TUTORIAL 8 NESTED PROCEDURE CALL AND ARITHMETIC LOGIC UNIT

Overview

- **We will review the following concept in this tutorial:**

- **Nested procedure**

- **Recursion (nested procedure calling itself)**
  - Register convention

- **1-bit Arithmetic Logic Unit (ALU)**
  - add, sub, and, or
- **32-bit ALU**
  - add, sub, and, or, beq, slt

`jr` and `jal` Instructions for Procedure Call

<table>
<tr><td>Jump Registers</td><td>Usage: jr <Register></td></tr>
<tr><td colspan="2">Jump to the address contained in register.</td></tr>
<tr><td>Jump and Link</td><td>Usage: jal <target></td></tr>
<tr><td colspan="2">Jumps to the calculated address and stores the return address in $ra
Operation: Update PC</td></tr>
</table>

Warn up exercise

- **Explain the problem of this MIPS program, and suggest the solution.**

```
.text
.globl main
main:
        jal calculate
        addi $s0, $v0, 0
        li $v0, 10
        syscall
calculate:
        addi $a0, $zero, 1
        addi $a1, $zero, 2
        jal add_two_numbers
        jr $ra
add_two_numbers:
        add $v0, $a0, $a1
        jr $ra
```

PUSH	addi $sp, $sp, -4 sw $ra, 0($sp)
POP	lw $ra, 0($sp) addi $sp, $sp, 4

PUSH

POP

Nested Procedure Call Example: `print_even`

```
# print even numbers in [1, 10]

.text #text segment
.globl main
#---------------------------------------------------------
# void main()
# for (int i=0; i<10; i++)
#       print_even(i);
# $s0: i
main: # caller-callee pair: main and print_even
        addi $s0, $zero, 1      # $s0 is i
loop:
        slti $t0, $s0, 10       # if (i < 10) iterate
        beq $t0, $zero, exit
        add $a0, $s0, $zero     # print_even(i)
        jal print_even
        addi $s0, $s0, 1        # i++
        j loop
exit:
        li $v0, 10              # exit
        syscall
#---------------------------------------------------------
```

Nested Procedure Call Example: `print_even`

```
#-------------------------------------------------------
# void print_even(i)
# $a0: i

# MIPS convention: $s registers should be preserved
# It's callee's responsibility to perserve any $s registers it's going to use
# If every callee does this, the all callers can feel safe to put their important data in $s registers
# $s register content remains unchanged before and after the procedure call

# $ra should also be preserved if there's nested procedure call

print_even: # caller-callee pair: print_even and is_even
        addi $sp, $sp, -4       # push $ra
        sw $ra, 0($sp)
        addi $sp, $sp, -4       # push $s0
        sw $s0, 0($sp)

        add $s0, $a0, $zero     # backup input argument i to preserved register $s0
        jal is_even             # is_even(i), returns 1 if i is even, 0 otherwise
        beq $v0, $zero, print_even_end
        add $a0, $s0, $zero     # print i if it's even
        li $v0, 1
        syscall
print_even_end:
        lw $s0, 0($sp)          # pop $s0
        lw $ra, 4($sp)          # pop $ra
        addi $sp, $sp, 8
        jr $ra                  # return
#-------------------------------------------------------
```

Nested Procedure Call Example: `print_even`

```
#----------------------------------------------------------
# bool is_even(i)
# $a0: i
# $v0: return value
is_even:
        addi $sp, $sp, -4       # push $ra (optional since is_even is a leaf procedure)
        sw $ra, 0($sp)
        addi $v0, $zero, 0
        andi $t0, $a0, 1        # $t0 is 0 if i is odd, 1 if i is even
        bne $t0, $zero, is_even_end
        addi $v0, $zero, 1      # $a0 is even
is_even_end:
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra
#----------------------------------------------------------
```

