Jon Bowen

CS 3210

In Class Exercise 1/30/2018

Cover Letter Discussions

1. Yes the code will function correctly without any lines in the conditional blocks because execution will simply branch to that execution section and then immediately branch back without performing any action.
2. Nesting if-else statements in a top-test loop would not require change to the if-else structure since all that is required for if-else statements is moving to the block to be executed (via branch or fall through) and skipping the block to be skipped (again via branch or fall through). Nesting top-test loops is likewise straightforward *in principal*. Care must be taken that execution proceeds to the correct position following tests and completion of each block. As nesting becomes more complicated, pulling out blocks into clearly labeled segments and branching to and from these complete blocks would likely keep the code cleaner.
3. The hardest thing about the decompiling process is the many ways that the same result can be implemented in PAL and HLLs. Provides a 1-1 mapping is always used exactly, this can be simplified because the programmer knows exactly the HLL code segment that generated the PAL segment and vice versa. Using this approach, the programmer need not consider all possible forms of top-test loops and if-else loops when looking for patterns in PAL to be converted to HLL.

I found the different approaches to getting the same result in PAL to be interesting. Initially for the if-else I was essentially branching to subroutines to execute the code blocks then branching back after the if-else clause. This may be cleaner for complex compound statements. However, simple if-else clauses have only two branches. In this case falling through to the first path and skipping the second or skipping the first path and branching to the second avoids needless complication and keeps relevant code close together.

1. From PAL’s perspective there is no difference between *while* and *for* loops. The *for* loop in a HLL is purely a syntactic convenience. Because of this, the decompiler explored here will decompile a HLL for loop into an equivalent while loop. This is simply a design choice. Other sensible choices include always decompiling HLL *for* and *while* loops to *for* loops or to attempt to make a guess about the original form based on the content of the PAL code within the loop.
2. Subroutine calls could be recognized by an unconditional branch to the subroutine start with that block at some point unconditionally branching back to the address immediately following the original branch instruction. Parameters to subroutine calls could be recognized by the subroutine code block using registers without setting their values first.
3. The constraint that if-else tests use only simple boolean expressions using a relational operator on integers simplifies the PAL code. Evaluating arbitrary test expressions that may include complex arithmetic expressions, multiple conditionals, or may evaluate to non-boolean results would make the decompiling process significantly more difficult. Arithmetic expressions would need to be evaluated before testing could be done. Multiple conditionals would require multiple tests, and non-boolean options would require a design decision about the meaning of values other than true and false (C is an example where this design choice was made).

PAL and HLL Code Fragments

TOP TEST LOOPS

\*\*\* Top Test Loop 1 \*\*\*

//--- Pseudo code (High Level Language)

int i = 0;

int sum = 0;

while (i < 5){

sum = sum + i;

i = i + 1;

}

;--- PAL

; R0: sum

; R1: loop counter

; R7: test

MOVE 0, R0

MOVE 0, R1

MOVE 5, R7

test:

BGT R7, R1, itr

BR fin

itr:

ADD R0, R1, R0

INC R1

BR test

fin:

END

\*\*\* Top Test Loop 2 \*\*\*

;--------------- Pseudo code (High Level Language)

int I = 0;

int sum = 0;

while (I <= 3){

sum = sum + I;

I = I + 1;

}

;--------------- PAL

; R0: sum

; R1: loop counter

; R7: test

MOVE 0, R0

MOVE 0, R1

MOVE 3, R7

test:

BGT R7, R1, itr

BEQ R7, R1, itr

BR fin

itr:

ADD R0, R1, R0

INC R1

BR test

fin:

END

IF-ELSE STATEMENTS

\*\*\* If-Else 1 \*\*\*

;--------------- Pseudo code (High Level Language)

int i = 2;

int sum = 3;

if (i < 4){

sum = 0;

}

;--------------- PAL

; R0: sum

; R1: i (variable to test)

; R7: test

MOVE 3, R0

MOVE 2, R1

MOVE 4, R7

BGT R1, R7, end

BEQ R1, R7, end

MOVE 0, R0

end:

END

\*\*\* If-Else 2 \*\*\*

;--------------- Pseudo code (High Level Language)

int i = 2;

int sum;

if (i <= 4){

sum = 0;

} else{

sum = 1

}

;--------------- PAL

; R0: sum

; R1: i (variable to test)

; R7: test

MOVE 2, R1

MOVE 4, R7

BGT R1, R7, else

MOVE 0, R0

BR end

else:

MOVE 1, R0

end:

END

Logic for Identifying PAL Code Fragments

There are many ways that both top-test loops and if-else statements can be written in PAL. To simplify matters, we assume that we know the specific and fixed forms that the compiler will use for each fragment.

