32-bit MIPS ISA

Introduction

- Four backward-compatible families: MIPS I, II, III, IV
  - We will focus on MIPS I only (32-bit ISA); what is this?
  - MIPS III onwards are fully 64-bit ISAs (in textbook)
- Load/store register ISA
  - Operates only on registers
  - Can access memory only through load/store (big endian)
  - No partial register access
- RISC philosophy
  - Emphasis on efficient implementation: Make the common case fast
  - Simplicity: provide primitives, not the solutions
    - A system can be so simple that it obviously has no bugs, or so complex that it has no obvious bugs [C. A. R. Hoare]

Data types

- Bit strings
  - Byte is 8 bits
  - Half word is 16 bits
  - Word is 32 bits
  - Double word is 64 bits
- Integers in 2's complement
- Floating-point: single and double precision
  - IEEE 754 standard

Storage model

- 32-bit address: 4 GB addressable memory
- Separate 31x32-bit GPRs ($0 is hardwired 0) for integer and 32x32-bit GPRs for floating-point
  - Writing to integer $0 will not change it (these are NOPs)
- Program Counter (PC) is incremented by 4 (except branch, jump)
  - Instructions are 32-bit in size (for all four families)
- Two special registers Hi and Lo for storing multiply/divide results
- Floating-point registers are paired for doing double-precision
  - The pair $f_2n and $f_2n+1 are accessed by name $f_2n e.g. $f2 specifies the 64 bits in $f2 and $f3 with the least significant word in $f2

single: add.s $f0, $f2, $f4 :

f_2 + f_4 = f_0

double : add.d $f0, $f2, $f4

must be even for (.d) commands

f_3 f_2 + f_5 f_4 = f_1 f_0

Computation

- ALU instructions
  - Classic 3-operand format: two sources and one dest.
  - Operands: GPRs or 16-bit immediate values
  - Both signed and unsigned arithmetic
  - Basic difference between signed and unsigned arithmetic: overflow not flagged for unsigned
    - In both cases, the operands are treated as signed
  - Sign extension of immediate in both signed and unsigned arithmetic; zero extension for immediates in logical inst.
  - Signed comparison is completely different from unsigned comparison (unsigned comparison treats the operands as unsigned; not true for unsigned arithmetic)
  - Integer multiply/divide take only two operands i.e. two sources and have implicit targets Hi and Lo: ISA offers instructions to move from Hi/Lo registers to GPR

ALU instructions

- Arithmetic

```
add $3, $2, $1
sub $3, $2, $1
addi $3, $2, 100
addu $3, $2, $1
subu $3, $2, $1
addiu $3, $2, 100
slt $3, $2, $1
slti $3, $2, 100
sltu $3, $2, $1
sltiu $3, $2, 100
mult/multu $3, $2
div/divu $3, $2  // Lo=q, Hi=r
mfhi $4
mflo $4
```

- Logical

```
and $3, $2, $1
or $3, $2, $1
xor $3, $2, $1
nor $3, $2, $1
andi $3, $2, 10
ori $3, $2, 10
xori $3, $2, 10
sll $3, $2, 10
srl $3, $2, 10
sra $3, $2, 10
sllv $3, $2, $1
srlv $3, $2, $1
srav $3, $2, $1
lui $3, 40
```


Floating point

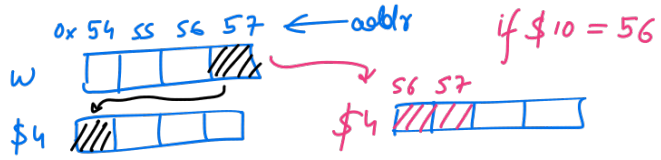
- Operates on floating-point registers
  - Supports both single and double precisions
  - No hardwired zero register
  - IEEE 754 compliant
  - Typical instructions
    - add, sub, mul, div, mov (one fp reg to another), neg, abs, cvt (precision conversion), mfc, mtc (move between fp and integer registers)
    - mul/div don't use Hi/Lo; target FPR is specified explicitly (remainder doesn't have a meaning here)