Recursion Example: `factorial`

```
# --------------------------------------------------------------------------
main: # caller-callee pair: main() and factorial()
    # print prompt
    li      $v0, 4              # pseudo instruction
    la      $a0, prompt         # pseudo instruction
    syscall                     # system call #4 - print string
    # read x
    li      $v0, 5              # system call #5 - read int
    syscall                     #x is returned in $v0
    # call factorial()
    move    $a0, $v0            # pseudo instruction, $a0 = n
    jal     factorial           # call factorial(n)
    move    $s0, $v0            # $vo = factorial(n), save it in $s0
    # print prompt
    li      $v0, 4
    la      $a0, result
    syscall
    # print result
    move    $a0, $s0            # $a0 = $s0 factorial(n)
    li      $v0, 1              # system call #1 - print int saved in $a0
    syscall                     # execute
    # exit
    li      $v0, 10             # system call #10 - exit
    syscall
# --------------------------------------------------------------------------
```

Recursion Example: `factorial`

```
# ------------------------------------------------------------------------
factorial: # caller-callee pair: factorial(n) and factorial(n-1)
        addi $sp, $sp, -4       #push $ra, factorial(n)'s duty as callee
        sw $ra, 0($sp)
base_case:
        bne $a0, $zero, recursive_case  # if n >= 1, goto recursive case
        addi $v0, $zero, 1              # base case, f(0) = 1
        j factorial_end
recursive_case:
# when factorial(n) calls factorial(n-1), $a0 will change from n to n-1
# but factorial(n) still needs to do n x factorial(n-1) after factorial(n-1) returns
# $a is not preserved in MIPS convention. So it's factorial(n) caller's repsonsibility to perserve it.
        addi $sp, $sp, -4       # push $a0 (to save value n), caller's role
        sw $a0, 0($sp)

        addi $a0, $a0, -1       # now $a0 = n-1
        jal factorial           # call factorial(n-1)
        # when factorial returns, f(n-1) is in $v0
        lw $a0, 0($sp)          # restores value n from stack, callee's role
        addi $sp, $sp, 4

        mult $a0, $v0   # f(n) = n * f(n-1), f(n-1) is in $v0, which is the return value of f(n-1)
        mflo $v0        # now $v0 holds f(n), f(n) is going to be returned to main() in $v0
factorial_end:
        lw $ra, 0($sp)  # pop $ra, factorial(n)'s duty to restore preserved register(s)
        addi $sp, $sp, 4
        jr $ra
# ------------------------------------------------------------------------
```

callee role

caller of fac(n-1)

caller of fac(n-1)

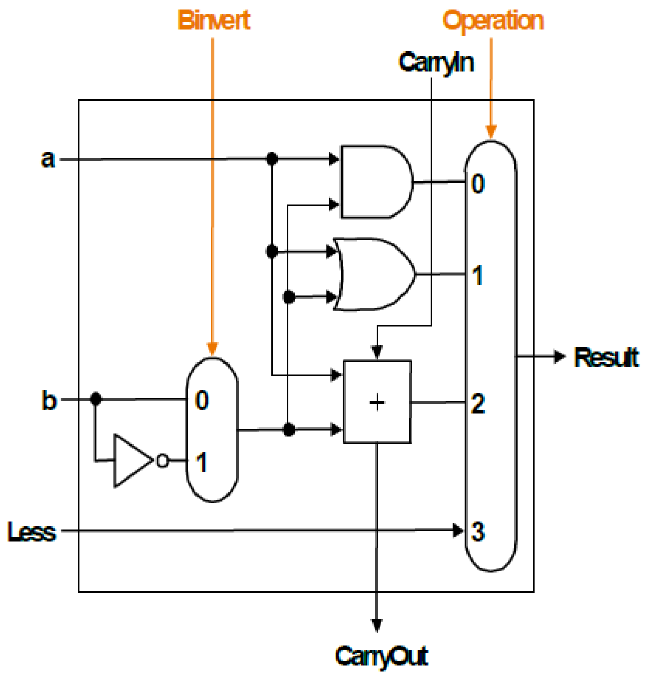
callee role

32-bit MIPS Arithmetic and Logic Unit (ALU)

