Target Code Generation

A Basic Block is a sequence of statements where the flow of control enters at the beginning of the block and leaves at the end (bottom) of the block. It only has sequential statements no branches and no labels (except to the first statement of the basic block).

Structure Preserving Transformations on a Basic Block.

1. Common-Subexpression Elimination

```
ADD B, C, A  ; A = B + C
SUB A, D, B  ; B = A - D
ADD B, C, C  ; C = B + C
SUB  A, D, D ; D = A – D
```

Transform

```
ADD B, C, A  ; A = B + C
SUB A, D, B  ; B = A - D
MOV C, A     ; C = A
MOV D, B     ; D = B
```

2. Dead-Code Elimination

Compute a value that is never used.

```
ADD A, B, D
STR D, LABEL_D
ADD A, B, C  ; Value C is never used, A + B = C
```

Transform

```
ADD A, B, D
STR D, LABEL_D
```

3. Renaming Temporary Variables

```
ADD A, B, t72
ADD C, t72, t100
```

Transform

```
ADD A, B, t200
ADD C, t200, t201
```

4. Interchange of two independent yet adjacent statements.
   No Data Dependencies – operands independent
   No Structural Dependencies – no shared resources
   No Control Dependencies – basic block can not have branches

   ADD A, B, C
   MOV J, K

Transform
   MOV J, K
   ADD A, B, C

Simple Target Code Generation Strategy

For each intermediate statement, generate a set of target statements while remembering if any operands are currently in registers. Leave results in registers as long as possible storing results only:

1) When a register must be reused,
2) When we encounter a:
   a) Procedure call or jump
      i) We leave a basic block
   b) Statement with a label
      i) We can enter a basic block at a label

We store (register to memory) when we exits a basic block (or entry a new basic block if you prefer) because we should not assume that other basic blocks will not alter the contents of registers.

Paranoid Target Code Generation Strategy

For each Intermediate Code statement, generate a set of target statements while always reading and writing data to memory. While this conflicts with our Simple Target Code Strategy of leaving results in registers as long as possible it will not invalidate your compiler "Only Makes It Run Slower".

Two Functions to aid in Target Code Generation.

1) getRegister()
Will return the next available register when invoked. It uses register descriptors to determine if a register is free (unused).

Register Descriptors keep track of what is currently in each register. Not the value, that is only known at run-time. What symbol data is currently associated with the register.

Register # → {symbol_id }

It is *possible to have more than one symbolic data item associated with a register*.

```
IC:   ADD A, B, C  ; A + B → C
      MOV C, D     ; C → D

TC:   LDR R3, A          ; Load R3 with value @ A
      LDR R5, B          ; Load R5 with value @ B
      ADD R5, R3         ; A + B → R5
      STR   R5, D        ; Store R5 @ D
```

R5 → {C, D} Register Descriptor with two elements! Doesn't have B because it was overridden with C

2) getLocation(symbol id)
Will return the location(s) of where a symbolic data item can be found at run-time.
Valid Locations are:
register(register#) – the symbol id is already in a register
stack(-offset) – the symbol id is located in the current activation record
heap(+offset) – the symbol id is for some attribute allocated by new
memory(Address) – the symbol id is global

A *symbolic data item can be in more than one location*. You will need to keep up with more than one location. Consider the following example:
If variable t23 is on the stack at some displacement and in registers R3 and R12.
If variable G is on the stack at some displacement and in registers R12.

t23 → {register(R3), register(R12), stack(-16) }
g → {register(R12), stack(-12) }

Because t23 and g are both in register R12 (e.g., MOV t23, g or MOV g, t23), the register descriptor for R12 must show both of these values:
R12 → {t23, g}

What happens when Target Code Generation encounters an Intermediate Instruction:

Overview

1. Save Register Data to Memory if Beginning or End of a Basic Block
2. Use getRegister() to find an unused Register for the result of the operation
    a. If no unused Registers then Write Data to Memory to Free up Registers
        i. Generate Target Code to Free Registers
    b. Repeat Step 2
3. Use getLocation(…) to find memory location of input operands
    a. Generate Target Code to get input operands into a Register if necessary
4. Check if a operand must be moved to the result register
    a. Converting from three address operand to only two address operand
    b. Generate Target Code to move Operand to Result Register
5. Generate Target Code for Operation

ADD A, B, C ; C = A + B

1) If the instruction is the beginning or end of a new basic block (aka has a Label), then
    a) All registers should be saved to memory locations
    b) All registers should be freed (register descriptors must reflect this)
2) Invoke getRegister to determine where to place C, the result of the operation. On some operations you may desire to not place the result into a register and to place the result directly into memory. In this case use getLocation on the result as well.
    a) If no registers are available:
        i) Generate Target code to free up one or more registers, move registers to memory or stack locations.
        ii) Update the Register Descriptors to reflect that registers no longer contain information about a symbolic data element.
        iii) Repeat the process to determine where to place C.
3) Invoke getLocation to determine the current location of each operand A and B.
4) Generate Target code based upon the results of the getRegister, getLocation and the intermediate code.
    a) Note: Because our target code only have two operands, one of the operands (A or B) must be moved to C before the Add operation is executed.
5) Generate an add instruction.

Intermediate Code (IC) to Target Code (TC) Conversion

When converting Intermediate Code to Target Code two basic processes will have to be performed.

1. Convert the Intermediate Code Operator to one or more Target Code Operators.
2. Convert the symbolic Intermediate Code Operands to physical Target Code locations.

Converting IC Operands to TC Operands

The operands of any Intermediate Code instruction (e.g., ADD A, B, C) can map to five physical locations in Target Code:

Register	If the operand has already been loaded from some place in memory.
Global Memory	If the operand was defined as a global variable. Kxi doesn't support global variables, however a literal is handled like a global variable.
Run-Time Stack	If the operand is a passed-parameter of a function, a local variable of a function or a temporary variable of a function.
Heap	If the operand is an instance variable or an array element. Arrays and instances can not be created on the Stack in kxi.
Instruction	If the operand is immediate (e.g., ADI A, #42).

How to Access the Physical Location of an integer Operand Y, when the location of Y is:

- Register, just use the name of the register that contains data for the operand Y

  register(RS) = getLocation(Y)

  ```
  ADD   RD, RS        ; Add Y to register RD
  ```

- Global Memory, use the label (Y) that is associated with the location in memory for the operand Y

  memory(Label_Y) = getLocation(Y)

  ```
  LDR   RD, Label_Y        ; Load register RD with the contents of location Y
  ```

- Run-Time Stack, use the displacement (D) that is an offset from the Frame Pointer (FP) to the location of the operand Y. Note: D will always be a negative number!

  stack(D) = getLocation(Y)

  Without Index Address Mode

  ```
  MOV   RT, FP        ; Move FP to register RT
  ADI   RT, #D        ; Add D which is negative to RT giving the address of Y
  LDR   RS, (RT)      ; Load Y to register RS, using register indirect addressing
  ```

  With Index Address Mode

  ```
  LDR   RS, (SP + D)  ; Load Y to Register RS
  ```

- Heap, use the displacement (D) that is an offset from the referenced object (this) to the location of the data item Y. D will always be a positive number! Note: The referenced object (this) will be on the stack or in a register; it is not on the heap from the perspective of getLocation (X would never be a global as shown).
  heap(D) = getLocation(Y)

  Without Index Address Mode

```
LDR   RT, X        ; Load RT with the address of this
                   ; The more likely case is we are accessing Y
                   ; relative to this which is always at the same place
                   ; on the stack
ADI   RT, #D       ; Add D to RT giving the address of Y
LDR   RS, (RT)     ; Load Y to register RS, using register indirect addressing
```

  With Index Address Mode

```
LDR   RT, X        ; Load RT with the address of this
LDR   RS, (RT + D) ; Load Y to source Register
```

  Note: You must be sure to handle problems like the following:
  When a public data item is being referenced from an object (X.Y) vs. a local instance variable (Y) being referenced, they are both on the heap, yet they are not the same.
  - A temp variable is created for the object reference and it will contain an address (RT + D in the above example), which points to the data item on the heap.
  - A local instance variable is the data item, but we must use a displacement relative to the this pointer (D in the example above) to create an address on the heap.

