Write efficient programs

- It is important to understand how the computer sees a high-level language program such as a program written in c.
- Sometime we can make the program run more than 10 times faster, just by understanding how the computer sees things
- Knowledge of instruction set architecture helps you write efficient programs

Example

- A simple c program:

```
int func (int a, int b) {
      int sum = a + b;
      if ((a % 2) == 1 || (a % 2) == -1) //test odd
            sum++;
      return sum / 2;
}
```

1

20

20

2

Conditional branch 1

Conditional branch 1

20

- Approximate its efficiency:
  - \+ takes 1 clock cycle
  - % takes 20 clock cycles
  - / takes 20 clock cycles
  - == takes 1 clock cycle
  - Sum++ takes 2 clock cycles

65 clock cycles

Tips for optimization

- ++i is never slower than i++, always use ++i
- Modulo dominates, so let's focus on that
- Test for a odd or even
  - We only need to look at the last bit
  - 110010 is even and 110011 is odd
  - Bitwise AND looks per bit
    - a&1
    - Shows whether last bit is set
    - Works regardless of sign

Improvement

- Improved c program

```
int func (int a, int b) {
    int sum = a + b;
    if ((a & 1) == 1) //test odd
        ++sum;
    return sum / 2;
}
```

1

1

1

Conditional branch
1

20

- Approximate its efficiency:
  - \+ takes 1 clock cycle
  - & takes 1 clock cycle
  - == takes 1 clock cycle
  - ++sum takes 1 clock cycle
  - / takes 20 clock cycle

24 clock cycles

Further optimization tips

- Division dominates
- Division by multiple of 2 can be achieved through shift right logical
  - Divide sum by 2 -> shift sum to the right by 1
- Finding: division by x where x is a multiple of 2 can be reduced to right shift by Log_2x.
  - X=2 -> $Log_2X=1$ -> right shift by 1
  - X=4 -> $Log_2X=2$ -> right shift by 2
  - X=8 -> $Log_2X=3$ -> right shift by 3
  - …

Further improvement

- Further optimized c program

```
int func (int a, int b) {
    int sum = a + b;
    if ((a & 1) == 1) //test odd
        ++sum;
    return sum >> 1;
}
```

1

1

1

Conditional branch
1

1

- Approximate its efficiency:
  - \+ takes 1 clock cycle
  - & takes 1 clock cycle
  - == takes 1 clock cycle
  - ++sum takes 1 clock cycle
  - \>> takes 1 clock cycle

5 clock cycles

Check yourself

- Why does C provide two sets of operators for the logic AND operation, i.e. && and &?

Check yourself

- Why does C provide two sets of operators for the logic AND operation, i.e. && and &?

- Answer:
  - && and & are used for different purposes and therefore are implemented differently
  - & is implemented through the logic and operation in MIPS
  - && is implemented through the conditional branch instructions.

Support procedure call

- Procedure: a stored subroutine that performs a specific task based on the parameters (i.e. arguments) which it is provided.

- Steps to fulfill a procedure call
  - Put parameters in a place where the procedure can access them.
  - Transfer control to the procedure
  - Acquire storage resources (heap or stack) needed for the procedure
  - Perform the desired task
  - Put the result value in a place where the calling program can access it.
  - Return control to the point of origin

Architectural support

- Register allocation conventions:
  - \$a0 - \$a3 (\$4 - \$7): four argument registers in which to pass parameters
  - \$v0 - \$v1 (\$2, \$3): two value registers in which to return values
  - \$ra (\$31): one return address register to return to the point of origin
- Instruction support
  - jump-and-link instruction (**jal**)
    - Jumps to an address and simultaneously saves the address of the following instruction in \$ra

```
jal  ProcedureAddress
```

Unconditional jump

- Jump (**j**):

  j label

  - Jump to a location as indicated by the label
  - Address is given based on PC+4