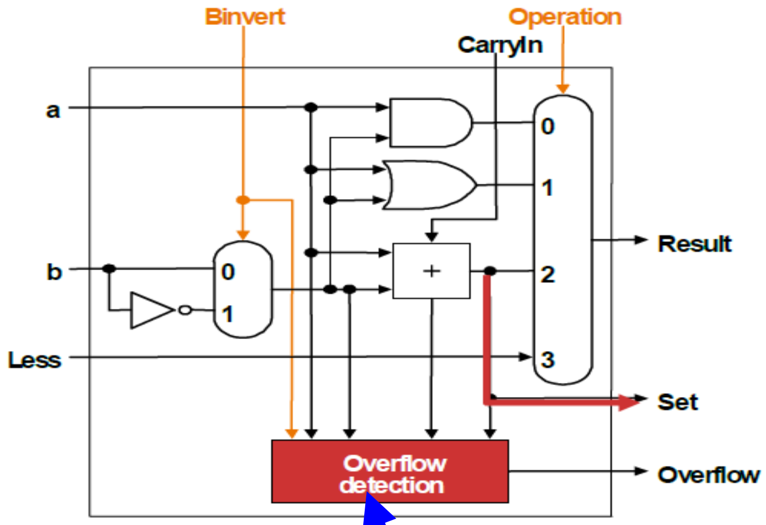
- **First, build 1-bit ALU**
  - 1-bit logic unit (AND and OR)
  - 1-bit full adder (ADD)
  - Combine the above and build 1-bit ALU (ADD, SUB, AND, OR)
- **Then, build 32-bit ALU by connecting 32 of 1-bit ALU**
  - Connect ALU_0 to ALU_{31} with ripple carry structure
  - ALU_{31} works on MSb (sign bit) so it’s a bit different from ALU_0 to ALU_{30}
  - Further add support to SLT and BEQ

1-bit ALU

- 1-bit ALU for bit 0 to bit 30



- 1-bit ALU for the MSb (ALU_{31})

