Blitz-64: Guide to Adding New Instructions

Harry H. Porter III

HHPorter3@gmail.com

18 October 2022

This document lists the steps involved in adding a new instruction to the Blitz-64 Instruction Set Architecture (ISA). It also discusses the steps to adding a new built-in function to KPL language and compiler in order to take advantage of the new instruction.

Table of Contents

Chapter 1: Checklist

Here is a checklist of the steps you need to take to add a new instruction to the Blitz Instruction Set Architecture (ISA). These are described in detail in the following chapters.

- ☐ Design and specify the instruction; select a format, select an op-code
- ☐ Update the ISA document
  - ☐ Add an entry to “All Instructions - Summary Listing”
  - ☐ Add an entry to “Machine Instructions, Grouped By Format”
  - ☐ Add a description of the instruction
  - ☐ Modify the text giving the count of existing instructions.
  - ☐ Add an entry to “Instruction List”, showing the op-code.
  - ☐ Add an entry to “Appendix: Recent Changes”.
  - ☐ Update the document’s date and revision history.
- ☐ Update the assembler (C Version)
- ☐ Update the assembler (KPL Version)
- ☐ Update the assembler verification suite
  - ☐ Verify the C Version and the KPL Version.
- ☐ Update the virtual machine emulator
  - ☐ Verify the execution of the instruction
- ☐ Modify KPL (optional)
  - ☐ Add an external function that uses the instruction
  - ☐ Add an built-in function that uses the instruction
  - ☐ Add a test file to the KPL execution test suite
- ☐ Modify ISAValidator
- ☐ Update the any hardware implementations
  - ☐ Verify the execution of the instruction
- ☐ Publish the changes
  - ☐ Update the modified documents on the website.
  - ☐ Update the website as needed to indicate updates.
  - ☐ Update the modified code on the website.

Chapter 2: Add a New Machine Instruction

Design and Specify the New Instruction

The process is described using an example that will run through this and following chapters.

We will add a new machine instruction called “mulu”, for “multiply unsigned”. Here is the description, to describe the instruction:

`MULU`	*RegD,Reg1,Reg2*	RegD ← Reg1 × Reg2 (unsigned)
This instruction multiplies the contents of Reg1 and Reg2 and places the result in RegD. The arguments and the result are treated as 64 bit unsigned integers. Overflow is ignored and no exceptions will be raised.		

This instruction is similar in format to “MUL”, so it will naturally be Format A-3. With format A instructions, we must select a new value for the OP2 opcode. The next available OP2 op-code is 70 (i.e., 0x46).

This instruction will not raise any exceptions.

Update the ISA document

First, we will update the listing of all instructions, grouped by functionality. We’ll add a new entry just after the existing MUL instruction:

All Instructions - Summary Listing

Arithmetic

```
ADD         RegD,Reg1,Reg2
…
MUL         RegD,Reg1,Reg2
MULU        RegD,Reg1,Reg2
…
```

Next, we add an entry to the list of instructions, grouped by format:

Machine Instructions, Grouped By Format

…

Format A-3 *RegD,Reg1,Reg2*

```
…
MUL        r7,r1,r2
MULU       r7,r1,r2
…
```

Next, we add the description of the instruction to the main body of the "Instructions" chapter.

Next, we modify the count of instructions. See highlight, where we changed 70→71; 111→112 and 69→70.

Currently there are…

Number of Format-A Instructions	**71**
Number of non-Format-A Instructions	**41**
Total Number of Machine Instructions	**112**
Current range of OP2 values	**0 … 70**
Current range of OP1 values	**1 … 41**

Next, we add an entry in the list of opcodes:

Arithmetic

OP1		OP2				
hex	**dec**	**hex**	**dec**	**format**		
		01	1	A	ADD	
01	1			B	ADDI	
		02	2	A	ADDOK	
		45	69	A	ADD3	*(Note that OP2 is out of order)*
		03	3	A	SUB	
		04	4	A	MUL	
		46	70	A	MULU	*(Note that OP2 is out of order)*
…						

Next, we add an entry to the end of the “Recent Changes” list:

24 May 2021

The MULU instruction was added.

