Chapter 2:

Instruction Set Architecture (ISA)

specifically : MIPS ISA

Add Instruction

Asm notation : `add a, b, c # put c+b into a`

C Example:

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

Mips equivalent:

```
add t0, i, j
add t1, g, h
sub f, t1, t0
```

32 bit

Example with MiPs Registers

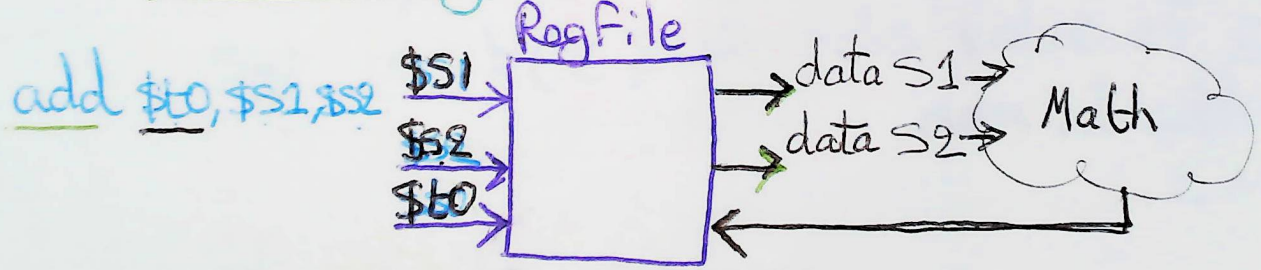
C code: `F = (g+h)-(i+j);`

MiPs:

```
add $t0, $s1, $s2      (i, j)
add $t1, $s3, $s4      (g, h)
sub $s0, $t0, $t1      (F)
```

Register File

- small storage of local variables



Memory operations

- Load from memory address to Reg File.
- store Reg to some address in Memory

MIPS operations for Mem access:

```
lw  a, b(c)   # takes "word" value, from memory at address: c+bx4
```

(mem → a; 32 bits; dest reg; b → reg)

Array[x] ←

c : Array

b : x

```
sw  a, b(c)   # puts value of a into mem at address: c+bx4
```

word : 32 bits → 4 bytes

byte: 8 bits

offset: in words

more complete example:

C code: F = (g+h) - (i+j);

Mips assembly:

assumption: $S5 is base of variable space.

loading
```
lw  $S1, 1($S5)  ->i
lw  $S2, 2($S5)  ->j
lw  $S3, 3($S5)  ->g
lw  $S4, 4($S5)  ->h
```
read from mem to reg

math
```
add $t0, $S1, $S2
add $t1, $S3, $S4
~~sub $S0, $t0, $t1~~
sub $S0, $t1, $t0
```

storing
```
sw  $S0, 0($S5)
```
write from reg mem.

-> f

Example mem operation to array

C code: `g = h + A[8];`

assump:
- g is in $S1
- h is in $S2
- base of array 'A' is in $S3

Mips asm:

8 x 4

```
lw  $t0, 32($s3)
    imm  reg

add $s1, $s2, $t0
    reg  reg  reg
```

opcode

why have registers?

- Registers are faster to access (because its small)
- Known capacity, so register indexing takes fixed number of bits.
- Register file (or physical place registers are stored) is part of ISA.
- MIPS: 32 word-sized registers

5 bit
add → 3 x 5 = 15 bit

Example Nested procedure calls

C:

```
int fact(int n) {
  if(n<1) return 1;
  else return (n x fact(n-1));
}
```

assump:
n in $a0
result in $v0

assembly:

fact:

stack push

```
addi $sp, $sp, -8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
```

2 DWs

$ra
$a0 ↓ -8 = 2 x 4 DW
$sp

set less than (imm) ←

```
slti $t0, $a0, 1
beq  $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, +8
jr   $ra
```

L1:

```
addi $a0, $a0, -1
jal  fact
```

stack pop

$sp

```
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, +8
mul $v0, $a0, $v0
jr $ra
```

