Supports only one memory addressing mode: `c(rx)`.

Assembly Programming

- Naming and Usage conventions applied by assembler. Use `#include <regdefs.h>` in order to use names for registers.
- Directives: pseudo opcodes used to influence assembler's behavior. You will need to generate these directives before various parts of generated code.
  1. Global data segments: data segments partitioned into initialized, uninitialized, and read-only data segments:
     - `.data`: Add all subsequence data to the `data` section. (No distinction about the segment in which data will be stored.)
     - `.rdata`: Add subsequent data in read-only data segment.
     - `.sdata`: Add subsequent data in uninitialized data segment.
     - `.sbss`: Add subsequent data in initialized data segment.
  2. Literals: Various kinds of literals can be added to various data segments through the following directives:
     - `.ascii str`: store string in memory. Use `.asciiz` to null terminate.
     - `.byte b1, ..., bn` assemble values (one byte) in successive locations. Similarly `.double`, `.float`, `.half`, `.word`.

Directives - cont'd.

- Code segments: A code segment is specified by the `.text` directive. It specifies that subsequent code should be added into text segment.
- Subroutines: The following directives are related to procedures:
  - `.ent procname`: sets beginning of `procname`.
  - `.end procname`: end of procedure.
  - `.global name`: Make the name external.
- `.align n`: align the next data on a 2^n boundary.

A typical assembly program

```
        .rdata
        .byte   0x24,0x52,0x65
        .align  2
$LC0:
        .word   0x61,0x73,0x74,0x72
        .text
        .align  2
        .globl  main
        .ent    main
main:
        .frame  $fp,32,$31
        subu    $sp,$sp,32
$L1:
        j       $31
        .end    main
```

Data Transfer Instructions

- Load instruction: `lw rt, offset(base)`. The 16-bit offset is sign-extended and added to contents of general register `base`. The contents of word at the memory specified by the effective address are loaded in register `rt`.
  Example: Say array `A` starts at `Astart` in heap. `g`, `h`, `i` stored in `$17`, `$18`, `$19`.
  Java^{--} code:
  ```
  g = h + A[i];
  ```
  Equivalent assembly code:
  ```
  lw   $8, Astart($19)  # $8 gets A[i]
  add  $17, $18, $8     # $17 contains h + A[i]
  ```
- Store instruction: `sw rt, offset(base)`
  Example: *Java*$^{--}$ code:
  ```
  A[i] = h + A[i];
  ```
  Equivalent assembly code:
  ```
  lw   $8, Astart($19)  # $8 gets A[i]
  add  $8, $18, $8      # $8 contains h + A[i]
  sw   $8, Astart($19)  # store back to A[i]
  ```
- MIPS has instructions for loading/storing bytes, halfwords as well.

Computational Instructions

- Perform Arithmetic, logical and shift operations on values in registers.
- Four kinds:
  1. ALU Immediate:
     - 1.1 Add immediate: `addi rt, rt, immediate`
     - 1.2 And immediate: `andi rt, rt, immediate`
  2. 3-operand Register type instruction
     - 2.1 Add: `add rd, rs, rt`
     - 2.2 Subtract: `sub rd, rs, rt`
     - 2.3 AND, OR etc.
  3. Shift instructions:
     - 3.1 Shift Left logical: `sll rd, rt, shamt`
     - 3.2 Shift Right Arithmetic: `sra rd, rt, shamt`
  4. Multiply/Divide instructions:
     - 4.1 Multiply: `mult rs, rt`
     - 4.2 Divide: `div rs, rt`
     - 4.3 Move from HI: `mfhi rd`
     - 4.4 Move from LO: `mflo rd`

Decision Making Instructions

- `beq`: similar to an *if* with *goto*

  `beq register1, register2, L1`

  Example: *Java*$^{--}$ code:

  ```
  if ( i != j) f = g + h;
  f = f - i;
  ```

  Assume: f, g, h, i, j in registers \$16 through \$20.
  Equivalent Assembly code:

  ```
      beq  $19, $20, L1     # L1 is a label
      add  $16, $17, $18    # $16 contains f + h
  L1: sub  $16, $16, $19    # f := f-1
  ```

- `bne`: `bne register1, register2, L1`. Jump to L1 if register1 and register2 are not equal.
  Example: *Java*$^{--}$ code

  ```
  if ( i = j) f = g + h;
  else f = g - h;
  ```

  Assume: f, g, h, i, j in registers \$16 through \$20.

  ```
        bne  $19, $20, Else   # L1 is a label
        add  $16, $17, $18    # $16 contains g + h
        j    Exit             # skip else part
  Else: sub  $16, $17, $18    # f := g - h
  Exit:
  ```

  instruction `j`: unconditional jump.

- Note that addresses for labels generated by assembler.

Instructions -cont'd.

- Using conditional and unconditional jumps to implement loops:

```
while (save[i] == k) {
   i = i + j;
}
```

  Assume a) i, j, k in registers $19 through $21; b) Sstart contains the address for beginning of `save`; c) $10 contains 4.

```
Loop: mult $9, $19, $10     # $9 = i * 4
      lw   $8, Sstart($9)   # $8 = save[i]
      bne  $8, $21, Exit    # jump out of loop
      add  $19, $19, $20
      j    Loop
Exit:
```

- Compare two registers: `slt`

  `slt $8, $19, $20`: compare $8 and $9 and set $20 to 1 if the first register is less than the second.

  An instruction called `blt`: branch on less than. Not implemented in the machine. Implemented by assembler. Used register `$1` for it. So DO NOT use `$1` for your code generation.

Branch Instructions - cont'd.

- `jr`: jump to an address specified in a register. Useful for implementing case statement.