Next, you must update the title page, to change the document’s date and add your name as an author. ***This is required!***

Blitz-64

Instruction Set Architecture Reference Manual

<Your name>, <Other info>
Harry H. Porter III, Portland State University

YourEmail@…
HHPorter3@gmail.com

24 May 2020

Finally, you must update the document's revision history. ***This is required!***

Document Revision History / Permission to Copy

Version numbers are not used to identify revisions to this document. Instead the date and the author's name are used. The document history is:

Date	Author
23 May 2018	Harry H. Porter III <initial version>
28 May 2019	Harry H. Porter III <document mostly completed>
24 May 2021	**YOUR NAME** <MULU added>
24 May 2021	**YOUR NAME** <current version>

Update the ISA Quick Reference

Modify the following document:

"Blitz-64 - ISA Quick Reference"

Add a line for the new instruction:

Synthetic	OP1	OP2	Exceptions	Operands	Description
…					
Multiply/Divide — Optional: may cause Emulated Instruction Exception					
MUL		04	Arithmetic	RegD,Reg1,Reg2	RegD ← Reg1 × Reg2
MULU		46		RegD,Reg1,Reg2	RegD ← Reg1 × Reg2 (unsigned)
DIV		05	Arithmetic	RegD,Reg1,Reg2	RegD ← Reg1 ÷ Reg2
REM		06	Arithmetic	RegD,Reg1,Reg2	RegD ← Reg1 % Reg2

Also, you must update the **author** and **date** on the document.

Update the Assembler (C Version)

We need to edit the file **asm.c**.

First, update the header comment. Add a line to the revision history and update the VERSION date.

```
...
// Original Author:
//   19 July 2018 - Harry H. Porter III
//
// Revision History:
//   …
//   16 May 2021       - Harry - Disabling of -g and -hex command
line options
//   24 May 2021       - YOUR NAME - Addition of MULU instruction
//
// Date of last modification:
//
#define VERSION "=====        24 May 2021       ====="
//              "=====    <spacing template>    =====“
...
```

Next, add an entry to the **enum** list of “symbols”:

```
enum {

// Machine instructions...

       ...
       MUL,
       MULU,
       DIV,
       ...
```

Next, add an entry to **StringForCode**, which maps from symbol to spelling:

```
char * StringForCode (int i) {
  switch (i) {

    ...
    case MUL:           return "mul";
    case MULU:          return "mulu";
    case DIV:           return "div";
    ...
```

Next, add an entry to set the opcode for this instruction. In our example, OP1 is 0x00 and OP2 is 0x46. For a non-Format A instruction (where OP2 is not used), we would use a value such as 0x99000000.

```
void InitializeOpcodeMapping () {

  ...
  opcodeMapping [ MUL ]        = 0x00040000;
  opcodeMapping [ MULU ]       = 0x00460000;
  opcodeMapping [ DIV ]        = 0x00050000;
  ...
```

Next, add an entry to **InitKeywords**, which will map from the opcode spelling to the symbol:

```
   void InitKeywords () {

     ...
     LookupAndAdd ("mul",        MUL);
     LookupAndAdd ("mulu",       MULU);
     LookupAndAdd ("div",        DIV);
     ...
```

Next, add an entry to **CategoryOf**, which maps from the symbol to the format of the instruction. Our instruction is a Format A-3, so we add our entry there:

```
   int CategoryOf (int tokType) {
     switch (tokType) {
   ...
     // Format A-3     RegD,Reg1,Reg2
       case ADD:
       case SUB:
       case MUL:
       case MULU:
       case DIV:
   ...
         return FORMAT_A3;
   ...
```

Next, compile the assembler, with **make**.

Update the Assembler (KPL Version)

Next, we must update the version of the assembler written in KPL. The updates are the same.

Update the Assembler Verification Suite

Begin by running this script to verify that there are no other errors.

```
     Shell% runAll | more
```

The following program contains every instruction:

```
AsmTestFiles/pgm2.s
```

Add a line to this file for the new instruction:

```
...
# Format A-3: RegD,Reg1,Reg2
   add       r7,r1,r2
   addok          r7,r1,r2
   sub       r7,r1,r2
   mul       r7,r1,r2
   mulu      r7,r1,r2
   div       r7,r1,r2
   ...
```

Next, run the script:

```
Shell% runAll | more
```

Scan for this line in the "listing" output and verify that the instruction is assembled correctly. Also, make sure there are no errors.

```
...
> 000000080 00460217    | 131:          mulu            r7,r1,r2
> 000000084 00050217    | 132:          div             r7,r1,r2
...
```

Next, update the files with:

```
Shell% updateAll
```

Then, a final run should show no problems:

```
Shell% runAll | more
```

Verify the KPL Version (asm2)

Next, verify the KPL version has no problems and produces the same output as the C version:

```
Shell% cd KPL-Code/asm2
Shell% ./runTest pgm2
=========== Running asm2 test: pgm2
Shell%
```

You can also run all the tests and check for discrepancies

```
Shell% runAll
...
```

There is a discrepancy in **pgm5**, which can be ignored[1]:

```
=========== Running asm2 test: pgm5
376c376
<  524  FLOAT  FLOAT  543  8  0  00000000  r0,r0,r0,r0  -1.23456e+80
---
>  524  FLOAT  FLOAT  543  8  0  00000000  r0,r0,r0,r0  -1.2345600000000003e+80
1035c1035
< 000ee0547 4f16c2a7    |
---
> 000ee0547 4f16c2a8    |
===================== pgm5: Known difference 4f16c2a7 vs. 4f16c2a8; KPL
assembler .double is not perfect...
```

Update the Emulator

We need to edit the file **blitz.c**.