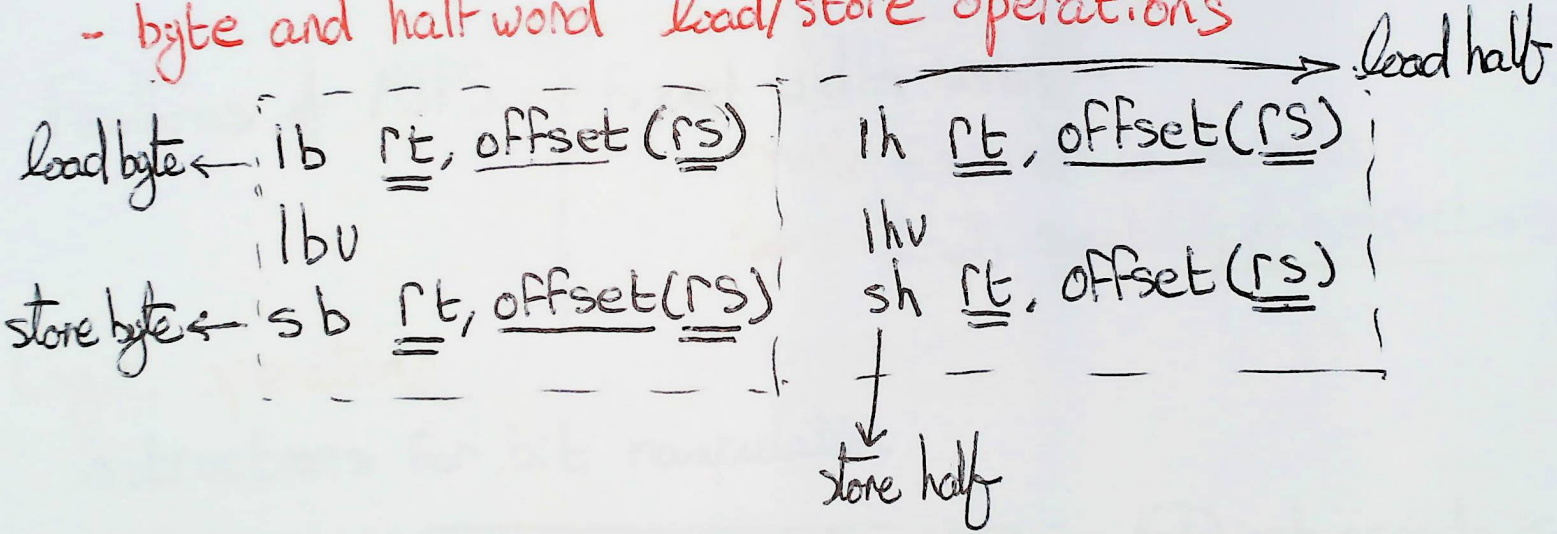
character data

need protocols when communicating "written data".

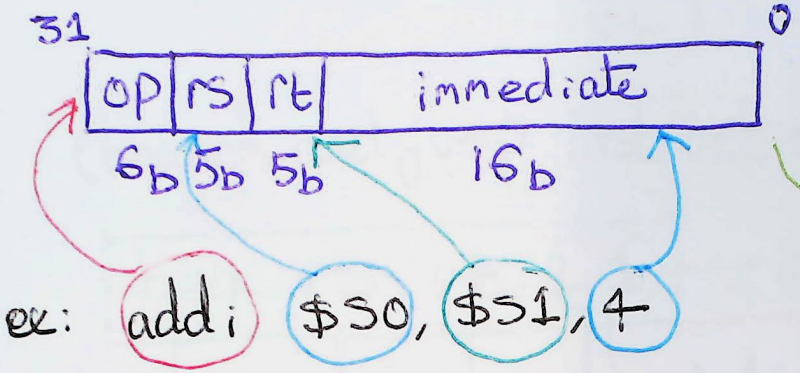
e.g. ASCII — 8 bit (128 char)

unicode — 32-bit (4 gig word, emoji, ...)

- byte and halfword load/store operations



MIPS I-Format Instruction



make common case faster
[most immediate instr. use small values]

Features of MIPS: { Fixed width inst:
pros: Easier to decode
cons: limits capability of instructions

- Logic operations
  instructions for bit manipulation

op	C	MIPS opcode
shift L	<<	sll
shift R	>>	srl
and, or not, xor...	&, |, ~	and, or not, xor

→ not inst opcode is unavailable.