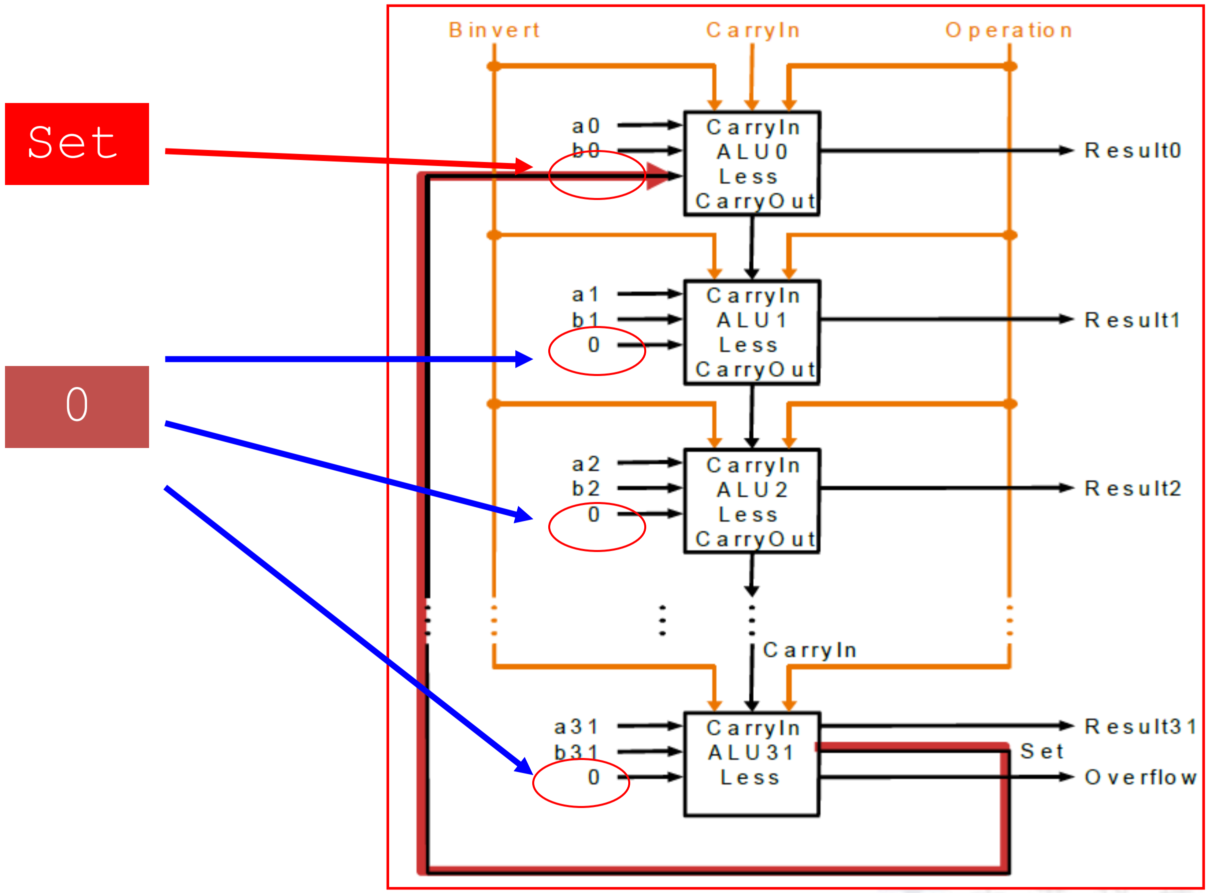


Overflow conditions

Operation	Sign bit of A	Sign bit of B	Sign bit of Result
A + B	0	0	1
A + B	1	1	0
A - B	0	1	1
A - B	1	0	0

32-bit ALU with support to SLT

- An extended 32-bit ALU (supports SLT) can be formed by connecting 32 1-bit ALUs as follows. Note the **0**'s at the "`Less`" input for ALU1-ALU31, note also the `set` signal from ALU_{31} to ALU_{0}.



Exercise 1

- Some argue that the control signals Binvert and CarryIn of the bit-0 ALU can be combined into one control signal. Justify this claim (refer to the ALU diagrams on slides 10 whenever necessary).

Solution:

Addition operation: both Binvert and CarryIn of the bit-0 ALU should be 0.

Subtraction operation: both Binvert and CarryIn of the bit-0 ALU should be 1.

OR/AND operations: Binvert should be 0, CarryIn should be “don’t care”.

Combining the two signals does not impair their functions. Therefore it is okay to combine both signals into one. In fact they are combined to form the Bnegate signal.

Exercise 2

- Refer to the previous ALU slides, explain how SLT operation can be performed. State the values for the control signals `Binvert`, `CarryIn` and `Operation`.

Solution:

SLT outputs an "1" when the upper operand A is less than the lower operand B. The subtraction operation A-B will be performed.

When A-B<0, the sign bit (result of the MSB) will be 1 and will be forwarded to ALU0 (so "Less" becomes 00…01) .

When A-B>=0, "Less" will be 00…00.

The signals Binvert and CarryIn of ALU0 should be set to "1" to enable the subtraction, the signal Operation should be set to 3 ($11_{(2)}$) to enable the resulting "set" to be forwarded to the output.

Exercise 3

Refer to the previous ALU slides, derive the logic expression in the Sum of Product form (SoP) for overflow conditions.