First, update the header comment. Add a line to the revision history and update the VERSION date.

[1] The problem is that the assembler (KPL version) is using an outdated lexical analyzer. It does not perform the conversion from decimal to floating point correctly. The lexer should be replaced with something like the version in the KPL compiler, which can handle separators ("_") and will convert from decimal to floating point correctly. Note that the KPL compiler will NEVER generate the "**.float**" pseudo-op, so this WILL NOT AFFECT KPL PROGRAMS.

```
...
// Original Author:
//   9 July 2018 - Harry H. Porter III
//
// Revision History:
//   …
//   6 May 2021        - Harry - Current version
//   24 May 2021       - YOUR NAME - Addition of MULU instruction
//
// Date of last modification:
//
#define VERSION "=====          24 May 2021           ====="
//              "=====       ...xx...xx...xx...       ====="
//              "=====       <spacing template>       ====="
...
```

Next, add a new **#define** constant with the new op-code. Our example instruction is has an OP2, but there is a similar section for OP1 instructions.

```
...
#define OP2_CHECKA      68
#define OP2_ADD3        69
#define OP2_MULU        70
...
```

Next, update the data used for disassembling, which is located in the function **initDisassemblerData()**.

First, add an element to either **op1codes** to **op2codes**. Since our instruction is format A, we add an element to **op2codes**:

```
  op2codes [OP2_MULU]         = "mulu";
```

Determine which format the instruction should be displayed in. We see that our new instruction will use FORMAT_A3.

```
    // For disassembly, these codes tell how to print the operands:
    //    Format A-0    <no operands>
    //    Format A-1    Reg1
    //    Format A-2    RegD,Reg1
    //    Format A-3    RegD,Reg1,Reg2
    //    Format A-4    RegD,Reg1,Reg2,Reg3
    //    Format A-5    Reg1,Reg2
    //    Format A-6    Reg2
    //    Format A-7    RegD,CSRReg1,Reg2
    //    Format A-8    RegD,CSRReg1
    //    Format A-9    RegD
    //    Format B-1    RegD,Reg1,immed-16
    //    Format B-2    RegD,immed-16(Reg1)
    //    Format B-3    RegD,Reg1,immed-3
    //    Format B-4    immed-10
    //    Format B-5    RegD,Reg1,immed-6
    //    Format B-6    CSRReg1,immed-16
    //    Format C-1    immed-16(Reg1),Reg2
    //    Format C-2    Reg1,Reg2,immed-16
    //    Format D-1    RegD,immed-20
```

Add an element to either **op1category** to **op2category**. Since our instruction is format A, we add a line to **op2Category**:

```
        op2category [OP2_ADD]          = FORMAT_A3;
```

Next, we must add some code to execute the instruction. Each instruction has a “case” within this code:

```
    op1 = ...
    switch (op1) {
      case 0:
        op2 = ...
        switch (op2) {
          // HERE ARE THE FORMAT-A INSTRUCTIONS
          case OP2_ADD:
            ...
          case OP2_ADDOK:
            ...
        }
        ...
      // HERE ARE THE INSTRUCTIONS WHICH ARE NOT FORMAT-A
      case OP1_ADDI:
        ...
      case OP1_ANDI:
        ...
    }
```

Here is the code we will add for our new instruction:

```
      // MULU    RegD,Reg1,Reg2
      case OP2_MULU:
        x = getReg1 (instr);
        y = getReg2 (instr);
        z = ((uint64_t) x) * ((uint64_t) y);
        putRegD (instr, z);
        return;
```

If some synthetic instruction will be implemented using our new instruction, we may wish to modify the **disassemble** function to special case on our instruction to print it in the synthetic form.

Verify the Execution of the Instruction

We must verify that the new instruction is executed properly and that our changes to the emulator work correctly. How we accomplish this will depend on the nature of the new instruction.