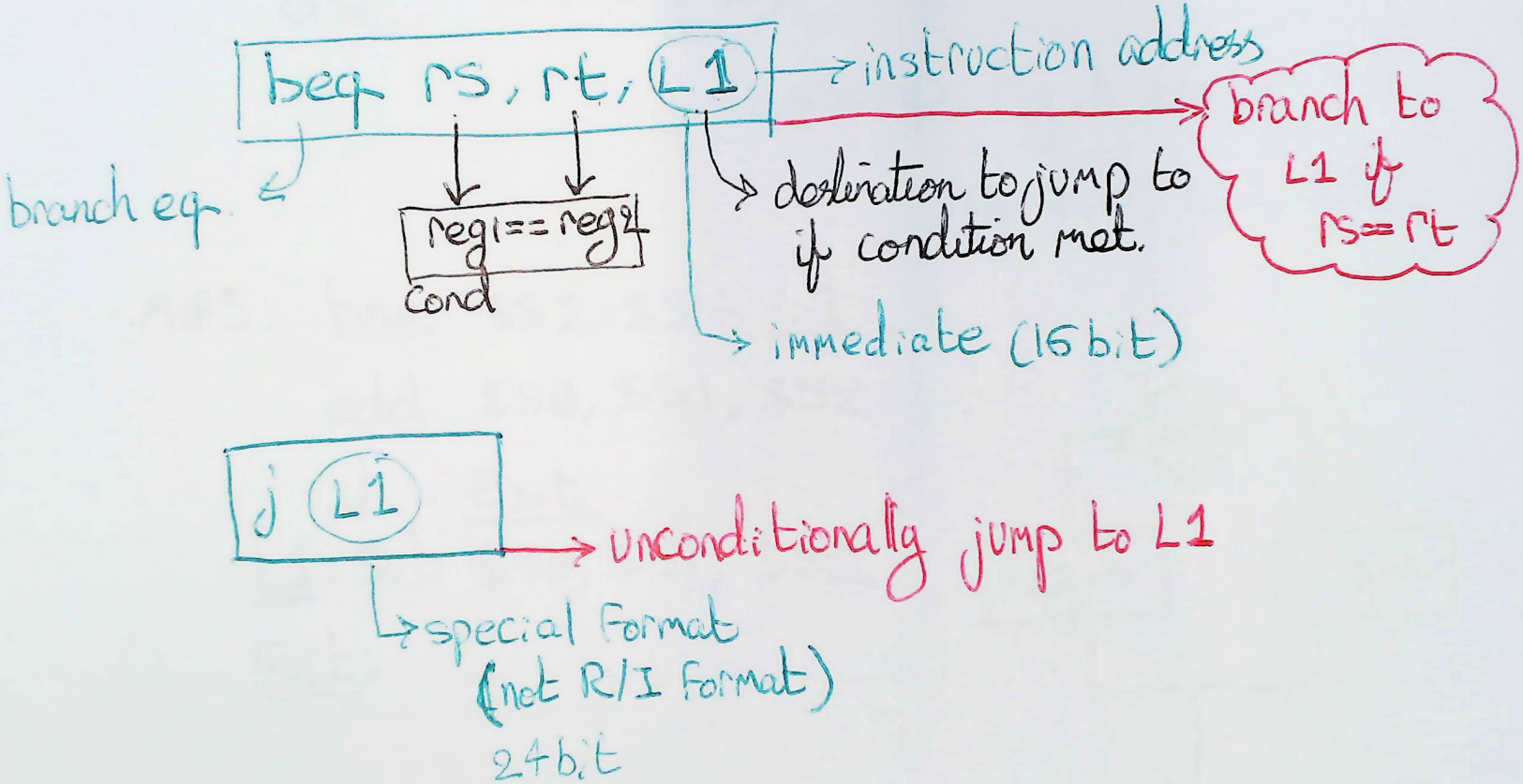
not: nor $s0, $s1, $zero

not $s0, $s1

why waste an opcode.

Conditional operations (Control-Flow)

(branch and jump instructions)

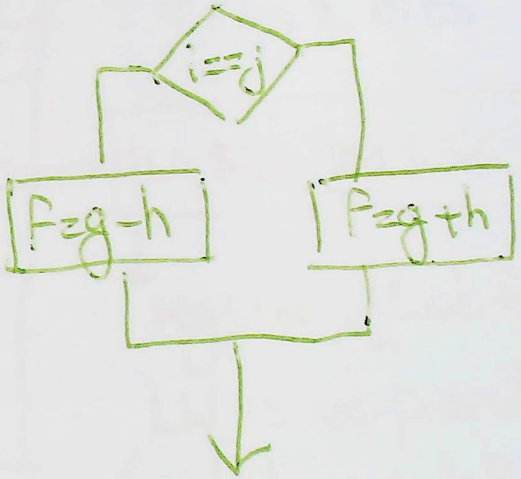


Example Branch

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

assump: f, g, h, ... are in reg $S0, $S1, $S2, ...

MIPS:
```
    bne  $S3, $S4, L1
    add  $S0, $S1, $S2
    j    Exit
L1: sub  $S0, $S1, $S2
Exit: ...
```



Loop Example

C: `while(save[i] == k) i++;`

assump: i is in $s3

k is in $s5

base of save is $s6

(save is of word size)

shift left

```
Mips: Loop: sll  $t1, $s3, 2
            add  $t1, $t1, $s6
            lw   $t0, 0($t1)
            bne  $t0, $s5, Ex
            addi $s3, $s3, 1
            j    Loop
Ex: ...
```

sll → shift $s3 left by 2 bits

lw → lw $t0, $s3($s6)

↓ register offset for Lw is not provided (would have cost 2 instructions)

Basic Blocks

a region of code with:

a) no branch/jump enters from outside

b) no branch/jump exits the region.

