Controlstructuretottpasm

Module 0303: "Compiling" control structure to TTPASM
About this module
Purpose of this module
C control structures
Condition statements
Pre-checking loop
Post-checking loop
Conditional goto and boolean operators
not
or
and
Simple reductions
Comparisons
General compare
About nesting and labels
Indentation
More useful names
Sequential numbers
Nested constructs
Boolean operator reduction transformations
A longer example

1. About this module

Prerequisites:
Objectives: This module explains how C control structures are compiled to assembly language code.

1. Purpose of this module

This module contains the instructions to convert a C program into assembly language code in a step-by-step manner. While this module is not intended for people who want to write compilers, it is certainly useful for that purpose.

1. C control structures

This section introduces simple transformations that turn regular C code into C code that has no control structures except for "conditional goto" constructs.

1. 1 Condition statements

First, it is important to understand the C syntax of conditional statements without braces.
if (a¡b) a++; b–; The above code is actually the same as the following code.
if (a¡b) a++; b–; This is because, after the parenthesized condition, C++ expects a single statement to specify the then-statement. a++;, by itself, is a statement. After the then-statement, there are two alternatives.
the else keyword is encountered: then C++ expects another statement as the else-statement.
the else keyword is not encountered: then C++ ends the conditional statement , treating whatever that is following as a continuation after the conditional statement.
Block statements (braces) are highly recommended in normal C++ coding because of this.
However, in the context of this module, the intention is "flattening" the structured code. As a result, the intention is to get rid of block statements one step at a time.
if (c) blk1; // blk1 is a placeholder for the then-statement else blk2; // blk2 is a placeholder for the else-statement Translates to
if (!c) goto L1; blk1; goto L2; L1: blk2; L2: If there is no else, then
if (c) blk1; translates to
if (!c) goto L1; blk1; L1: This is because the following code optimizes to no code:
goto L2; L2:

1. 2 Pre-checking loop

while (c) blk1; // blk1 is a placeholder for the code in repetition Translates to

if (!c) goto L2; blk1; goto L1; L2:

1. 3 Post-checking loop

do blk1; // blk1 is a placeholder for the code in repetition while (c); Translates to
L1:
blk1; if (c) goto L1;

1. Conditional goto and boolean operators

From the previous section, if a condition involves negation (not), disjunction (or) or conjunction (and), we must first take care of those operators. This section discusses how those operators, while used in conditional goto statements, can be transformed.

1. 1 not

if (!c) goto L1; Transforms to
if (c) goto L2; goto L1; L2:

1. 2 or

if (c —— d) goto L1; becomes
if (c) goto L1; if (d) goto L1;

1. 3 and

if (c d) goto L1; becomes
if (!c) goto L2; if (d) goto L1; L2:

1. 4 Simple reductions

Although the transformations will work in all cases, they are not always necessary.
For example, !(x ¿= y) is (x ¡ y).
Also, (x ¡ y) —— (x ¿ y) is (x != y).
Using algebra to reduce boolean operators saves a bit of control structure transformations, and can lead to more efficient code.

1. Comparisons

With the previous sections, regular C control structures are reduced to conditional goto statements that do not use any boolean operators. This means that the conditional goto statements must only contain comparison operators.
Because the toy processor can only confirm “less than” and “equal to” using a single instruction, we need to translate all comparison operators to these two operators (with the help of some logical operators):
is just that
is reversed to
is
is
is just that
is

1. 1 General compare

if (x r y) goto L1; is a generalized conditional goto statement. x r y is the abstraction of x is r y. In an actual program, x is an expression, r is a comparison operator (the toy processor is confined to equal-to and less-than), and y is another expression.
For now, however, let us assume that x and y are registers.
This conditional goto statement transforms to the following pseudo assembly instructions:
cmp x , y jri L x ¡ y x ¿ y y ¡ x x y (x ¡ y)(x = y) x y (y ¡ x)(y = x) x = y x y ¬(x = y) For example, let us assume the C code is as follows (x and y are unsigned):
if (x ¡ y) goto L1; Then the matching assembly code is as follows:
cmp x,y jci L Because the toy processor can only compare registers, compare to constants need to first load the constant to a register using the ldi instruction.

1. About nesting and labels

Although the previous sections use the generic labels L0, L1 and L2, your program should use more useful label names. This section recommends a structured method to name the labels.

1. 1 Indentation

While most assembly language programs do not make use of indentation, it makes sense to use indentation the same way as in a high level programming language. This is because indentation reflects the structure of an algorithm, and assembly language programs can be well structured like a high level programming language.

1. 2 More useful names

Use more useful names, such as loopBegin, loopEnd, else, endIf and etc. whenever it is

1. 3 Sequential numbers

Constructs of the same nesting level should use sequential numbers as a suffix to differentiate from each other. For example,

if (x ¿= y) x++; if (y ¿= x) y++; translates to

if (x ¡ y) goto endIf1; x++; endIf1: if (y ¡ x) goto endIf2; y++; endIf2:

1. 4 Nested constructs

Nested constructs should technically use a suffix or prefix notation to indicate how they are nested. A prefix (where the deeper structure is named first) is easier to read, as the first part of the label shows the meaning of the label. However, a suffix is more natural, especially to people who are used to C/C++ programming.