For MULU, we will write a simple assembly program called **testmulu.s**.

```
     .begin    kernel,startaddr=0x8,gp=undefined
_entry:
     .export   _entry
     mulu      r5,r3,r4
     debug
     jump      _entry
```

We can assemble, link, and execute with:

```
Shell% asm testmulu.s
Shell% link testmulu.o -k -o testmulu.exe
Shell% blitz testmulu.exe
```

Let's start by verifying that the instruction is loaded and disassembles correctly, using the "**dis**" command:

```
Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about
commands.
E> dis
Enter the beginning address (in hex): << ENTER >>
     000000000: 00000000   <inaccessible>
     000000004: 00000000   <inaccessible>
   _entry:
     000000008: 00460435       mulu    r5,r3,r4
     00000000C: 00280000       debug
     000000010: 19FFFF80       jump    _entry     # PC - 0x8 (PC + 0xFFFFFFF8)
     000000014: 00000000
     ...
E>
```

Next, we will load some test values into registers using the "**r3**" and "**r4**" commands.

```
E> r3
  r3 = 0x0000000010000000     ( decimal: 268435456 )
Enter the new value (in hex): 12345
  r3 = 0x0000000000012345     ( decimal: 74565 )
E> r4
  r4 = 0x0000000000010000     ( decimal: 65536 )
Enter the new value (in hex): 22222
  r4 = 0x0000000000022222     ( decimal: 139810 )
E>
```

Next, we use the "**go**" command to begin execution. After executing the MULU instruction, our little program immediately executes a DEBUG instruction, so we can check the results.

```
E> g
Beginning execution...
****  A DEBUG machine instruction was executed  *****
...
Done!
E>
```

We will use the "**regs**" command to examine the contents of the registers.

```
E> r
...
========================  REGISTERS  ========================
  ...
  r3         = 0x0000000000012345     ( decimal: 74565 )
  r4         = 0x0000000000022222     ( decimal: 139810 )
  r5         = 0x000000026d5fd92a     ( decimal: 10424932650 )
  r6         = 0x0000000000000008     ( decimal: 8 )
  ...
E>
```

We see that register r5 contains the correct result. Looking at the values in decimal, note that:

74,565 × 139,810 = 10,424,932,650

Next, using this same procedure, we'll try more values to test the extremal cases. For example, the following computations would result in an overflow if performed with signed addition:

```
0x7fff_ffff_ffff_ffff × 2              = 0xffff_ffff_ffff_fffe
0x00ff_ffff_ffff_ffff × 0x40_0000      = 0xffff_ffff_ffc0_0000
0x1_0000_0000 × 0x0123_4567_89ab_cdef = 0x89ab_cdef_0000_0000
```

It is always necessary to test extremal, "edge cases" to make sure the instruction works 100% as it is supposed to.

Later, when we add KPL code to exercise this instruction, we want to perform a thorough testing, including all edge cases, since the KPL code will go into the validation suite, to be checked whenever future changes are made.

Chapter 3: Modify KPL

Modify KPL

It may be desirable to somehow allow the KPL programmer to take advantage of this new instruction. There are two possible approaches:

- Add an external function that uses the new instruction
- Add a built-in function to the language

In either case, you'll need to add a function to the KPL test suite to verify that the new function works properly.

Adding an External Function

In our example, we first create a hand-coded assembly function that uses the MULU instruction. We begin by adding this to **System.h**, which makes this function accessible to all KPL programs.

```
external UnsignedMult (x, y: int) returns int
```

Next, we modify runtime.s to add this line, to export the function's name:

```
.export    UnsignedMult
```

Next, we add a function, such as:

```
#
# =====================  UnsignedMult  =====================
#
# external UnsignedMult (x, y: int) returns int
#
# This function return x*y. All values are treated as 64 bit unsigned integers.
# Overflow is not detected and no error can arise.
#
# Stack usage:     not used
# Arguments:       r1 = x; r2 = y
# Returned value:  r1
# Interrupts:      Don't care
# Max Stack Usage: 0 bytes (leaf function)
#
UnsignedMult:
   .function      "UnsignedMult",line=0,framesize=0
   mulu            r1,r1,r2
   ret
   .endfunction
```

Recall that KPL passes arguments in **r1**, **r2**, **r3**, ..., so this function will expect **x** and **y** to be in **r1** and **r2** on entry. Functions return their results in **r1**.

This function has no need for a stack frame since it calls no other functions. Therefore, this is a leaf function. This is indicated by the debugging information **.function** statement, with "**framesize=0**".

The overhead involved in using the MULU instruction in KPL is therefore 2 additional instructions: the CALL and the RET.[2]

Adding a Built-In KPL Function (C Version)

Some instructions may be performance-critical and we may not want the overhead that a call to an externally defined function requires. Also, by creating a built-in function, the optimizer may be able to improve the code by using registers more effectively.

However, creating and adding a built-on function is a more complicated approach.