- Jump register (**jr**)

  jr $ra

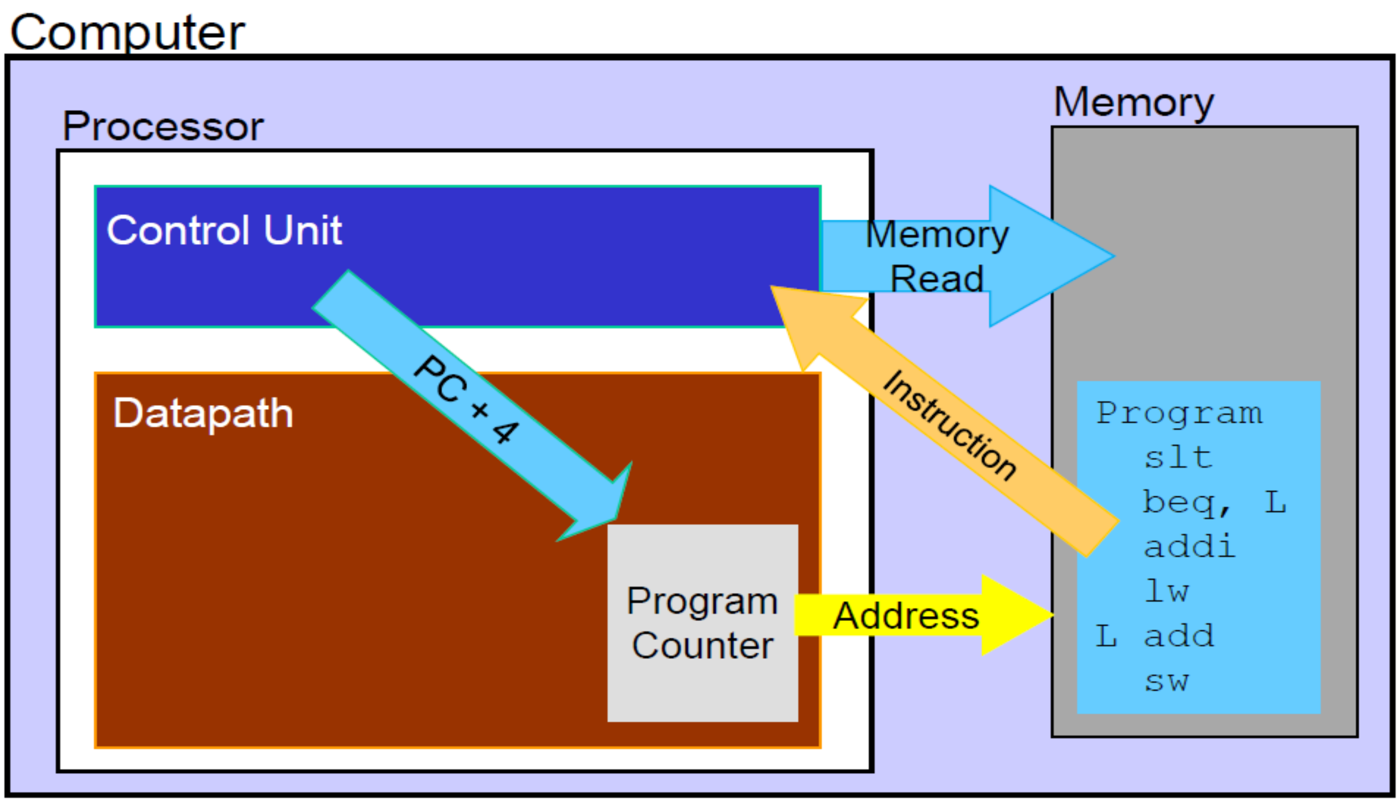
  - Unconditional jump to the address specified in a register
  - Address is given in the absolute form

Quick exercise

- Which of the following is true about jump and conditional branch instructions
  - A. In a 32 bit instruction you can not include a 32-bit jump destination
  - B. The offset in a bne instruction is a signed value.
  - C. The branch offset in a beq instruction cannot be zero.
  - D. The address in a j instruction is not defined as an offset.

Program counter

- Program counter (PC) is a piece of hardware inside the datapath that contains the memory address of the instruction to be executed next



Using the stack

- Stack
  - a last-in-first-out queue in memory
  - for spilling registers (why?)