Solution: Two types of overflows according to the table below,

1) addition overflow — **Binvert=0, a3=b3=0, set=1 or**
Binvert=0, a3=b3=1, set=0

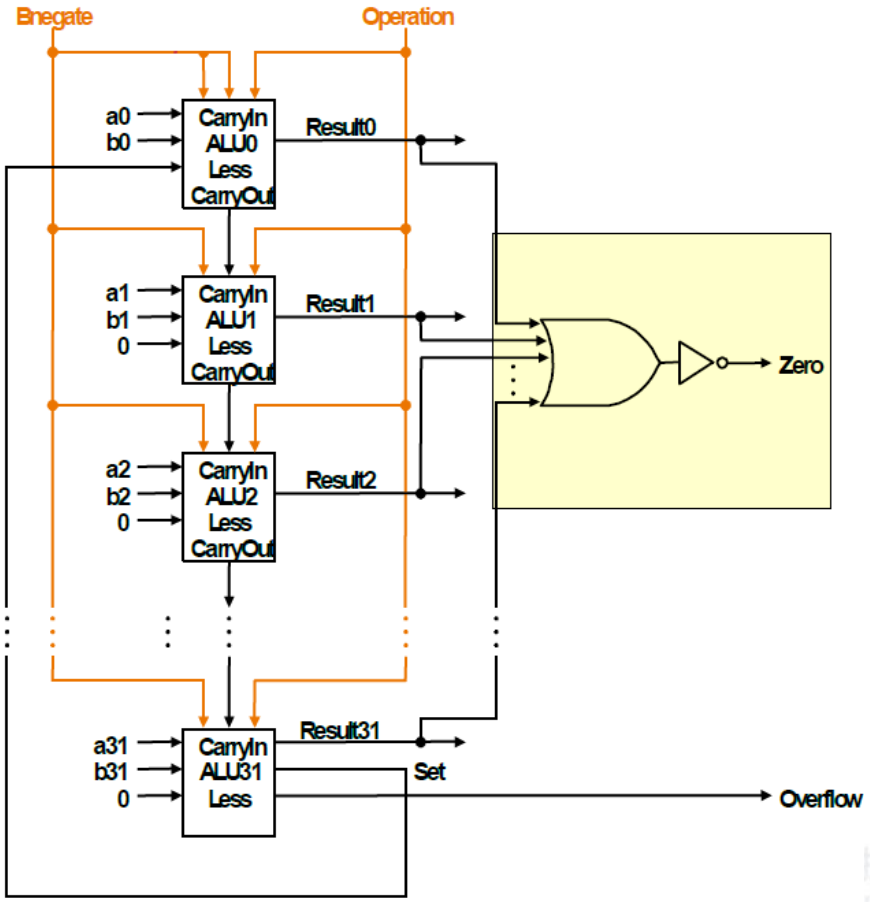
2) subtraction overflow — **Binvert=1, a3=0,b3=1, set=1 or**
Binvert=1, a3=1,b3=0, set=0

Operation	Sign bit of A	Sign bit of B	Sign bit of Result
A + B	0	0	1
A + B	1	1	0
A - B	0	1	1
A - B	1	0	0

The corresponding SoP is: $\overline{\text{Binvert}} \cdot \overline{\text{a3}} \cdot \overline{\text{b3}} \cdot \text{set} + \overline{\text{Binvert}} \cdot \text{a3} \cdot \text{b3} \cdot \overline{\text{set}} + \text{Binvert} \cdot \overline{\text{a3}} \cdot \text{b3} \cdot \text{set} + \text{Binvert} \cdot \text{a3} \cdot \overline{\text{b3}} \cdot \overline{\text{set}}$

Exercise 4

- Refer to the modified 32-bit ALU below, explain how the condition `A==B` is detected. State the values for the control signals `Bnegate` and `Operation`



Exercise 4

- **Solution**: To check A==B, we perform the subtraction A-B, if the result is 0 (i.e. result0=...=result31=0) then A==B. The NOR gate in the figure will output 1 iff all the result bits are 0. Thus if the NOR gate outputs 1, then A==B. To perform the subtraction, Bnegate is set to 1 and Operation is set to 10.