[2] There may be additional overhead, if the values must be moved from some other registers into r1 and r2, since the compiler is restricted to using the registers in only this one way. In other words, placing the MULU in separate function ties the hands of the optimizer.

First, modify **main.h** to add a new string:

```
   extern String * stringUnsignedMult;
```

Next, add a new symbol to **main.h** to represent this operation:

```
   // Symbols used to identify build-in function ids, and message selectors

   UPCAST_TO_HALFWORD, UPCAST_TO_WORD, UPCAST_TO_INT,
   UPCAST_TO_DOUBLE, FORCE_TO_BYTE, FORCE_TO_HALFWORD,
   FORCE_TO_WORD, FORCE_TO_DOUBLE, FORCE_TO_INT, ADD_OK, UNSIGNED_MULT,
```

Next, add these line to **main.cc** for this string.

```
   String * stringUnsignedMult;
   stringUnsignedMult = lookupAndAdd ("unsignedMult", ID);
```

Next, add the following to **main.cc** to associate this symbol with this string:

```
   stringUnsignedMult -> primitiveSymbol = UNSIGNED_MULT;
```

Also modify **main.cc** to add your name and date of modification:

```
   #define YOUR_NAME            "Harry Porter"
   #define DATE_OF_LAST_MOD     "24 May 2021"       // Today's date in format "DD Month YYYY"
   ...
   // Revision History:
   //   ...
   //   August 2020    - Harry Porter - Additional testing and cleanup
   //   24 May 2021    - Harry Porter - Add "unsignedMult" built-in function
```

Next, modify **lexer.cc** to print this symbol out:

```
       case MUL_U: return "MUL_U";
```

Next, modify **check.cc** by adding a case to the function **evalExprsIn**:

```
case UNSIGNED_MULT:                    // in evalExprsIn - CALL_EXPR
  if ((argCount (callExpr->argList) == 2) &&
      (isAnyIntegerConst (callExpr->argList->expr)) &&
      (isAnyIntegerConst (callExpr->argList->next->expr))) {
    arg1 = getAnyIntegerConstValue (callExpr->argList->expr);
    arg2 = getAnyIntegerConstValue (callExpr->argList->next->expr);
    arg2 = arg1 * arg2;      // C++ will ignore overflow here
    return convertValueToSmallestIntegerConst (arg2, callExpr);
  }
  return node;
```

This code will try to compute the function at compile time, if it meets these conditions:

- The are exactly 2 arguments
- Both arguments are integer constants, known at compile-time

If so, this code performs the operation and returns the result. Otherwise, it ignores the call.

Next, modify the **checkTypes** function in **check.cc** to add the following code:

```
case UNSIGNED_MULT:                        // in checkTypes for CALL_EXPR...
   if (argCount (callExpr->argList) == 2) {
     if ((callExpr->destSize != -1) || (callExpr->srcSize != -1)) {
       return basicIntType;
     }
     callExpr->knownOpSymbol = UNSIGNED_MULT;
     callExpr->destSize = 8;

     callExpr->argList->expr = checkAssignment (
         callExpr->argList->expr,
         basicIntType,
         "Built-in function 'unsignedMult' expects both arguments to be type "
                                       "INT but the first is not",
         NULL);
     callExpr->argList->next->expr = checkAssignment (
         callExpr->argList->next->expr,
         basicIntType,
         "Built-in function 'unsignedMult' expects both arguments to be type "
                                      "INT but the second is not",
         NULL);
   } else {
     error (callExpr, "Built-in function 'unsignedMult' expects "
                                                 "exactly two arguments");
   }
   return basicIntType;
```

This code first checks to see if it has already visited this node and done the work. If this node has not yet been dealt with, it sets a couple of variables in the node that will be used during code generation. Then it checks that there are exactly two arguments, printing an error if not. It then checks that, whatever expressions are present, they can be assigned to a variable of type **int**, printing an error if not. The result of invoking **mulu** will be of type **int**, and this is indicated by returning **basicIntType**.

Next, modify **ir.h** and **ir.cc** to add a new IR instruction to represent the MULU machine instruction.