Load/store

- One address mode for memory
  - Displacement only (always sign-extended)
  - Most loads/stores are aligned (except lwl/lwr, swl/swr)
  - Loads are supported for signed and unsigned data; unsigned loads zero-extend the loaded value
  - Supports three sizes: byte, half word, word (double word is supported in 64-bit ISA)
  - Byte load: lb $3, 4($13) or lbu $3, 27($20)
  - Half word load: lh $3, 12($10) or lhu $6, 0($7)
  - Word load: lw $20, 52($29) [no unsigned flavor]
  - Byte store: sb $3, 5($2)
  - Half word store: sh $3, 56($29)
  - Word store: sw $2, 20($2)
  - The immediate is *not* shifted by load/store size; it is just sign-extended and added to the base register content
  - In addition, floating-point load/store: lf $f1, 60($22)

lwl

$10 = 57



Direct addressing

- In some cases the address is known at compile-time
  - Happens mostly for statically allocated global variables
  - How do you emulate direct addressing?
  - Suppose I want to load from address 0x123456 (addr)

```
lui $2, 0x12
ori $2, $2, 0x3456
lw $4, 0($2)
```

li $2, addr

  - Could save one instruction by using non-zero displacement

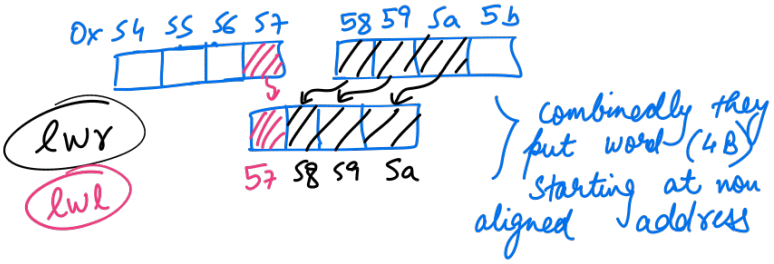
```
lui $2, 0x12
lw $4, 0x3456($2)
```

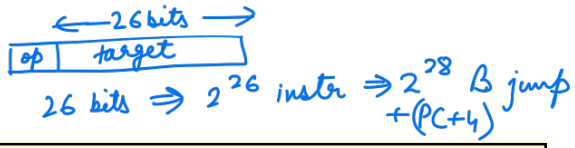
  - What if the address is 0x789abc?

78 0000 + ffff 9a bc ≠ 789abc

lwl/lwr

- Load-word-left and load-word-right
  - Example: lwl $4, 0($10) // Suppose $10 has 0x57
    - Let the word containing this byte address be w
    - Extract the bytes contained by w that *start* from this address (remember this is big endian)
    - Put these bytes (in this case one byte) in the *upper* portion of $4 and leave the remaining bytes (in this case 3 lower bytes) unchanged
  - Example: lwr $4, 0($10) // Suppose $10 has 0x5a
    - Extract the bytes contained by w that *end* at this address
    - Put these bytes (in this case three bytes) in the *lower* portion of $4 and leave the remaining bytes (in this case the upper byte) unchanged
  - Why are these useful?





Control transfer

- Jump
  - Unconditional jumps and procedure calls use absolute address
  - MIPS ISA offers 26 bits to encode the absolute target; shift this target to left by 2 bits (4-byte instruction) and borrow the upper four bits of the next PC (i.e. PC+4) to form the complete 32-bit target
    ((PC+4) & 0xf0000000) | tgt << 2

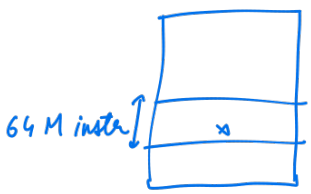
not PC but PC+4

  - The procedure call instruction (known as jump and link) saves the return PC (PC+8) in a fixed GPR ($31)
  - Indirect jumps where target is not known (e.g. procedure return or case/switch or procedure call via function pointer) use jump register (jr) or jump and link register (jalr) instructions; both take a register operand where the target is found
    - Example: procedure return: jr $31

PC → jmp f()
PC+4 (ignore)
PC+8 saved in $31

Control transfer

- Conditional branches
  - All conditional branches are compare-and-jump type
  - PC-relative immediate offsets are always sign-extended
  - Examples:
    - beq $1, $2, 100 /* target = PC + 4 + sext(100 << 2) */
    - bne $1, $2, 100
    - bgez $1, 100
    - blez $1, 100
    - bltz $1, 100
    - bgtz $1, 100
  - Can use slt, slti, sltiu first followed by beq or bne with $0
- All branch and jump instructions have a delay slot
  - Instruction right after the jump or branch is always executed



can't cross the 64 M instr (PC+4) boundary. Can roam around only in this. (Upper 4 bits of PC+4 only)

PC+8 : because the j, jal, jalr, jr etc all come into effect one cycle later. So (PC+4) has already been executed. So we come back to PC+8. That PC+4 is always a Branch Delay Slot