ISAs tend to not have/allow arbitrary destination jump/branch instruction. this to have clear basic block boundries.

2's complement → most significant bit

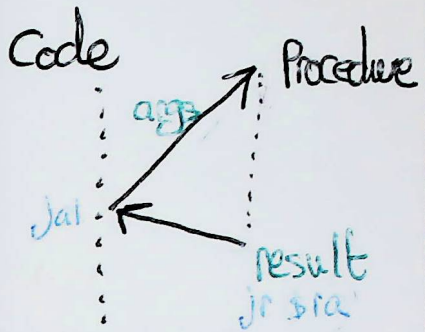
sign: determined by MSB → 1: negative
0: positive

to make negative:
flip the bit
add 1

ex: (1)1101 → neg.
00010 flip
00011 +1
-3

procedure calls:

what steps involved in procedure call?



1. place parameters of procedure call in specific register locations.
2. "transfer control" to procedure. (i.e. jump to start of procedure code)
3. Get local storage (stack)
4. Execute procedure code
5. place result of procedure in a known register for caller to access.
6. Release local storage
7. return to caller (for continuation)

Mips To handle procedure calls

Two main instructions:

```
jal Lable   # jumps to Lable
            # and puts next instruction address into $ra.
```

jal: Jump and link

```
jr $ra      # copies $ra to "program counter"
```

jr: jump register

The Mips Register File

5 bit → 32 total:

8 $S0, $S1, ... $S7 → on return from procedure call must have old value.

10 $t0, $t1, ... $t9 → not guaranteed to have old values when returning from a procedure call.

$ra: return address.

mem base pointers:

- $sp: stack pointer → the base of local variable address
- $Fp: Frame pointer
- $gp: global pointer (for base of static memory)

$v0, $v1: procedure call result.

$a0, .. $a3: argument registers

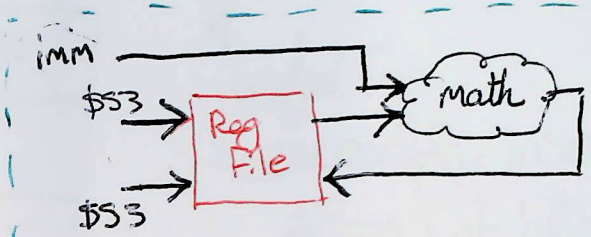
Immediate operands

C: a++; assump: 'a' is in $s3

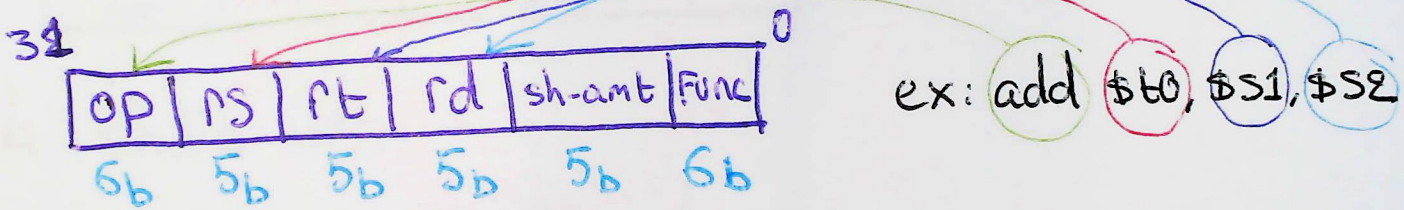
MIPS asm:

addi $s3, $s3, 1

↳ 2's complement.



MIPS R-Format Instruction



- The bits that tell our processor "what to do".
- MIPS has a fixed-width ISA (32 bits)

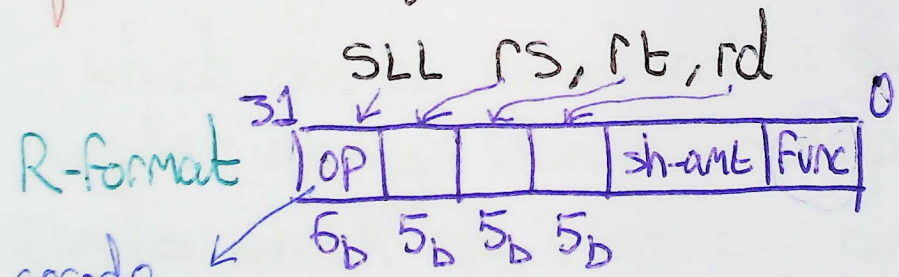
Logic operations

instructions for bitwise manipulation

op	C	MIPS opcode
shift left	<<	sll
shift right	>>	srl
and, or not nor	&, | ~	and, or not, nor

useful for extracting / inserting groups of bits.