First modify **ir.h** to give this instruction a code. (This has nothing to do with the OP1 or OP2 opcodes; it is an opcode internal to the compiler.) We find an unused number (108) and add this line:

```
#define OP_Mulu     108
```

Next, we add this code to **ir.h** to define this IR instruction:

```
//----------  Mulu  ----------

void IR_Mulu (const char * destReg, const char * regA, const char * regB);

class IRMulu : public IR {
  public:
    const char * destReg;
    const char * regA;
    const char * regB;

    IRMulu (const char * d, const char * a, const char * b) : IR (OP_Mulu) {
      destReg = d;
      regA = a;
      regB = b;
    }
    ~ IRMulu () {}
    virtual void Print();
    virtual void PrintCode ();
    virtual bool WritesReg (const char * r);
    virtual bool ReadsReg (const char * r);
};
```

Next, we modify **ir.cc** to add the code for this IR:

```
//----------  Mulu  ----------

void IR_Mulu (const char * destReg, const char * regA, const char * regB) {
  linkIR (new IRMulu (destReg, regA, regB));
}

void IRMulu::Print () {
  printf ("    MULU        %s, %s, %s\n", destReg, regA, regB);
}

bool IRMulu::WritesReg (const char * r) {
  return r == destReg;
}

bool IRMulu::ReadsReg (const char * r) {
  return (r == regA) || (r == regB);
}

void IRMulu::PrintCode () {
  // fprintf (outputFile, "==========  TEST-Mulu\n");
  fprintf (outputFile, "\tmulu\t\t%s,%s,%s\n", destReg, regA, regB);
}
```

This **Print** method might be used in debugging the compiler or optimizer to see this IR instruction. The **PrintCode** method will be used to create the actual assembly code.

The **ReadsReg** function is passed a register and must respond with true if this IR may read that register. The **WritesReg** function is passed a register and must return true if the instruction might modify that register. These functions will be used by the optimizer.

Next, in **gen.cc**, we add the code that will go from a use of the built-in “**mulu**” function to the IR_MULU instruction. We add this code:

```
case MUL_U:                          // in genCallExpr

  arg1 = callExpr->argList->expr;
  arg2 = callExpr->argList->next->expr;
  x = genExpr (arg1, 8);
  y = genExpr (arg2, 8);
  if (trueLabel) {
    programLogicError ("In genCallExpr, MUL_U");
  }
  GenIR_GetIntoReg (Reg6, x, 8);
  GenIR_GetIntoReg (Reg7, y, 8);
  IR_Mulu (Reg7, Reg6, Reg7);
  IR_Stored (target, Reg7, false);
  return;
```

Adding a Built-In KPL Function (KPL Version)

If we are defining a new built-in functions, we must do the same steps to the KPL version of the compiler.

Add this to **KPLBasic.h**:

```
stringUnsignedMult: ptr to KPLString
```

and this:

```
-- Symbols used to identify build-in function ids, and message selectors

UPCAST_TO_HALFWORD, UPCAST_TO_WORD, UPCAST_TO_INT,
UPCAST_TO_DOUBLE, FORCE_TO_BYTE, FORCE_TO_HALFWORD,
FORCE_TO_WORD, FORCE_TO_DOUBLE, FORCE_TO_INT, ADD_OK, UNSIGNED_MULT,
```

Add this to **KPLMain.c**:

```
stringUnsignedMult.primitiveSymbol = UNSIGNED_MULT
```

and this:

```
stringUnsignedMult = lookupAndAdd ("unsignedMult", ID)
```

Also modify **KPLMain.c** to add your name and date of modification:

```
const YOUR_NAME =             "Harry Porter"
const DATE_OF_LAST_MOD =      "24 May 2021"       -- Today's date in format "DD Month YYYY"
...
-- Revision History:
--   ...
--   21 September 2020  - Harry Porter - Conversion into KPL begins
--   24 May 2021        - Harry Porter - Add "unsignedMult" built-in function
```

Add this to **KPLLexer.c**:

```
case UNSIGNED_MULT: return “UNSIGNED_MULT"
```

In **KPLCheck.c,** add a case to the function **evalExprsIn**:

```
    case UNSIGNED_MULT:           -- in evalExprsIn - CALL_EXPR
      if ((argCount (callExpr.argList) == 2) &&
          (isAnyIntegerConst (callExpr.argList.expr)) &&
          (isAnyIntegerConst (callExpr.argList.next.expr)))
        arg1 = getAnyIntegerConstValue (callExpr.argList.expr)
        arg2 = getAnyIntegerConstValue (callExpr.argList.next.expr)
        arg2 = unsignedMult (arg1, arg2)
        return convertValueToSmallestIntegerConst (arg2, callExpr)
      endIf
      return node
```

Note that we are actually using the “mulu” function. Gotta love bootstrapping. This works because this code will be compiled with the C version of the KPL compiler. But if we no longer have that version, i.e., we do not have a version of the compiler that already handles “mulu”, we would have to leave this case out for now. After we get the compiler functioning with “mulu” without static expression evaluation, we can go back and add this code in and recompile.