Using the suffix method,

while (x ¡ 3) if (y == x) z++; x++;

transforms to

beginLoop1: if (x ¿= 3) goto endLoop1; if (y != x) goto $loop1_endIf1; z++; loop1_endIf1 : x++; gotobeginLoop1; endLoop1 :$ $Thefollowingrepresentsthesamelogic, butusingtheprefixmethod(nestedstructurenamed$ first):

beginLoop1: if (x ¿= 3) goto endLoop1; if (y != x) goto $endIf1_loop1; z++; endIf1_loop1 : x++; gotobeginLoop1; endLoop1 :$ $Notethatthe\backslash sequencenumber"ofeachnestingcanrestartfrom1becausetheprefixor$ suffix of the parent structure makes it unique.

An alternative (to more experienced programmers) is to simply use sequential numbers for each construct. As one construct is converted, simply pick the next available sequential number. This method requires that the programmer only perform the translation of one construct at a time. Furthermore, the label names no longer reflect the nested structure of the code.

1. 5 Boolean operator reduction transformations

Boolean operator reduction transformation are low level transformations. As such, the labels generated as a result should simply add a suffix to the original labels. For example,

if ((x == y) (x ¡ 0)) goto $endIf2_loop5; translatesto$

if (!(x == y)) goto $endIf2loop51$; if (x ¡ 0) goto $endIf2_loop5; endIf2loop51 :$

A longer example

Here is a long(er) example to illustrate the concepts introduced in this module. To make the

"diff " between revisions stand out more, you can copy-and-paste each section of code into

files. Then, use vimdiff or a similar visualized diff command to have the changes

highlighted. If you have XWindow running, there is a whole list of these tools: tkdiff, xxdiff,

diffuse, gvimdiff, etc.

Let's say we want to convert the following code into assembly code:

int x,y,z; int main (void) x = 5 ; z = 0 ; while (x ¿ 0) y = 0 ; while (y ¡ 5) if (((x ¿ 2) (x != y)) —— (y == 1)) z = z + x; This program no particular "meaning" as in useful behavior, its only purpose is to serve as an

example.

First, I can comment out the entire program, and start to work on it little-by-little. This can be

done with a single command in vi, there is a reason to learn how to use it!

// int x,y,z; // int main(void) // // x = 5; // z = 0; // while (x ¿ 0) // // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // // z = z + x; // // // // Next, let us declare the global variables:

// int x,y,z; // int main(void) // // x = 5; // z = 0; // while (x ¿ 0) // // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // // z = z + x; // // // // halt x: byte 0 y: byte 0 z: byte 0 The initialization portion is also quite easy.

// int x,y,z; // int main(void) // // x = 5; ldi a,x ldi b, 5 st (a),b // z = 0; ldi a,z ldi b, 0 ; st (a),b // while (x ¿ 0) // // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // // z = z + x; //

//

//

//

halt x: byte 0 y: byte 0 z: byte 0 Adding x to z requires an intermediate register, but otherwise it is also straightforward.

// // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // // z = z + x; ldi a,x // a = x ld a,(a) // a = x ldi b,z // b = z ld b,(b) // b = z add a,b // a = x+z ldi b,z // b = z st (b),a // z = x+z // // //

//

halt x: byte 0 y: byte 0 z: byte 0 Now let us take care of the inner-most conditional statement. First, we reduce to "ugly"

conditional statement code:

// int x,y,z; // int main(void) // // x = 5; ldi a,x ldi b, 5 st (a),b // z = 0; ldi a,z ldi b, 0 ; st (a),b // while (x ¿ 0) // // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // if (!(((x ¿ 2) (x != y)) —— (y == 1))) goto while1*while1*endif1; // // z = z + x; ldi a,x // a = x ld a,(a) // a = x ldi b,z // b = z ld b,(b) // b = z add a,b // a = x+z ldi b,z // b = z st (b),a // z = x+z // DeMorgan's law allows us to simplify the negated disjunction a little. DeMorgan's law says

.

while1*while1*endif1: // // // halt x: byte 0 y: byte 0 z: byte 0 // int x,y,z; // int main(void) // // x = 5; ldi a,x ldi b, 5 st (a),b // z = 0; ldi a,z ldi b, 0 ; st (a),b // while (x ¿ 0) // // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // if (!(((x ¿ 2) (x != y)) —— (y == 1))) goto while1*while1*endif1; // if ((!((x ¿ 2) (x != y)) !(y == 1))) goto while1*while1*endif1; // // z = z + x; ldi a,x // a = x ld a,(a) // a = x ldi b,z // b = z ld b,(b) // b = z add a,b // a = x+z X + Y = X Y Now is a good time to take care of the outer conjunction. We can reduce the conjunction by

converting the conditional-goto into multiple statements:

ldi b,z // b = z st (b),a // z = x+z // while1*while1*endif1: // // // halt x: byte 0 y: byte 0 z: byte 0 // int x,y,z; // int main(void) // // x = 5; ldi a,x ldi b, 5 st (a),b // z = 0; ldi a,z ldi b, 0 ; st (a),b // while (x ¿ 0) // // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // if (!(((x ¿ 2) (x != y)) —— (y == 1))) goto while1*while1*endif1; // if ((!((x ¿ 2) (x != y)) !(y == 1))) goto while1*while1*endif1; // if ((x ¿ 2) (x != y)) goto while1*while1*if1; // if (!(y == 1)) goto while1*while1*endif1; // // z = z + x; We can now take care of the simple transformations:

while1*while1*if1: ldi a,x // a = x ld a,(a) // a = x ldi b,z // b = z ld b,(b) // b = z add a,b // a = x+z ldi b,z // b = z st (b),a // z = x+z // while1*while1*endif1: // // // halt x: byte 0 y: byte 0 z: byte 0 // int x,y,z; // int main(void) // // x = 5; ldi a,x ldi b, 5 st (a),b // z = 0; ldi a,z ldi b, 0 ; st (a),b // while (x ¿ 0) // // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // if (!(((x ¿ 2) (x != y)) —— (y == 1))) goto while1*while1*endif1; // if ((!((x ¿ 2) (x != y)) !(y == 1))) goto while1*while1*endif1; Now we translate the other conjunction.

// if ((x ¿ 2) (x != y)) goto while1*while1*if1; // if (!(y == 1)) goto while1*while1*endif1; ldi a,y // a = y ld a,(a) // a = y ldi b, 1 // b = 1 cmp a,b // a-b jzi while1*while1*if jmpi while1*while1*endif // // z = z + x; while1*while1*if1: ldi a,x // a = x ld a,(a) // a = x ldi b,z // b = z ld b,(b) // b = z add a,b // a = x+z ldi b,z // b = z st (b),a // z = x+z // while1*while1*endif1: // // // halt x: byte 0 y: byte 0 z: byte 0 // int x,y,z; // int main(void) // // x = 5; ldi a,x ldi b, 5 st (a),b // z = 0; ldi a,z ldi b, 0 ; st (a),b // while (x ¿ 0) // // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // if (!(((x ¿ 2) (x != y)) —— (y == 1))) goto while1*while1*endif1; // if ((!((x ¿ 2) (x != y)) !(y == 1))) goto while1*while1*endif1; // if ((x ¿ 2) (x != y)) goto while1*while1*if1; // if (!(x ¿ 2)) goto while1*while1*if1$_l$1; $//if((x < 2)||(x == 2))gotowhile1while1if1_l1; //if(x! = y)gotowhile1while1if1; while1while1if1_l1 : //if(!(y == 1))gotowhile1while1endif1; ldia, y//a = ylda, (a)//a = yldib, 1//b = 1cmpa, b//a-bjziwhile1while1ifjmpiwhile1while1endif////z = z + x; while1while1if1 : ldia, x//a = xlda, (a)//a = xldib, z//////haltx : byte0y : byte0z : byte0$

Then we turn the conditional goto into native assembly language code:

// int x,y,z; // int main(void) // // x = 5; ldi a,x ldi b, 5 st (a),b // z = 0; ldi a,z ldi b, 0 ; st (a),b // while (x ¿ 0) // // y = 0; // while (y ¡ 5) // // if (((x ¿ 2) (x != y)) —— (y == 1)) // if (!(((x ¿ 2) (x != y)) —— (y == 1))) goto while1*while1*endif1; // if ((!((x ¿ 2) (x != y)) !(y == 1))) goto while1*while1*endif1; // if ((x ¿ 2) (x != y)) goto while1*while1*if1; // if (!(x ¿ 2)) goto while1*while1*if1$_l$1; $//if((x < 2)||(x == 2))gotowhile1while1if1_l1; ldia, x//a = xlda, (a)//a = xldib, 2//b = 2cmpa, b//a-bjciwhile1while1if1_l1jziwhile1while1if1_l1//if(x! = y)gotowhile1while1if1; ldia, x$
$xlda, (a)//a = xldib, y//b = yldb, (b)//b = ycmpa, b//a-bjziwhile1while1if1_l1jmpwhile1while1if1while1while1if1_l1 :$
$//if(!(y == 1))gotowhile1while1endif1; ldia, y//a = ylda, (a)//a = yNowisagoodtimetotestthecodewithdifferentcombinat$

Once the conditional statement is checked, we can work on the inner loop.

Note the use of indentation makes it (relatively) easy to locate the beginning and the end of the

inner while loop. Next, we translate the inner loop to assembly code:

This is a good time to test the inner loop. Once the inner loop is tested (to a certain extent), it is

time to complete the outer loop. I am combining all the steps, as it gets boring after a while!

And that's it!

ldi b, 1 // b = 1 cmp a,b // a-b jzi while1*while1*if1 jmpi while1*while1*endif1 // // z = z + x; while1*while1*if1: ldi a,x // a = x ld a,(a) // a = x ldi b,z // b = z ld b,(b) // b = z add a,b // a = x+z ldi b,z // b = z st (b),a // z = x+z // while1*while1*endif1: // // // halt x: byte 0 y: byte 0 z: byte 0