For top-tested loops (both for and while loops are assumed to produce identical PAL fragments), we are given that loop counters always start at 0, are incremented by 1 each loop, and the loop runs until the test is false.

for( i = 0; i < n; i++ ) and while( i < n):

PAL form:

loop:

BGT R1, R0, body

BR skip

body:

; LOOP BODY

BR loop

skip:

; CONTINUE

Psuedocode to recognize:

if( statement has label )

lab1 = label

if (next instruction is BGT)

n = 1st branch operand

i = 2nd branch operand

lab2 = 3rd branch operand

if (“INC i” followed by “BR lab1" is the first branching after lab2)

lab3 = next label below "BR lab1" statement

if (next instr after BGT is "BR lab3")

return true

for( i = 0; i <= n; i++ ) and while( i <= n):

PAL form:

loop:

BGT R1, R0, body

BEQ R1, R0, body

BR skip

body:

; LOOP BODY

INC R0

BR loop

skip:

; CONTINUE

Psuedocode to recognize:

if( statement has label )

lab1 = label

if (next instruction is BGT)

n = 1st branch operand

i = 2nd branch operand

lab2 = 3rd branch operand

if (“INC i” followed by “BR lab1" is the first branching after lab2)

lab3 = next label below "BR lab1" statement

if (next instr after BGT is "BEQ n, i lab2")

if (next instr after "BEQ n, i, lab2" is "BR lab3")

return true

if ( i < n )

PAL Form:

BGT R1, R0, doif

BR skip

doif:

; IF BLOCK

skip:

; CONTINUE

Psuedocode to recognize:

if (next instr is BGT)

n = 1st branch operand

i = 2nd branch operand

lab1 = 3rd branch operand

if (BR follows BGT)

lab 2 = BR operand

if (1st label after BR == lab1 && 2nd label after BR == lab2)

return true

if ( i <= n )

PAL Form:

BGT R0, R1, addr

; IF BLOCK

addr:

; CONTINUE

Psuedocode to recognize:

if (next instr is BGT)

i = 1st branch operand

n = 2nd branch operand

lab1 = 3rd branch operand

if (1st label after BGT == lab1)

return true

if ( i == n )

PAL Form:

BEQ R0, R1, doif

BR skip

doif:

; IF BLOCK

skip:

; CONTINUE

Psuedocode to recognize:

if (next instr is BEQ)

i = 1st branch operand

n = 2nd branch operand

lab1 = 3rd branch operand

if (next instr after BEQ is BR)

lab2 = BR operand

if (1st label after BR == lab1 && 2nd label after BR == lab2)

return true

if ( i != n )

PAL Form:

BEQ R0, R1, skip

; IF BLOCK

skip:

; CONTINUE

Psuedocode to recognize:

if (next instr == BEQ)

i = 1st branch operand

n = 2nd branch operand

lab1 = 3rd branch operand

if (1st label after BEQ == lab1)

return true

if (R0 < R1) else:

PAL Form:

BGT R1, R0, doif

; ELSE BODY

BR skip

doif:

; IF BLOCK

skip:

; CONTINUE

Psuedocode to recognize:

if (next instr == BGT)

i = 1st branch operand

n = 2nd branch operand

lab1 = 3rd branch operand

if (if next branch is BR && next label == lab1)

lab2 = BR operand

if( next label == lab 2)

return true

if (R0 <= R1) else:

PAL Form:

BGT R0, R1, else

; IF BLOCK

BR skip

else:

; ELSE BODY

skip:

; CONTINUE

Psuedocode to recognize:

if (next instr == BGT)

n = 1st branch operand

i = 2nd branch operand

lab1 = 3rd branch operand

if (next branch == BR and next label == lab1)

if (next label == lab2)

return true

Logic to Convert PAL Fragments to HLL Fragments

while( i < n):

Psuedocode to recognize:

if( statement has label )

lab1 = label

if (next instruction is BGT)

n = 1st branch operand

i = 2nd branch operand

lab2 = 3rd branch operand

if (“INC i” followed by “BR lab1" is the first branching after lab2)

lab3 = next label below "BR lab1" statement

if (next instr after BGT is "BR lab3")

return CODE\_STRING

CODE\_STRING =

“int i = ” + i + “;”

<make any assignments from loop variables assigned before loop>

“while ( i < ” + n + “ ){”

<loop body>

“i = i + 1;”

“}”

while( i <= n):

if( statement has label )

lab1 = label

if (next instruction is BGT)

n = 1st branch operand

i = 2nd branch operand

lab2 = 3rd branch operand

if (“INC i” followed by “BR lab1" is the first branching after lab2)

lab3 = next label below "BR lab1" statement

if (next instr after BGT is "BEQ n, i lab2")

if (next instr after "BEQ n, i, lab2" is "BR lab3")

return CODE\_STRING

CODE\_STRING =

“int i = ” + i + “;”

<make any assignments from loop variables assigned before loop>

“while ( i <= ” + n + “ ){”

<loop body>

“i = i + 1;”

“}”

if ( i < n )

Psuedocode to recognize:

if (next instr is BGT)

n = 1st branch operand

i = 2nd branch operand

lab1 = 3rd branch operand

if (BR follows BGT)

lab 2 = BR operand

if (1st label after BR == lab1 && 2nd label after BR == lab2)

return CODE\_STRING

CODE\_STRING =

<make any assignments for if block>

“if ( i < ” + n + “) {”

<if block>

“}”

if ( i <= n )

Psuedocode to recognize:

if (next instr is BGT)

i = 1st branch operand

n = 2nd branch operand

lab1 = 3rd branch operand

if (1st label after BGT == lab1)

return CODE\_STRING

CODE\_STRING =

<make any assignments for if block>

“if ( i <= ” + n + “) {”

<if block>

“}”

if ( i == n )

Psuedocode to recognize:

if (next instr is BEQ)

i = 1st branch operand

n = 2nd branch operand

lab1 = 3rd branch operand

if (next instr after BEQ is BR)

lab2 = BR operand

if (1st label after BR == lab1 && 2nd label after BR == lab2)

return CODE\_STRING

CODE\_STRING =

<make any assignments for if block>

“if ( i == ” + n + “) {”

<if block>

“}”

if ( i != n )

Psuedocode to recognize:

if (next instr == BEQ)

i = 1st branch operand

n = 2nd branch operand

lab1 = 3rd branch operand

if (1st label after BEQ == lab1)

return CODE\_STRING

CODE\_STRING =

<make any assignments for if block>

“if ( i != ” + n + “) {”

<if block>

“}”

if (R0 < R1) else:

Psuedocode to recognize:

if (next instr == BGT)

i = 1st branch operand

n = 2nd branch operand

lab1 = 3rd branch operand

if (if next branch is BR && next label == lab1)

lab2 = BR operand

if( next label == lab 2)

return CODE\_STRING

CODE\_STRING =

<make any assignments for if and else blocks>

“if ( i < ” + n + “) {”

<if block>

“} else{”

<else block>

“}”

if (R0 <= R1) else:

Psuedocode to recognize:

if (next instr == BGT)

n = 1st branch operand

i = 2nd branch operand

lab1 = 3rd branch operand

if (next branch == BR and next label == lab1)

if (next label == lab2)

return CODE\_STRING

CODE\_STRING =

<make any assignments for if and else blocks>

“if ( i <= ” + n + “) {”

<if block>

“} else{”

<else block>

“}”