  Information that is (or should be) stored in your Symbol Table will help you distinguish between this and other similar situations. As you will soon discover virtually every data item in kxi must be accessed by combining a base address ± a displacement. In target code this means indirect addressing must be used repeatedly to get or store the value of a data item.
  - Variables on the run-time stack (local variables, passed parameters and temp variables) are always accessed from the current frame pointer – a displacement.
  - Public instance variables are always accessed from the reference object + a displacement to the instance variable.
  - Local instance variables are always accessed from the "this" pointer which is on the stack + a displacement to the instance variable.

Note: For the following examples the operands of each Intermediate Instruction will map to the following Target Code locations:

Operand A → Register (Ra)
Operand B → Register (Rb)
Operand C → Register (Rc)

Mathematical Instructions Conversions

Note: First Target Code (TC) example is generally the simplest and does not involve leaving data in register. The Second TC example is generally move complex and leaves data in register for later use.

```
IC:  ADD  A, B, C    ; A + B → C
TC:  ADD  Ra, Rb     ; Will over write Ra
Or
     MOV Rc, Ra      ; Operand Reg. to Result Reg.
     ADD Rc, Rb      ; Ra, Rb still have original data

IC:  ADI  A, #, C    ; A + # → C
TC:  ADI  Ra, #
Or
     MOV  Rc, Ra
     ADI  Rc, #

IC:  SUB  A, B, C    ; A – B → C
TC:  SUB  Ra, Rb
Or
     MOV  Rc, Ra
     SUB  Rc, Rb
```

```
IC:  MUL  A, B, C   ; A * B → C
TC:  MUL  Ra, Rb
Or
     MOV  Rc, Ra
     MUL  Rc, Rb

IC:  DIV  A, B, C   ; A / B → C
TC:  DIV  Ra, Rb
Or
     MOV  Rc, Ra
     DIV  Rc, Rb

IC:  MOD  A, B, C   ; A % B → C – NOT USED!!!
TC:  MOD  Ra, Rb
Or
     MOV  Rc, Ra
     MOD  Rc, Rb
```

Boolean Instructions Conversions

Note: TRUE is 1 and FALSE is 0
Note: Register R0 always contains a 0
Note: Register R1 always contains a 1
Note: There are two possible implementation for the CMP command

1st CMP Rd, Rs is the same as SUB Rd, Rs, thus After CMP
- Rd is zero if Rd = Rs
- Rd is negative if Rd < Rs
- Rd is positive if Rd > Rs