  Example:

```
switch(k) {
   case 0:  f = i + j; break;
   case 1:  f = g + h; break;
   case 2:  f = g - h; break;
   case 3:  f = i -j; break;
}
```

  Assumption: JumpTable contains addresses corresponding to labels `L0`, `L1`, `L2`, and `L3`.

  `f`, `g`, `h`, `i`, `j`: in registers $16 through $20. $21 contains value 4.

```
Loop: mult $9, $19, $21      # $9 = k * 4
      lw   $8, JumpTable($9)# $8 = JumpTable[k]
      jr   $8                # jump based on $8
L0:   add  $16, $19, $20     # k = 0
      j    exit
L1:   add  $16, $17, $18     # k = 1
      j    exit
L2:   sub  $16, $17, $18     # k = 2
      j    exit
L3:   sub  $16, $19, $20     # k = 3
Exit:
```

Procedures

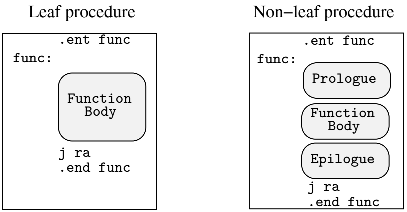
- `jal`: jump to an address and simultaneously save the address of the following instruction (return address) in $31: `jal ProcedureAddress`
- Assume A calls B which calls C
  - A is about to call B:
    1. Save A's return address (in $31) on stack
    2. Jump to B (using jal)
    3. $31 contains return address for B.
  - B is about to call C:
    1. Save B's return address (in $31) on stack
    2. Jump to C (using jal)
    3. $31 contains return address for C.
  - Return from C: jump to address in $31
  - On returning from B: restore B's return address by loading $31 from stack.
- MIPS assembly code:

```
A:  :
    jal B
    :
B:  :
    add $29, $29, $24
    sw  $31, 0($29)   # save return address
    jal C             # call C + save ret addr in $31
    lw  $31, 0($29)   # restore B's return address
    sub $29, $29, $24 # adjust stack
    jr  $31
C:  :
    jr  $31
    :
```

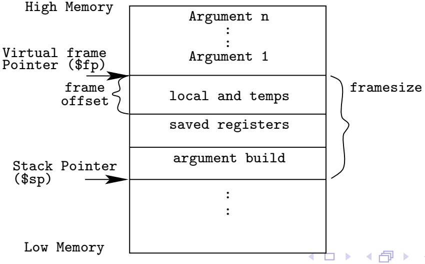
Procedures

- Two kinds of routines:
  1. Leaf: do not call any other procedures
  2. Non-leaf: call other routines

  Determine type of your routine
- How does the generated procedure look?



- How does stack frame look?



Parameter Passing

- General registers \$4 – \$7 and floating point registers \$f12 and \$f14 used for passing first four arguments (if possible).
- A possible assignment:

Arguments	Register Assignments
(f1,f2,...)	f1→\$f12, f2→\$f14
(f1,n1,f2, ...)	f1→\$f12 n1→ \$6,f2→stack
(f1,n1,n2, ...)	f1→\$f12, n1→ \$6,n2→\$f7
(n1,n2,n3,n4,...)	n1→\$f4, n2→ \$5, n3→\$f6, n4→\$f7
(n1,n2,n3,f ...)	n1→\$f4, n2→ \$5, n3→\$f6, f1→stack
(n1,n2,f1)	n1→\$f4, n2→ \$5, f1→(\$6, \$7)

Prologue for procedure

- Define an entry for procedure first

```
        .ent proc
    proc:
```

- Allocate stack space:

```
        subu $sp, framesize
```

  `framesize`: size of frame required. Depends on
  - local variables and temporaries
  - general registers: \$31, all registers that you use.
  - floating point registers if you use them
  - control link

Prologue for procedure - cont'd.

- Include a `.frame` psuedo-op:

```
    .frame  framereg, framesize, returnreg
```

  Creates a virtual frame pointer ($fp): $sp + framesize

- Save the registers you allocated space for

```
    .mask   bitmask, frameoffset
    sw      reg, framesize+frameoffset-N($sp)
```

  `.mask`: used to specify registers to be stored and where they are stored.

  One bit in bitmask for each register saved.

  `frameoffset`: offset from virtual frame pointer. Negative.

  N should be 0 for the highest numbered register saved and then incremented by 4 for each lowered numbered register:

```
sw  $31, framesize+frameoffset($sp)
sw  $17, framesize+frameoffset-4($sp)
sw  $6, framesize+frameoffset-8($sp)
```

- Save any floating point register:

```
.fmask   bitmask, frameoffset
s.[sd]   reg,framesize+frameoffset-N($sp)
```

  Use .fmask for saving register

- Save control link (frame pointer) information
- Save access link/display information (if any).

Epilogue of a procedure

- Restore registers saved in the previous step

```
lw    reg, framesize+frameoffset($sp)
```

- Restore floating point registers
- Restore control link information
- Restore access link/display information
- Get return address

```
lw    $31, framesize+frameoffset($sp)
```

- Clean up stack:

```
addu  $sp, framesize
```

- Return:

```
j     $31
```

Example Pascal and MIPS assembly program

Pascal Program — Equivalent Assembly

```
Program test;
  procedure p(x: integer);
  begin
  end
begin
  p(1);
end.
```

```
        .text
        .align  2
        .globl  main
        .ent    main
main:
        subu    $sp, 24
        sw      $31, 20($sp)
        .mask   0x80000000, -4
        .frame  $sp, 24, $31
        li      $4, 1
        addu    $2, $sp, 24
        jal     p
        move    $2, $0
        lw      $31, 20($sp)
        addu    $sp, 24
        j       $31
        .end    main

        .text
        .align  2
        .ent    p
p:
        subu    $sp, 8
        sw      $4, 8($sp)
```