example shift left



same for: and, or, shift right

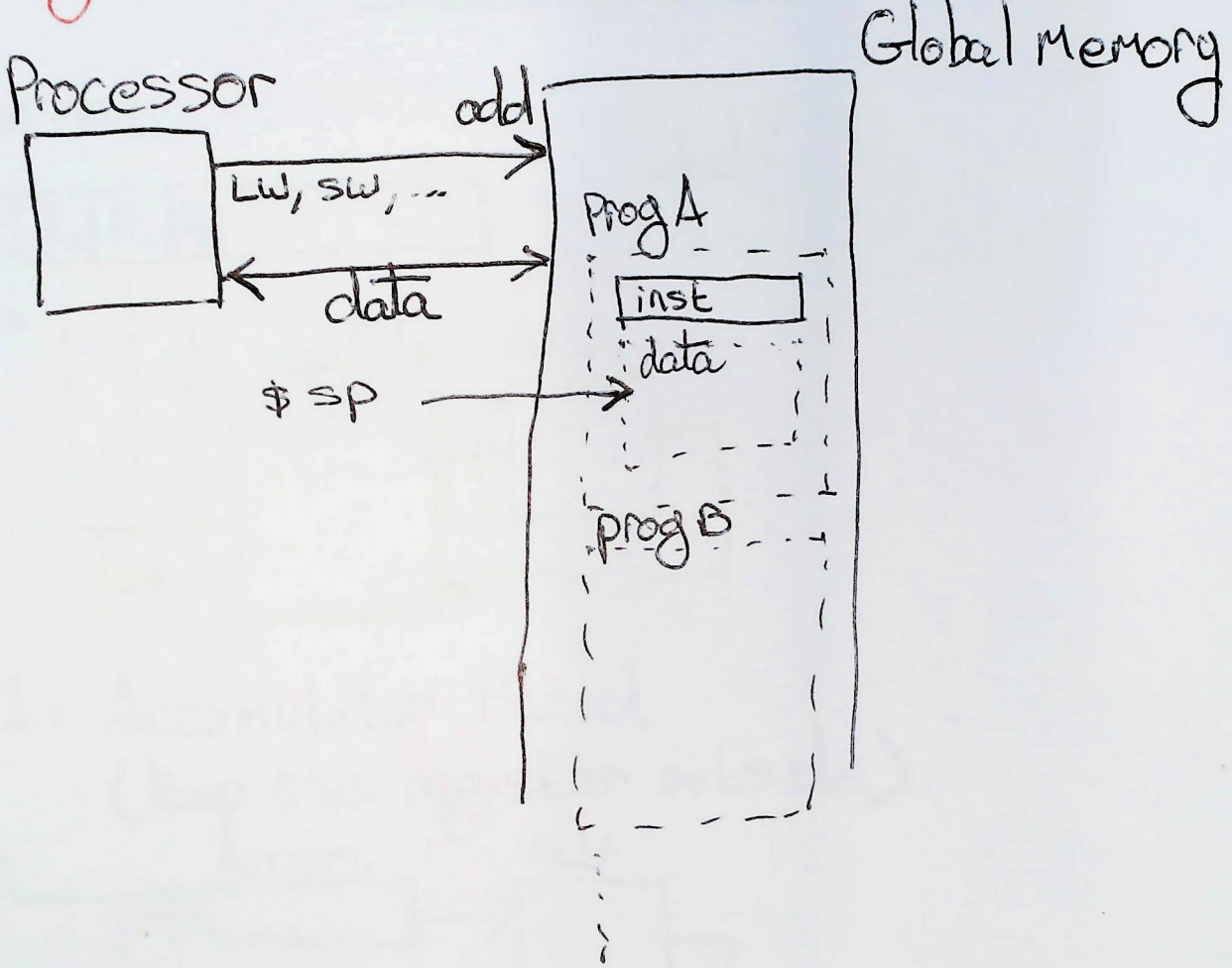
Type of Register-Register access

1) Full multiported Reg File
   - is complex to design
   - need "double pumping" clock
   - circuit design constraints
   + no restriction on what registers can be used together.

2) Bank Register File
   - splits registers into banks,
     which can be accessed in same clock.
   + simpler design than Full multiported reg file.
   - Limits which registers can be used by same instruction.