PC+4: because instr after PC has already in execution. So we find dist from PC+4.

instr after branch will always execute before branch takes effect

Control transfer

```
count = 0;                          addiu $5, $0, 0
for (i=head; i<=n; i++) {           addu $6, $1, $0
  if (A[i]==key) count++;   loop:   slt $9, $4, $6
}                                   bne $9, $0, exit
                                    sll $7, $6, 2
Assume following allocation         addu $7, $7, $2
   count in $5                      lw $8, 0($7)
   i in $6                          bne $8, $3, next
   head in $1                       addiu $6, $6, 1
   start addr of A in $2            addiu $5, $5, 1
   key in $3                next:   j loop
   n in $4                          nop
   A is an integer array    exit:   …
```

will always execute as after branch. So we do i++ here. ctr++ may not be done.

could not fill anything so nop.

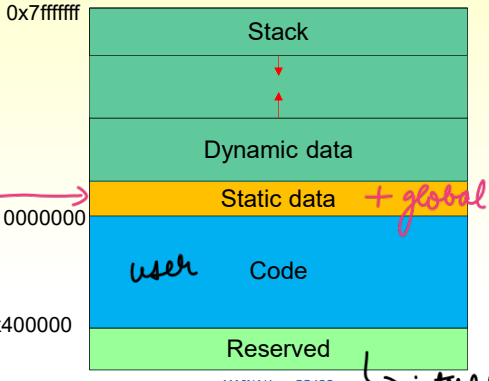
Procedure call

- MIPS calling convention
  - Parameters are passed in registers; extra parameters (more than 4) are spilled on stack
  - Compiler tries best to allocate local scalar variables in registers; stack is used for spilling
  - Two dedicated GPRs point to top of stack (stack pointer) and start of procedure frame (frame pointer)
- Global pointer
  - Global static scalar variables are allocated in a 64 KB static area at compile time
  - Accessed with $gp + offset (these are determined at link time)

non array

Stack grows downwards

MIPS memory map



global ptr at mid point

+ global data

user

interrupt Vectors

MIPS memory map

- User gets 2 GB
  - Upper 2 GB is kernel space
  - On processor reset execution starts at 0xbfc00000
  - Stack grows downward while heap grows upward

(Kernel space)

lui + ori = la = li (needs temp reg)

Register convention

- 32 integer registers
  - $0 is hardwired to 0 (really not implemented as register)
  - $1 or $at is reserved for assembler (la to lui conversion)
  - $2, $3 (or $v0, $v1) are return values of function with upper 32 bits in $v1 and lower 32 bits in $v0; $v1 also holds the syscall number before executing system call
  - $4 to $7 (or $a0 to $a3) are procedure arguments; saved by caller
  - $8 to $15 (or $t0 to $t7) are temporaries; saved by caller
  - $16 to $23 (or $s0 to $s7) are callee saved
  - $24 and $25 (or $t8, $t9) are two more temporaries
  - $26 and $27 (or $k0, $k1) are reserved for kernel
  - $28: global pointer ($gp), $29: stack pointer ($sp), $30: frame pointer ($fp), $31: return address ($ra)

$1 = $at = $ assembler temporary.
$26, $27 → return from exceptions

Register saving

- Caller saves the registers that are not preserved across call
  - Of course, needed only if caller wants some value to be preserved
  - $a0 to $a3, $v0, $v1, $t0 to $t9
- Callee saves the registers that are *required* to be preserved across call
  - Needed only if callee uses these registers
  - $s0 to $s7, $gp, $sp, $fp, $ra
- MIPS gcc cross-compiler and native cc compiler do not use frame pointer
  - $30 is treated as callee-saved $s8

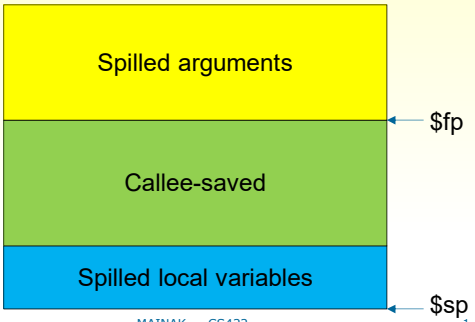
if only 1 arg then save only $a0 and then put arg. Rest not reg to be saved.
arg
ret val

$ra — can be both callee or caller saved. depends on implementation.
→ for main: it has to be callee saved before calling any other func.