To **KPLCheck.c** add this case to function **checkTypes**:

```
 case UNSIGNED_MULT:              -- in checkTypes for CALL_EXPR...
    if (argCount (callExpr.argList) == 2)
      if ((callExpr.destSize != -1) || (callExpr.srcSize != -1))
        return basicIntType
      endIf
      callExpr.knownOpSymbol = UNSIGNED_MULT
      callExpr.destSize = 8

      callExpr.argList.expr = checkAssignment (
          callExpr.argList.expr,
          basicIntType,
          "Built-in function 'unsignedMult' expects both arguments to be"
                                             "type INT but the first is not",
          null)
      callExpr.argList.next.expr = checkAssignment (
          callExpr.argList.next.expr,
          basicIntType,
          "Built-in function 'unsignedMult' expects both arguments to be"
                                             "type INT but the second is not",
          null)
    else
      error (callExpr, "Built-in function 'unsignedMult' expects exactly "
                                             "two arguments")
    endIf
    return basicIntType
```

To **KPLAst.h** add this code:

```
------------  Mulu  ----------

functions IR_Mulu (destReg_: String, regA_: String, regB_: String)

class IRMulu
  superclass IR
  fields
    destReg: String
    regA: String
    regB: String
  methods
    WritesReg (r: String) returns bool
    ReadsReg (r: String) returns bool
    Print()
    PrintCode ()
endClass
```

To **KPLAst.c** add this:

```
------------  Mulu  ----------

function IR_Mulu (destReg_: String, regA_: String, regB_: String)
  linkIR (alloc IRMul { destReg = destReg_,
                        regA = regA_,
                        regB = regB_,
                        next = null,
                        prev = null })
endFunction

behavior IRMulu

  method Print ()
    printf ("\tMULU\t\t\t%s, %s, %s\n", destReg, regA, regB)
  endMethod

  method PrintCode ()
    -- printf (outputFile, "==========  TEST-Mulu\n")
    printf (outputFile, "\tmulu\t\t%s,%s,%s\n", destReg, regA, regB)
  endMethod

  method ReadsReg (r: String) returns bool
    return (r == regA) || (r == regB)
  endMethod

  method WritesReg (r: String) returns bool
    return r == destReg
  endMethod

endBehavior
```

In **KPLGen.c**, add this:

```
case UNSIGNED_MULT:                    -- in genCallExpr

  arg1 = callExpr.argList.expr
  arg2 = callExpr.argList.next.expr
  x = genExpr (arg1, 8)
  y = genExpr (arg2, 8)
  if (trueLabel)
    programLogicError ("In genCallExpr, UNSIGNED_MULT”)
  endIf
  GenIR_GetIntoReg (Reg6, x, 8)
  GenIR_GetIntoReg (Reg7, y, 8)
  IR_Mulu (Reg7, Reg6, Reg7)
  IR_Stored (target, Reg7, false)
  return
```

Add a Test Program to the Execution Validation Suite

In order to test any changes to the emulator, a test program must be created, or an existing test program must be modified.

For our example, we will create a new program called **TestEmulator**. Add files:

```
Testing/ExecutionTests/TestEmulator.h
Testing/ExecutionTests/TestEmulator.c
```

Here is **TestEmulator.h**:

```
header TestEmulator
  uses System, PrintPackage
  functions
    main ()
endHeader
```

Here is **TestEmulator.c**:

```
code TestEmulator

  function main ()
    testUnsignedMult (3, 4, 12)
    testUnsignedMult (0x7fffffffffffffff, 0x100000, 0xfffffffffff00000)
    testUnsignedMult (0x8000000000000000, 0x1, 0x8000000000000000)
    testUnsignedMult (0x1234567890abcdef, 0x1234567890abcdef, 0xa6475f09a2f2a521)
    …
    EmulatorShutdown (0)
  endFunction


  function testUnsignedMult (x, y, expectedAns: int)
    var ans: int
    printf ("%#016.16x × %#016.16x = ", x, y)
    ans = unsignedMult (x, y)
    if ans == expectedAns
      printf ("%#016.16x\n", ans)
    else
      printf ("%#016.16x   ***** ERROR: EXPECTED %#016.16x *****\n", ans, expectedAns)
      EmulatorShutdown (1)
    endIf
  endFunction

endCode
```

This program will:

Print a line for each test.
Compare the result to an expected result.
If there is a mismatch…
    It prints an error message.
    It invokes **EmulatorShutdown (1).**
If all tests pass okay…
    It invokes **EmulatorShutdown (0).**

This test program follows a general pattern which makes it convenient to test our new function on a large a number of test cases.

First, we must make sure the program compiles correctly.