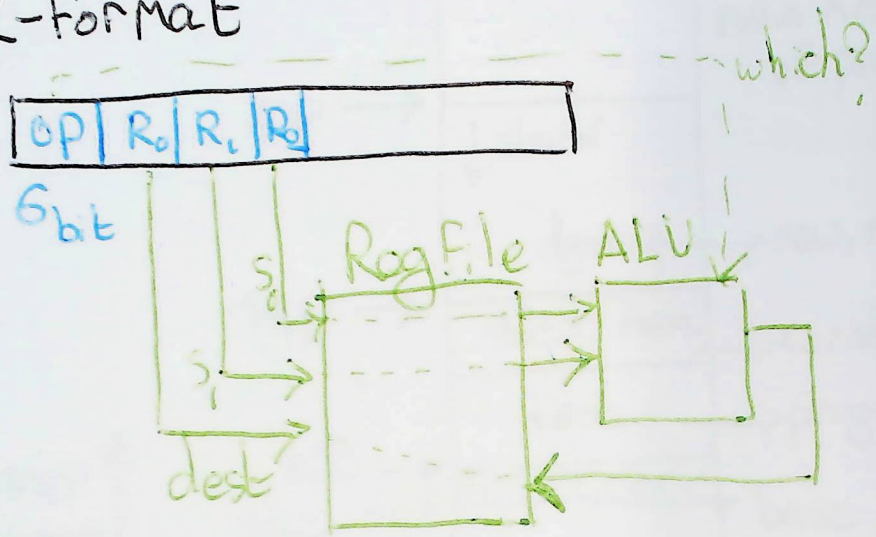
3) Accumulator-Based Register File *
   - Banked, but one bank is a single register
   + one register is implied (no address needed)
   - greatly limits registers that can be used together.