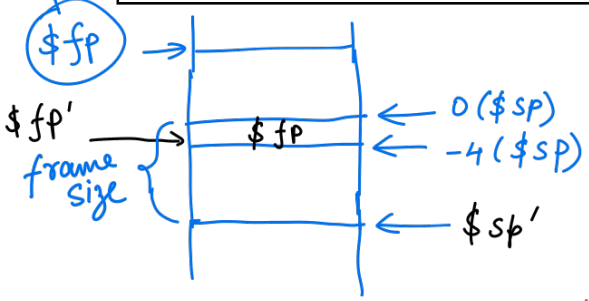
Procedure call

- What does the caller do?
  - Save caller-saved registers if needed
  - Load arguments: first four in $a0 to $a3, rest on stack
  - jal or jalr
- What does the callee do? (will use frame pointer)
  - Save frame pointer at -4($sp)
  - Allocate stack for frame: $sp = $sp – frame size (compiler knows the frame size for this procedure)
  - Save callee-saved registers if needed
  - Adjust frame pointer to point to the beginning of the frame: $fp = $sp + frame size – 4
- What happens on return?
  - Callee places return value in $v0 (and $v1 if 64-bit value)
  - Restore any callee-saved register from stack
  - Pop stack frame: $sp = $sp + frame size; jr $ra

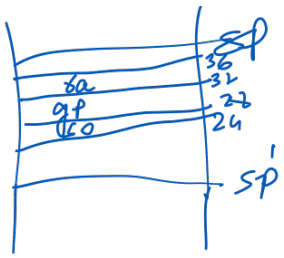
Procedure stack

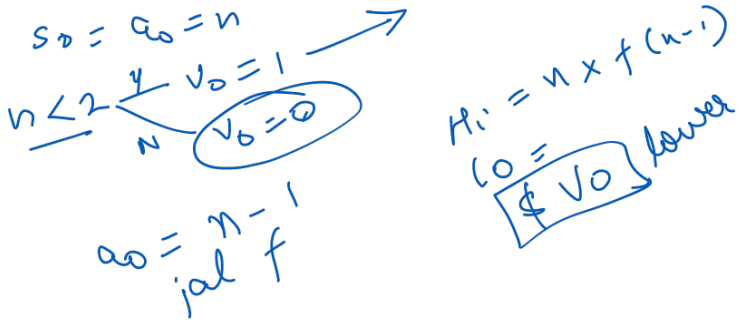


of caller



$sp = TOS
$fp = BOS





MIPS procedure call

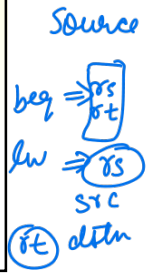
```
int factorial (int n) {
  if (n <= 1) return 1;
  else
    return (n*factorial (n-1));
}

factorial: addiu $sp, $sp, -40
           sw $s0, 24($sp)
           addu $s0, $a0, $0
           slti $v0, $s0, 2
           sw $ra, 32($sp)
           bnez $v0, label1
           sw $gp, 28($sp)
           addiu $a0, $s0, -1
           jal factorial
           nop
           mult $s0, $v0
           mflo $v0
           j label2
           nop
label1:addiu $v0, $0, 1
label2:lw $ra, 32($sp)
           lw $s0, 24($sp)
           jr $ra
           addiu $sp, $sp, 40
```

MAINAK CS422 1

MIPS encoding

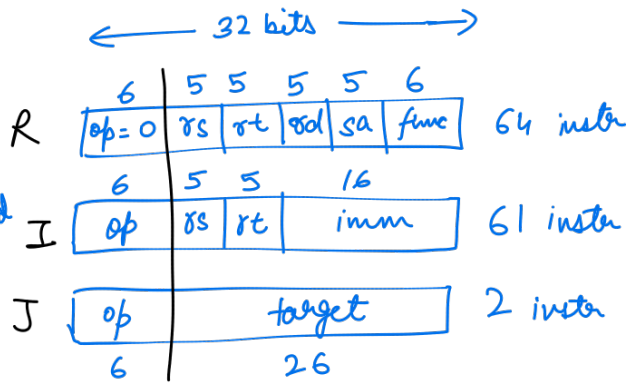
- Three formats
  - All 32 bits; opcode 6 bits
  - Register format or ALU format
    - opcode = 0, rs, rt, rd, sa (5 bits each), func (6 bits)
  - Immediate format
    - Arithmetic, branch, load/store, … (anything that uses an immediate value)
    - Different opcode for each instruction
    - opcode, rs, rt, immediate
    - rs is the first source or base, rt is the second source or result
  - Jump format
    - Used for j, jal
    - opcode, target (can jump across 64 M instructions)