Code Walkthroughs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Top Test Loop 1 |  |  |  |  |
| PAL |  |  |  |  |
|  | R0 | R1 | R7 | Comment |
| MOVE 0, R0 | 0 | ? | ? |  |
| MOVE 0, R1 | 0 | 0 | ? |  |
| MOVE 5, R7 | 0 | 0 | 5 |  |
| BGT R7, R1, itr | 0 | 0 | 5 | 5 > 0 so branch to "itr" |
| ADD R0, R1, R0 | 0 | 0 | 5 |  |
| INC R1 | 0 | 1 | 5 |  |
| BR test | 0 | 1 | 5 | branch to "test" |
| BGT R7, R1, itr | 0 | 1 | 5 | 5 > 1 so branch to "itr" |
| ADD R0, R1, R0 | 1 | 1 | 5 |  |
| INC R1 | 1 | 2 | 5 |  |
| BR test | 1 | 2 | 5 | branch to "test" |
| BGT R7, R1, itr | 1 | 2 | 5 | 5 > 2 so branch to "itr" |
| ADD R0, R1, R0 | 3 | 2 | 5 |  |
| INC R1 | 3 | 3 | 5 |  |
| BR test | 3 | 3 | 5 | branch to "test" |
| BGT R7, R1, itr | 3 | 3 | 5 | 5 > 3 so branch to "itr" |
| ADD R0, R1, R0 | 6 | 3 | 5 |  |
| INC R1 | 6 | 4 | 5 |  |
| BR test | 6 | 4 | 5 | branch to "test" |
| BGT R7, R1, itr | 6 | 4 | 5 | 5 > 4 so branch to "itr" |
| ADD R0, R1, R0 | 10 | 4 | 5 |  |
| INC R1 | 10 | 5 | 5 |  |
| BR test | 10 | 5 | 5 |  |
| BGT R7, R1, itr | 10 | 5 | 5 | 5 not > 5, fall through |
| BR fin | 10 | 5 | 5 | R0 = 0 + 1 + 2 + 3 + 4 as expected |
| END | 10 | 5 | 5 |  |
|  |  |  |  |  |
| Top Test Loop 1 |  |  |  |  |
| HLL |  |  |  |  |
|  | i | sum |  | Comment |
| int i = 0; | 0 | NA |  |  |
| int sum = 0; | 0 | 0 |  |  |
| while (i < 5){ | 0 | 0 |  | 0 < 5 so enter loop |
| sum = sum + i; | 0 | 0 |  |  |
| i = i + 1; | 1 | 0 |  |  |
| while (i < 5){ | 1 | 0 |  | 1 < 5 so enter loop |
| sum = sum + i; | 1 | 1 |  |  |
| i = i + 1; | 2 | 1 |  |  |
| while (i < 5){ | 2 | 1 |  | 2 < 5 so enter loop |
| sum = sum + i; | 2 | 3 |  |  |
| i = i + 1; | 3 | 3 |  |  |
| while (i < 5){ | 3 | 3 |  | 3 < 5 so enter loop |
| sum = sum + i; | 3 | 6 |  |  |
| i = i + 1; | 4 | 6 |  |  |
| while (i < 5){ | 4 | 6 |  | 4 < 5 so enter loop |
| sum = sum + i; | 4 | 10 |  |  |
| i = i + 1; | 5 | 10 |  |  |
| while (i < 5){ | 5 | 10 |  | 5 not < 5 don't enter loop |
| END | 5 | 10 |  | sum = 0 + 1 + 2 + 3 + 4 as expected |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Top Test Loop 2 |  |  |  |  |
| PAL |  |  |  |  |
|  | R0 | R1 | R7 | Comment |
| MOVE 0, R0 | 0 | ? | ? |  |
| MOVE 0, R1 | 0 | 0 | ? |  |
| MOVE 3, R7 | 0 | 0 | 3 |  |
| BGT R7, R1, itr | 0 | 0 | 3 | 3 >= 0 so branch to "itr" |
| ADD R0, R1, R0 | 0 | 0 | 3 |  |
| INC R1 | 0 | 1 | 3 |  |
| BR test | 0 | 1 | 3 | branch to "test" |
| BGT R7, R1, itr | 0 | 1 | 3 | 3 >= 0 so branch to "itr" |
| ADD R0, R1, R0 | 1 | 1 | 3 |  |
| INC R1 | 1 | 2 | 3 |  |