To this file:

```
Testing/runCompAll
```

add this line:

```
   ./runOneComp 0 ExecutableTests/TestEmulator -unsafe -s -testNoHash $1
```

To file

```
   Testing/updateCompAll
```

add

```
   ./updateOneComp ExecutableTests/TestEmulator -unsafe -s -testNoHash $1 $2
```

To file

```
   Testing/runExecAll
```

add

```
   ./runOneExec 0 TestEmulator -unsafe -testNoHash $1
```

To file

```
   Testing/updateExecAll
```

add

```
   ./updateOneExec TestEmulator -unsafe -testNoHash
```

The **-unsafe** option is only needed if the test program will perform unsafe operations. Normally, the KPL compiler adds the date and time to its output files. This causes many minor, irrelevant differences with the expected output. The **-testNoHash** option is used to suppress the data and time, suppressing a lot of useless messages. The **-s** option causes the compiler to print a bunch of symbol table information, which serves as an extra check against unexpected behavior of the compiler.

From here on, we assume we are in directory **Testing**:

```
   Shell% cd Testing
```

After the **TestEmulator.h** and **TestEmulator.c** files are created, run this

```
Shell% runOneComp 1 ExecutableTests/TestEmulator -unsafe -s -testNoHash
```

Make sure there are no errors and that you see:

```
KPL        OK
Assembly   OK
Link       OK
```

Then run this:

```
Shell% updateOneComp ExecutableTests/TestEmulator -unsafe -s -testNoHash
```

to create the “BAK” files, which are used for comparison. After that, you can run the following command to see any differences in the **.s** file produced by the compiler.

```
Shell% runOneComp 0 ExecutableTests/TestEmulator -unsafe -s -testNoHash
======================== RUNNING TEST: ExecutableTests/TestEmulator
Shell%
```

To execute the program, run this:

```
Shell% runOneExec 1 TestEmulator -unsafe -testNoHash
```

Make sure there are no errors and that you see:

```
KPL        OK
Assembly   OK
Link       OK
```

This will suggest a line you execute to run the program. In this case, we see:

```
CONSIDER EXECUTING:
  ...
  blitz ExecutableTests/ExpectedResults/TestEmulator.exe \
                -g -fp -nowarn -args "..."
```

So we can type this to run our test program:

```
Shell% blitz ExecutableTests/ExpectedResults/TestEmulator.exe -g -fp -nowarn
```

To create the BAK files, run this:

```
Shell% updateOneExec TestEmulator -unsafe -testNoHash
```

After that, you can run the following line to see what changes in the output:

```
Shell% runOneExec 0 TestEmulator -unsafe -testNoHash
====================== RUNNING TEST: TestEmulator
Shell%
```

Since we have added a new string (**stringUnsignedMult**), the parser tests must be updated since all the numbers have shifted. Run this and look for any problems:

```
Shell% runParserAll
```

If there are no errors reported, run this:

```
Shell% updateParserAll
```

When you are all done, you should be able to run the following with no surprises:

```
Shell% runCompAll
Shell% runExecAll
```

Modify KPL Documentation

Alter the document(s) to describe any new built-in functions.

Chapter 4: Modify Hardware

Modify ISAValidator.s

This is an assembly language program which is used to check and verify that a Blitz-64 core implementation executes all instructions correctly.

Add a new test to execute the MULU instruction in ways that test all boundary cases.

Modify the Hardware Implementations

It impossible to describe this step.

Chapter 5: Publish Changes

Push all changes to software and documentation to the Internet.

About This Document

Document Revision History / Permission to Copy

Version numbers are not used to identify revisions to this document. Instead the date and the author's name is used. The document history is:

Date	**Author**
24 May 2021	Harry H. Porter III <document created>
17 June 2021	Harry H. Porter III
18 October 2022	Harry H. Porter III <current version>

Corrections and Errors

Please contact the author if you find…

- Inaccurate information that you can correct
- Incomplete information that you can fill in
- Confusing text that needs to be reworded

Thanks!

About the Author

Professor Harry H. Porter III teaches in the Department of Computer Science at Portland State University. He has produced several video courses, notably on the Theory of Computation. Recently he built a complete computer using the relay technology of the 1940s. The computer has eight general purpose 8 bit registers, a 16 bit program counter, and a complete instruction set, all housed in mahogany cabinets as shown. Porter also designed and constructed the Blitz System, a collection of software designed to support a university-level course on Operating Systems. Using the software, students implement a small, but complete, time-sliced, VM-based operating system kernel. Porter has habit of designing and implementing programming languages, the most recent being a language specifically targeted at kernel implementation.

Porter holds an Sc.B. from Brown University and a Ph.D. from the Oregon Graduate Center.

Porter lives in Portland, Oregon. When not trying to figure out how his computer works, he skis, hikes, travels, and spends time with his children building things.

Professor Porter's website: www.cecs.pdx.edu/~harry