stored-program concept

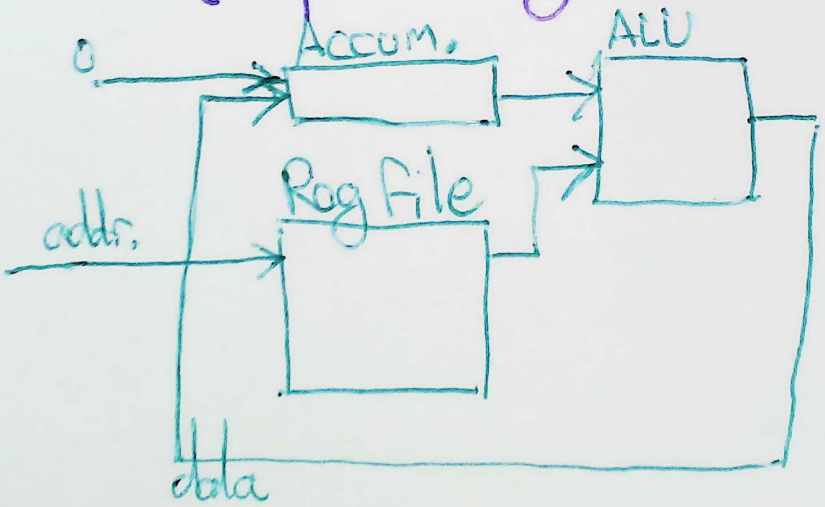


Register File with multiple ports

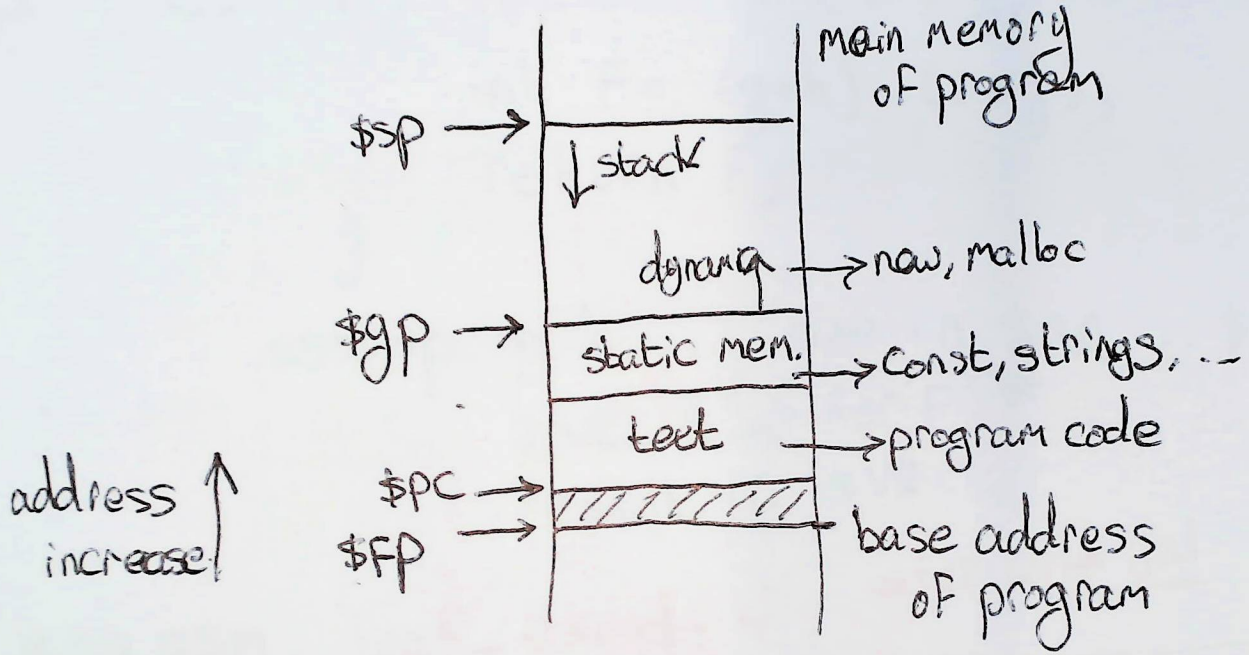
R-Format



Solution 1: Accumulator Based
(Keep one register outside)



MIPS Memory Layout



Example procedure

C:
```
int leaf_exmpl(int g, int h, int i, int j){
    int F= (g+h)-(i+j);
    return F;
}
```

assump: g, h, i, j are in $a0, ... $a3
use $s0 for 'F' *
result in $v0

Mips asm:
```
leaf_exmpl: <- enter with jal
    addi $sp, $sp, -4
    sw   $s0, 0($sp) # store $s0      store
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    addi $v0, $s0, 0
    lw   $s0, 0($sp) # load $s0       return
    addi $sp, $sp, +4
    jR   $rd
```

Example Nested procedure call

n! = n(n-1)(n-2)(n-3)... 1

```
c: int Fact(int n){
       if(n<1) return 1;
       else return (n * Fact(n-1));
   }
```