2nd CMP Rd, Rs is a true compare operator, thus After CMP
- Rd is 0 if Rd = Rs
- Rd is –1 if Rd < Rs
- Rd is +1 if Rd > Rs

```
IC:   EQ    A, B, C    ; A == B → C
TC:         MOV   Rc, Ra
            CMP   Rc, Rb
            BRZ   Rc, L3     ; A == B GOTO L3
            MOV   Rc, R0     ; Set FALSE
            JMP   L4
      L3:   MOV   Rc, R1     ; Set TRUE
      L4:                    ; Next Statement

IC:   LT    A, B, C    ; A < B → C
TC:         MOV   Rc, Ra
            CMP   Rc, Rb
            BLT   Rc, L3     ; A < B GOTO L3
            MOV   Rc, R0     ; Set FALSE
            JMP   L4
      L3:   MOV   Rc, R1     ; Set TRUE
      L4:                    ; Next Statement

IC:   GT    A, B, C    ; A > B → C
TC:         MOV   Rc, Ra
            CMP   Rc, Rb
            BGT   Rc, L3     ; A > B GOTO L3
            MOV   Rc, R0     ; Set FALSE
            JMP   L4
      L3:   MOV   Rc, R1     ; Set TRUE
      L4:                    ; Next Statement
```

```
IC:   NE    A, B, C    ; A != B → C
TC:         MOV   Rc, Ra
            CMP   Rc, Rb
            BNZ   Rc, L3      ; A != B GOTO L3
            MOV   Rc, R0      ; Set FALSE
            JMP   L4
      L3:   MOV   Rc, R1      ; Set TRUE
      L4:                     ; Next Statement

IC:   LE    A, B, C    ; A <= B → C
TC:         MOV   Rc, Ra      ; Test A < B
            CMP   Rc, Rb
            BLT   Rc, L3      ; A < B GOTO L3
            MOV   Rc, Ra      ; Test A == B
            CMP   Rc, Rb
            BRZ   Rc, L3      ; A == B GOTO L3
            MOV   Rc, R0      ; Set FALSE
            JMP   L4
      L3:   MOV   Rc, R1      ; Set TRUE
      L4:                     ; Next Statement


IC:   GE    A, B, C    ; A >= B → C
TC:         MOV   Rc, Ra      ; Test A > B
            CMP   Rc, Rb
            BGT   Rc, L3      ; A > B GOTO L3
            MOV   Rc, Ra      ; Test A == B
            CMP   Rc, Rb
            BRZ   Rc, L3      ; A == B GOTO L3
            MOV   Rc, R0      ; Set FALSE
            JMP   L4
      L3:   MOV   Rc, R1      ; Set TRUE
      L4:                     ; Next Statement
```

Logical Instructions

Note: TRUE is 1 and FALSE is 0
Note: Register R0 always contains a 0
Note: Register R1 always contains a 1

```
IC:   AND   A, B, C    ; A && B → C
TC:   MOV   Rc, Ra
      AND   Rc, Rb

IC:   OR    A, B, C    ; A || B → C
TC:   MOV   Rc, Ra
      OR    Rc, Rb
```

Control Flow Instructions

Note: TRUE is 1 and FALSE is 0
Note: Register R0 always contains a 0
Note: Register R1 always contains a 1

```
IC:   BF    A, LABEL   ; A == FALSE GOTO LABEL
TC:   BRZ   A, LABEL

IC:   BT    A, LABEL   ; A == TRUE GOTO LABEL
TC:   BNZ   A, LABEL

IC:   JMP   LABEL      ; GOTO LABEL
TC:   JMP   LABEL
```

Run-Time Stack Instructions

```
IC:   PUSH A           ; Push A on Stack
TC:   STR Ra, (SP)     ; Push A on Stack; A in Ra
      ADI SP, #-4      ; Modify Stack Pointer

IC:   POP  A           ; Pop the top of the Stack into A
TC:   ADI  SP, #4      ; Modify Stack Pointer
      LDR  Ra, (SP)    ; Pop the Stack

IC:   PEEK A           ; Peek the top of the Stack into A
TC:   MOV R5, SP       ; though we implement it different
      ADI R5, #4
      LDR  Ra, (R5)    ; Peek the Stack
```

Function Invocation Instructions

```
IC:  FRAME F, X      ; Create a frame for function F
TC:  See the document provided on Activation Records
```

- Test for overflow (SP < SL) using the space needed for the Frame
- Save off current FP in a Register, this will be the PFP
- Point at Current Activation Record (FP = SP)
- Adjust Stack Pointer for Return Address
  - Not Stored Yet (See CALL F)
- Store PFP to Top of Stack
- Adjust Stack Pointer for PFP
- Store this pointer to Top of Stack
- Adjust Stack Pointer for this

```
IC:  CALL F          ; Invoke function F
TC:  MOV Ra, PC      ; Get PC
     ADI Ra, #       ; Compute Return Address
     STR Ra, (FP)    ; Store Return Address
     JMP F           ; Jump to function F

IC:  RTN             ; Return from a Function
TC:  See the document provided on Activation Records
```