- Push
  - add element to stack
- Pop
  - remove element from stack

Performed on the top of the stack

- Stack pointer ($sp or $29)
  - The most recently allocated memory address in a stack
  - Location for spilling new registers
  - Location for retrieving old registers

Procedure call example

- Compile a c procedure that doesn't call another procedure

```
int leaf_example (int g, int h, int i, int j)
{
        int f;

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

$t0 keeps the temporary result

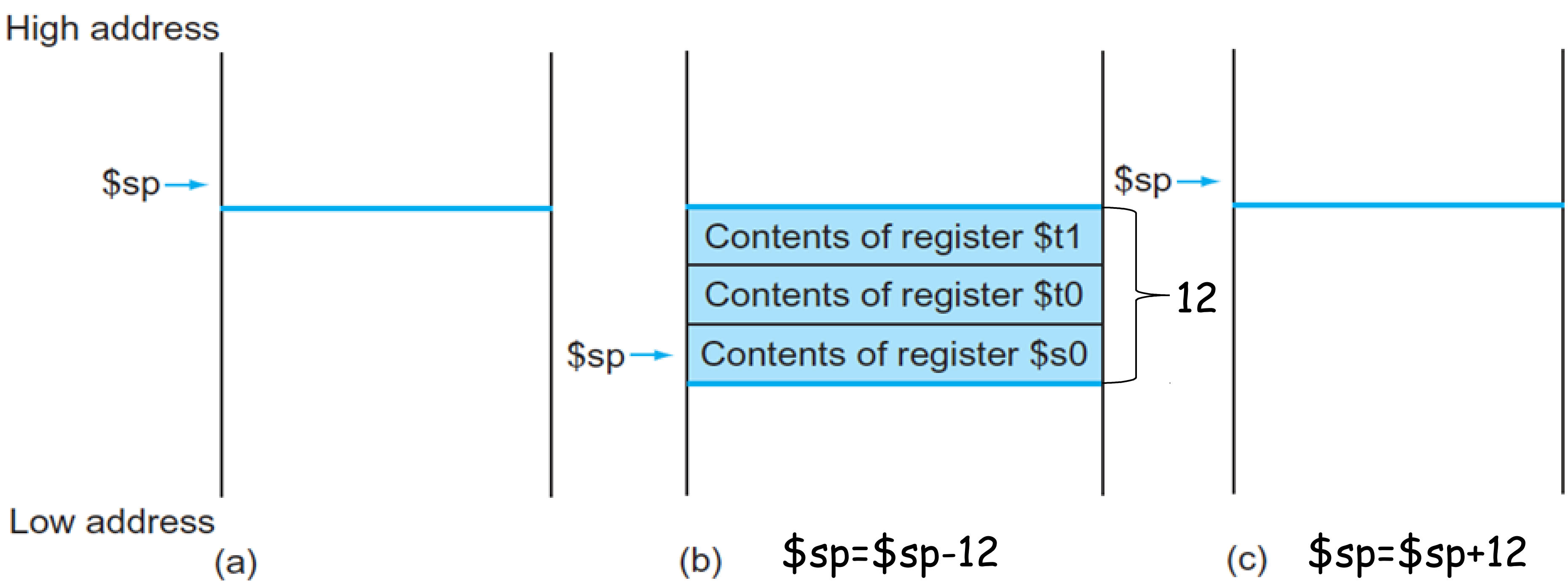
$t1 keeps the temporary result

- Register allocation:
  - g: $a0
  - h: $a1
  - i: $a2
  - j: $a3
  - f: $s0

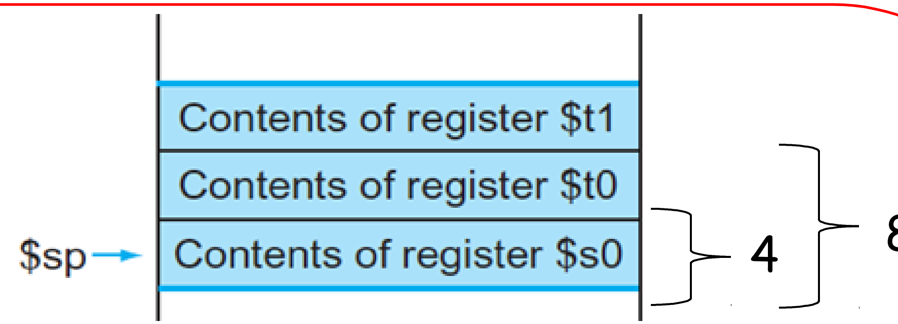
arguments

Change of stack through the procedure call

- Stack moves backward in memory from high address to low address



The MIPS version



- MIPS assembly code:

```
leaf_example:
addi $sp, $sp, -12    # adjust stack to make room for 3 items
sw   $t1, 8($sp)      # save register $t1 for use afterwards
sw   $t0, 4($sp)      # save register $t0 for use afterwards
sw   $s0, 0($sp)      # save register $s0 for use afterwards

add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)

add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)

lw  $s0, 0($sp)  # restore register $s0 for caller
lw  $t0, 4($sp)  # restore register $t0 for caller
lw  $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12  # adjust stack to delete 3 items

jr  $ra     # jump back to calling routine
```

Quick exercise

- Which code segment below is correct in order to preserve the value of $s0 in the stack?

  - A.
    ```
    addi $sp, $sp, -1
    sw $s0, 0($sp)
    ```
  - B.
    ```
    addi $sp, $sp, -4
    sw $s0, 0($sp)
    ```
  - C.
    ```
    sw $s0, 0($sp)
    addi $sp, $sp, -1
    ```
  - D.
    ```
    sw $s0, 0($sp)
    addi $sp, $sp, -4
    ```

Convention that reduces register spilling

- Temporary registers are not preserved by the callee
  - $t0 - $t9

- Saved registers must be preserved on a procedure call
  - $s0 – $s7
  - If used, the callee must save and restore them

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1

Summary of register conventions

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Quick exercise

- Which of the registers below should be preserved in the stack by the leaf function?

```
void leaf() {
    $t0 = 1;
    $s0 = $t0+1
}
```

- A. $t0
- B. $sp
- C. $ra
- D. $s0

Nested procedures

- Procedure that call other procedures (including itself)

- Things to be aware
  - Conflict use of argument registers ($a0 - $a4)
  - Conflict use of $ra

- Things to do:
  - The caller pushes any argument registers or temporary registers that are needed after the call.
  - The callee pushes the return address and any saved registers used by the callee.
  - Adjust $sp accordingly.

Example

- Compile a recursive (nested) c procedure

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

- Register allocation:
  - n: $a0

fact(4)
-> 4 * fact(3)
-> 4 * 3 * fact(2)
-> 4 * 3 * 2 * fact(1)
-> 4 * 3 * 2 * 1 * fact(0)
=4!=24

```
fact:
     addi   $sp, $sp, -8  # adjust stack for 2 items
     sw     $ra, 4($sp)   # save the return address
     sw     $a0, 0($sp)   # save the argument n

     slti   $t0,$a0,1       # test for n < 1
     beq    $t0,$zero,L1    # if n >= 1, go to L1

     addi   $v0,$zero,1  # return 1
     addi   $sp,$sp,8    # pop 2 items off stack
     jr     $ra          # return to caller

L1:  addi  $a0,$a0,-1    # n >= 1: argument gets (n - 1)
     jal   fact          # call fact with (n - 1)

     lw    $a0, 0($sp)    # return from jal: restore argument n
     lw    $ra, 4($sp)    # restore the return address
     addi  $sp, $sp, 8    # adjust stack pointer to pop 2 items

     mul  $v0,$a0,$v0    # return n * fact (n - 1)

     jr   $ra            # return to the caller
```

Quick exercise

- What is missing in the process of making a function call?

1. Adjust $sp
2. Preserve $ra
3.
4. Call function using jal
5. Restore $ra
6.
7. Adjust $sp

  - A. Preserve and restore $v0 and $v1
  - B. Preserve and restore $s0 to $s7
  - C. Preserve and restore $a0-$a3
  - D. Preserve and restore $gp

Recursion or iteration?

- Some recursive procedures can be implemented iteratively without using recursion.
- Iteration can significantly improve performance by removing the overhead associated with procedure calls and therefore is more preferable.

```
int sum (int n, int acc) {
  if (n > 0)
      return sum(n - 1, acc + n);
  else
      return acc;
}
```

```
sum:
      beq$a0, $zero, sum_exit
      add$a1, $a1, $a0
      addi$a0, $a0, -1
      j sum
sum_exit:
      add$v0, $a1, $zero
      jr $ra
```

$acc + n + (n-1) + (n-2) + \ldots + 1$

Quick exercise

- How to allow a function to return more information that cannot be completely stored in $v0 and $v1?
  - A. Use other registers such as $a0, $s0, etc. to carry more information.
  - B. Store extra information in the stack when returning back to the function caller.
  - C. Store the return results in the heap and pass the memory address back to the function caller.
  - D. None of the above