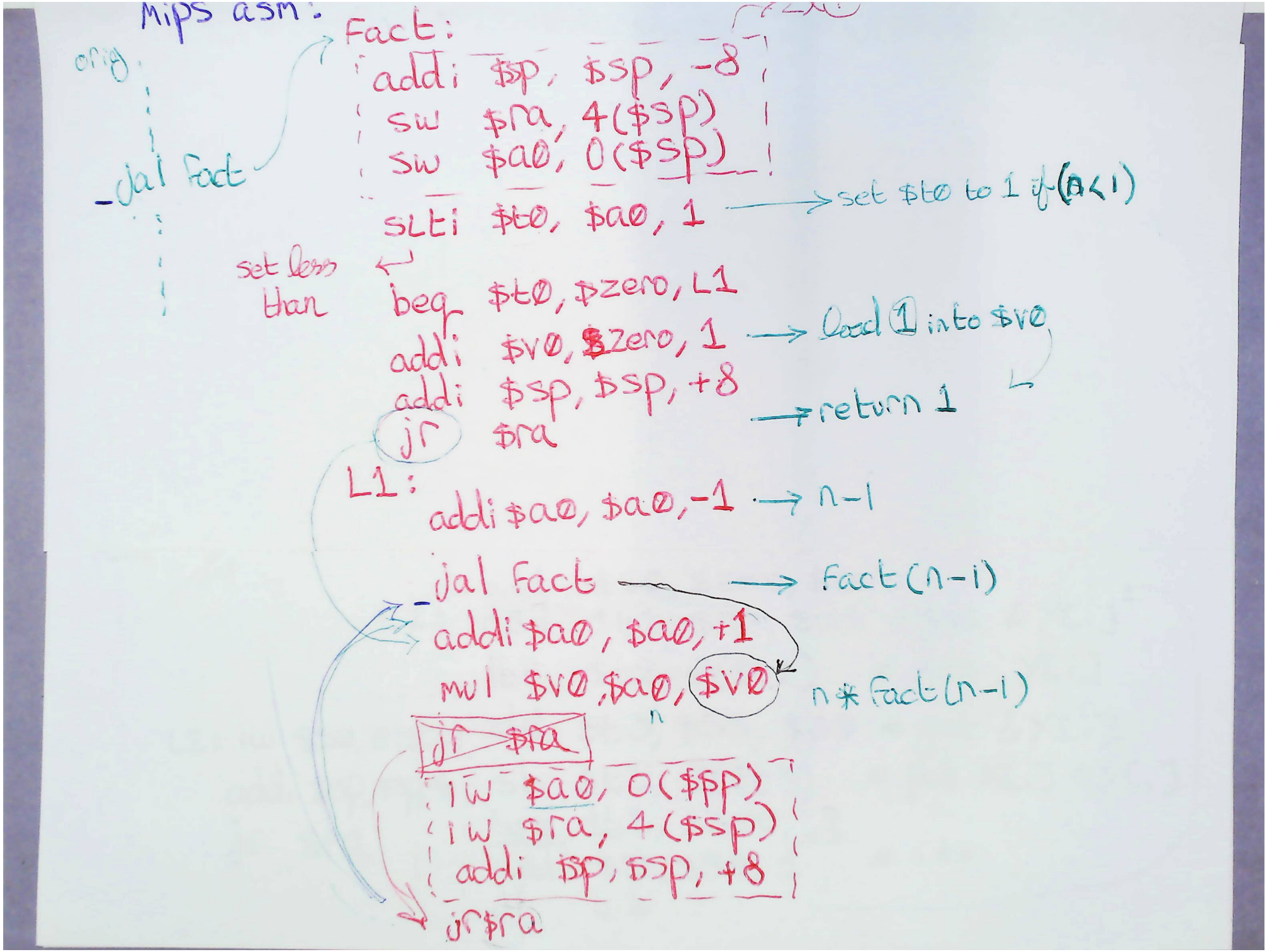
assump
n is in $a0
result is in $v0

Mips asm:

```
Fact:
    addi $sp, $sp, -8        → 2x4   (4 bytes in w)
    sw   $ra, 4($sp)
    sw   $a0, 0($sp)
    slti $t0, $a0, 1         → set $t0 to 1 if (n<1)
    beq  $t0, $zero, L1      (set less than)
    addi $v0, $zero, 1       → load 1 into $v0
    addi $sp, $sp, +8
    jr   $ra                 → return 1
L1:
    addi $a0, $a0, -1        → n-1
```

orig.

jal Fact


Mips asm:
orig.
jal fact
Fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slti $t0, $a0, 1
set $t0 to 1 if (n<1)
set less than
beq $t0, $zero, L1
addi $v0, $zero, 1
load 1 into $v0
addi $sp, $sp, +8
jr $ra
return 1
L1:
addi $a0, $a0, -1
n-1
jal fact
fact(n-1)
addi $a0, $a0, +1
mul $v0, $a0, $v0
n * fact(n-1)
jr $ra
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, +8
jr $ra

String manipulation Example:

```
C: void strcpy (char x[], char y[])        #copy y
   {                                        into x
       int i=0;
       while((x[i]=y[i])!='\0')
           i++;
   }
```

assump: x, y in $a0, $a1
i in $s0

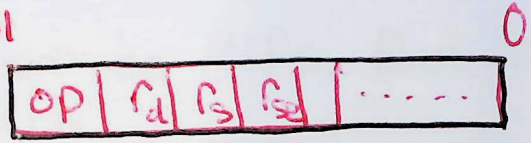
```
Mips: strcpy: addi  $sp, $sp, -4      # 4 bytes
              sw    $s0, 0($sp)

              add   $s0, $zero, $zero
        L1:   add   $t1, $s0, $a1     # get &y[i]
              lbu   $t2, 0($t1)       # get  y[i]
              add   $t3, $s0, $a0     # get &x[i]
              sb    $t2, 0($t3)       # put x[i] <- y[i]
              beq   $t2, $zero, L2
              addi  $s0, $s0, 1       # i++
              jf    L1

        L2: lw  $s0, 0($sp)
            addi $sp, $sp, 4
            jr  $ra
```

main ... jal strcpy

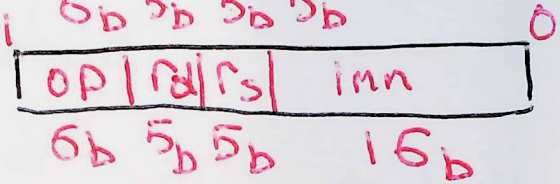
J-Format

Reminder

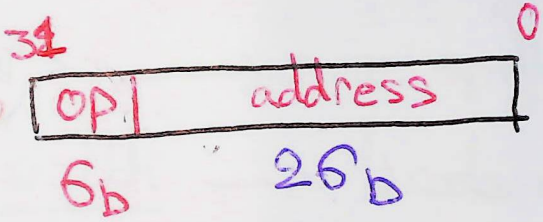
R-Format



I-Format



J-Format



↓

val

j

$target = PC_{31:28} : Add_{26_b}$

MIPs v. Arm v. x86 v. Risc-V

Risc-V: very similar to MIPs. But open source.
Fixed width

ARM: popular mobile

16 Register
Fixed width (later extended).

x86:
8 Registers ⟶ more spilling of reg to memory.*

variable width ⟶ decode more complicated
+more instruction flexibility