- De-allocate Current Activation Record
- Test for Underflow (SP > SB)
- Set Previous Frame to Current Frame and Return
  - Load Return Address from the Frame
  - Load PFP from the Frame
    - Set FP = PFP
  - Jump using JMR to Return Address

```
IC:  RETURN A        ; Return A from a Function
TC:  See the document provided on Activation Records
```

- De-allocate Current Activation Record
- Test for Underflow (SP > SB)
- Set Previous Frame to Current Frame and Return A
  - Load Return Address from Frame
  - Load PFP from the Frame
  - Store Return Value on Top of Stack
  - Jump using JMR to Address in Register

```
IC:  FUNC F          ; Initialize Function F
TC:  ADI  SP, #      ; Allocate # bytes of space on
                     ; the Stack for Temporary and
                     ; Local Variables. # is a negative
                     ; number.
```

Memory Allocation Instructions

```
IC:  NEWI #, A       ; Allocate # bytes on the heap and
                     ; place the starting address in A
TC:  LDR  Rc, FREE   ; Load address of free heap
     MOV  Ra, Rc     ; Save address in register Ra
     ADI  Rc, #      ; Inc free heap by # bytes
     STR  Rc, FREE   ; Update FREE

IC:  NEW B, A        ; Allocate the number of bytes in
                     ; variable B on the heap and
                     ; place the starting address in A
TC:  LDR  Rc, FREE
     MOV  Ra, Rc
     ADD  Rc, Rb     ; Inc free heap by Rb bytes
     STR  Rc, FREE
```

Miscellaneous (Other) Instructions

```
IC:  MOV  A, B       ; A → B
TC:  MOV  B, A

IC:  MOVI A, #       ; # → A
TC:  SUB  Ra, Ra
     ADI  Ra, #

IC:  WRITE A         ; Write the value of A
This code can vary from VM to VM, however if your
traps uses Rc as your source register
TC:  MOV  Rc, Ra
     TRP  1          ; Write as an Integer
Or
     MOV  Rc, Ra
     TRP  3          ; Write as a Char

IC:  READ A          ; Read from the screen into A
                     This code can vary from VM to VM,
                     however if your traps uses Rc as
                     your source register
TC:  TRP  2          ; Read an Integer
     MOV  Ra, Rc
Or
```

```
     TRP  4          ; Read a Char
     MOV  Ra, Rc

IC:  REF A, B, C     ; Using A as a Base Address add to
                     ; it the offset to B and assign
                     ; the address to C.
TC:  MOV  Rc, Ra
     ADD  Rc, Rb
```

Sample kxi Conversion

```
int go(int i) {
     int x = 0;
     Cat c = new Cat(7, i);

     x = c.age() * i;

     if (x > 100)
          return -1;
     else
          return x;
}
```

Offset	*go Frame*
40	t8
36	t7
32	t6
28	t5
24	t4
20	c
16	x
12	i
8	this
4	PFP
0	Return Address

Offset	*Cat Frame*
16	7
12	i
8	this
4	PFP
0	Return Address

Kxi Code	Intermediate Code	Target Code

`int go(int i) {`	`Dog_Go: FUNC Dog_Go`	`Dog_Go: ADI SP, #-28 ; x,c & 5 temps`
		`MOV R5, SP ; Test Overflow`
		`CMP R5, SL`
		`BLT R5, OVERFLOW`

`int x = 0;`	`MOV x, C0`	`MOV R5, FP`
		`ADI R5, #-16 ; address of x`
		`STR R0, (R5)`

`Cat c = new Cat(7, i);`	`NEWI 12, t4`	`LDR R6, FREE ; Get this`
		`MOV R7, R6 ; this → R7`
		`ADI R6, #12`
		`STR R6, FREE`
		`MOV R5, FP`
		`ADI R5, #-24`
		`STR R7, (R5) ; Store t4`