MAINAK CS422 1



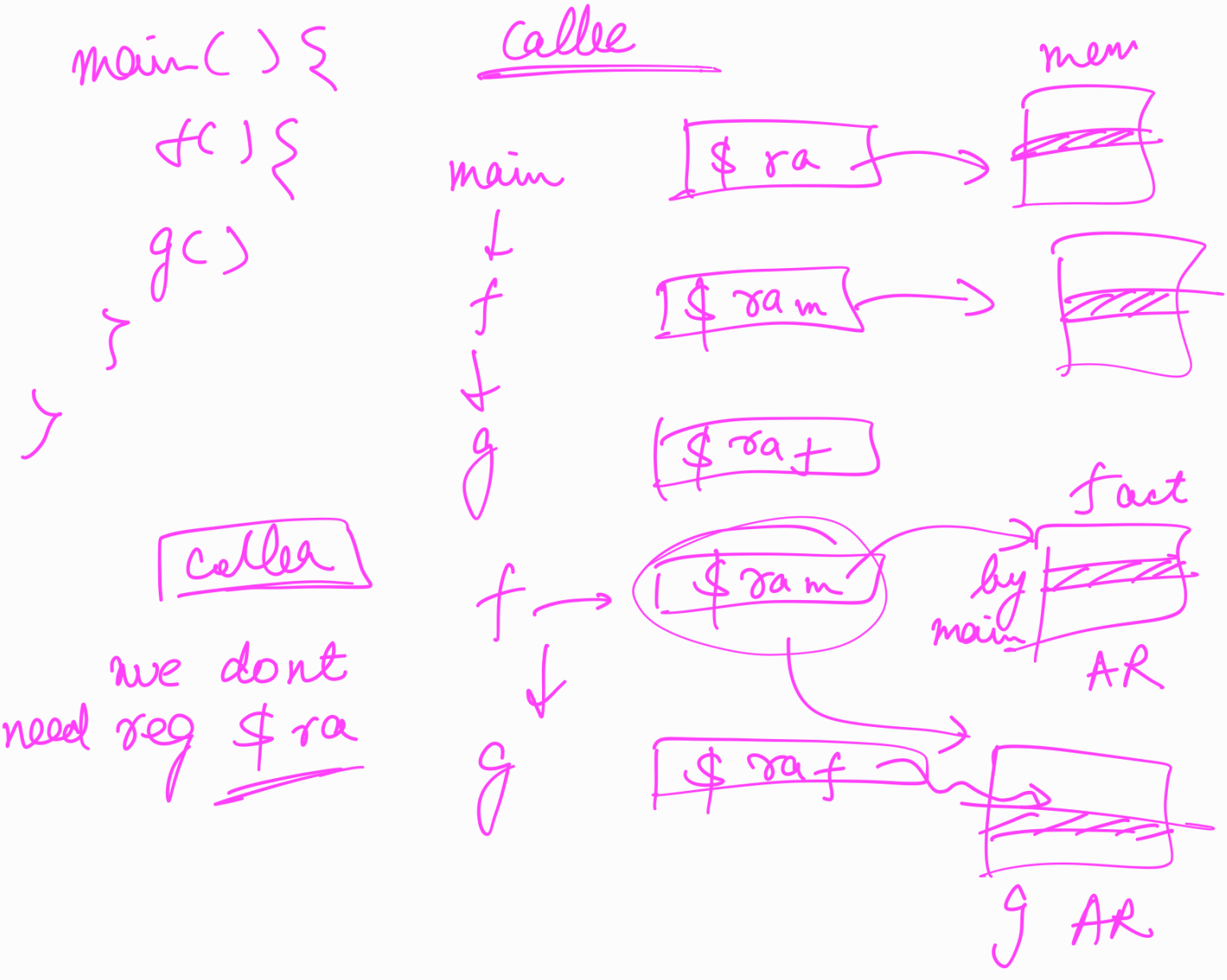
ret type = int
so mflo enough (32 bits)

rs, rt = src regs
rd = dest reg
sa = shift amt
func = 64 types of instr (not all fields may be used by all instrs)
sll $3, $2, 6



```
main(){
    f(){
        g()
    }
}
```

Callee



Caller

we dont need reg $ra