| BR test | 1 | 2 | 3 | branch to "test" |
| BGT R7, R1, itr | 1 | 2 | 3 | 3 >= 0 so branch to "itr" |
| ADD R0, R1, R0 | 3 | 2 | 3 |  |
| INC R1 | 3 | 3 | 3 |  |
| BR test | 3 | 3 | 3 | branch to "test" |
| BGT R7, R1, itr | 3 | 3 | 3 | 3 >= 0 so branch to "itr" |
| ADD R0, R1, R0 | 6 | 3 | 3 |  |
| INC R1 | 6 | 4 | 3 |  |
| BR test | 6 | 4 | 3 | 3 not >= 4, fall through |
| BR fin | 6 | 4 | 3 | R0 = 0 + 1 + 2 + 3 = 6 as expected |
| END |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Top Test Loop 2 |  |  |  |  |
| HLL |  |  |  |  |
|  | i | sum |  | Comment |
| int i = 0; | 0 | NA |  |  |
| int sum = 0; | 0 | 0 |  |  |
| while (i <= 3){ | 0 | 0 |  | 0 <= 3 so enter loop |
| sum = sum + i; | 0 | 0 |  |  |
| i = i + 1; | 1 | 0 |  |  |
| while (i <= 3){ | 1 | 0 |  | 1 <= 3 so enter loop |
| sum = sum + i; | 1 | 1 |  |  |
| i = i + 1; | 2 | 1 |  |  |
| while (i <= 3){ | 2 | 1 |  | 2 <= 3 so enter loop |
| sum = sum + i; | 2 | 3 |  |  |
| i = i + 1; | 3 | 3 |  |  |
| while (i <= 3){ | 3 | 3 |  | 3 <= 3 so enter loop |
| sum = sum + i; | 3 | 6 |  |  |
| i = i + 1; | 4 | 6 |  |  |
| while (i <= 3){ | 4 | 6 |  | 4 not <= 3; don't enter loop |
| END | 4 | 6 |  | sum = 0 + 1 + 2 + 3 as expected |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Top Test Loop 1 |  |  |  |  |
| PAL |  |  |  |  |
|  | R0 | R1 | R7 | Comment |
| MOVE 3, R0 | 3 | ? | ? |  |
| MOVE 2, R1 | 3 | 2 | ? |  |
| MOVE 4, R7 | 3 | 2 | 4 |  |
| BGT R1, R7, end | 3 | 2 | 4 | 2 not > 4 so fall through |
| BEQ R1, R7, end | 3 | 2 | 4 | 2 != 4 so fall through |
| MOVE 0, R0 | 0 | 2 | 4 |  |
| END | 0 | 2 | 4 | R0 = 0 as expected |
|  |  |  |  |  |
|  |  |  |  |  |
| Top Test Loop 1 |  |  |  |  |
| HLL |  |  |  |  |
|  | i | sum |  | Comment |
| int i = 2; | 2 | NA |  |  |
| int sum = 3; | 2 | 3 |  |  |
| if (i < 4){ | 2 | 3 |  | 2 < 3 so enter if block |
| sum = 0; | 2 | 0 |  |  |
| END |  |  |  | sum = 0 as expected |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Top Test Loop 2 |  |  |  |  |
| PAL |  |  |  |  |
|  | R0 | R1 | R7 | Comment |
| MOVE 2, R1 | ? | 2 | ? |  |
| MOVE 4, R7 | ? | 2 | 4 |  |
| BGT R1, R7, else | ? | 2 | 4 | 2 not > 4 so fall through |
| MOVE 0, R0 | 0 | 1 | 4 |  |
| BR end | 0 | 1 | 4 |  |
| END | 0 | 1 | 4 | R0 = 0 as expected |
|  |  |  |  |  |
|  |  |  |  |  |
| Top Test Loop 1 |  |  |  |  |
| HLL |  |  |  |  |
|  | i | sum |  | Comment |
| int i = 2; | 2 | NA |  |  |
| int sum; | 2 | ? |  |  |
| if (i <= 4){ | 2 | ? |  | 2 <= 4 so enter if block |
| sum = 0; | 2 | 0 |  |  |
| END | 2 | 0 |  | sum = 0 as expected |