	FRAME Cat, t4	*MOV R5, SP ; Test Overflow* *ADI R5, #-20 ; rtn,pfp,this,i,7* *CMP R5, SL* *BLT R5, OVERFLOW* *MOV R3, FP ; Old Frame* *MOV FP, SP ; New Frame* *ADI SP, #-4 ; PFP* *STR R3, (SP) ; Set PFP* *ADI SP, #-4* *STR R7, (SP) ; Set this on Stack* *ADI SP, #-4*
	PUSH i	*MOV R5, R3 ; Old Frame R3* *ADI R5, #-12 ; Address of i* *LDR R6, (R5) ; i in R6* *STR R6, (SP) ; i on stack* *ADI SP, #-4*
	PUSH c7	*LDR R6, c7 ; c7 in R6* *STR R6, (SP) ; c7 on stack* *ADI SP, #-4*
	CALL Cat	MOV R6, PC ADI R6, #6 ; Compute rtn addr STR R6, (FP) ; Set rtn addr JMP Cat
	PEEK t5	LDR R6, (SP) ; this → R6
	MOV c, t5	MOV R3, FP ADI R3, #-20 ; address of c STR R6, (R3) ; this → c

x = c.age() * i;	FRAME Cat_Age, c
	Call Cat_Age
	PEEK t6
	
	MUL t6, i, t7
	
	MOV x, t7
		; left for you to do
		; left for you to do
		; left for you to do
		
		MOV R5, FP
		ADI R5, #-12 ; address of i
		LDR R7, (R5) ; i → R7
		MOV R5, FP
		ADI R5, #-28 ; address of t6
		LDR R6, (R5) ; t6 → R6
		MUL R6, R7
		MOV R5, FP
		ADI R5, #-32
		STR R6, (R5)
		
		MOV R5, FP
		ADI R5, #-32 ; address of t7
		LDR R6, (R5) ; t7 → R6
		MOV R5, FP
		ADI R5, #-16 ; address of x
		STR R6, (R5)

if (x > 100) {	GT x, C100, t8
	
	BF t8, L9
		LDR R7, C100 ; C100 → R6
		MOV R5, FP
		ADI R5, #-16 ;address of x
		LDR R7, (R5) ; x → R7
		CMP R7, R6
		BGT R7, L7 ; x>C100 GOTO L7
		MOV R6, R0 ; Set FALSE
		MOV R5, FP
		ADI R5, #-40 ;address of t8
		STR R6, (R5)
		JMP L8
		L7: MOV R6, R1 ; Set TRUE
		MOV R5, FP
		ADI R5, #-40 ;address of t8
		STR R6, (R5)
		
		L8: MOV R5, FP
		ADI R5, #-40 ;address of t8
		LDR R6, (R5)
		BRZ R6, L9

return -1;	RETURN CN1
	
	JMP L10
		LDR R4, CN1 ; -1 → R4
		MOV SP, FP ;
		MOV R6, FP
		CMP R6, SB ;
		BGT R6, UNDERLOW
		LDR R6, (FP) ; rtn addr
		MOV R5, FP
		ADI R5, #-4
		LDR FP, (R5) ; PFP → FP
		STR R4, (SP) ; return -1
		JMR R6 ; GOTO rtn addr
		
		JMP L10

else
return x;
	L9: RETURN x	MOV R5, FP
		ADI R5, #-16 ; address of x
		LDR R4, (R5) ; x → R4
		MOV SP, FP ;
		MOV R6, FP
		CMP R6, SB ;
		BGT R6, UNDERLOW
		LDR R6, (FP) ; rtn addr
		MOV R5, FP
		ADI R5, #-4
		LDR FP, (R5) ; PFP → FP
		STR R4, (SP) ; return -1
		JMR R6 ; GOTO rtn addr

}	L10: RTN	MOV SP, FP
		MOV R6, FP
		CMP R6, SB
		BGT R6, UNDERLOW
		LDR R6, (FP) ; rtn addr
		MOV R5, FP
		ADI R5, #-4
		LDR FP, (R5) ; PFP → FP
		JMR R6 ; GOTO rtn addr