# An Assembler for the MSSP Distiller

*Eric Zimmerman*
*University of Illinois, Urbana-Champaign*

## Abstract
It is important to have a means of manually testing a potential optimization before laboring to fully implement it in the MSSP distiller. To achieve this, we have written a tool to generate disassembled output from a distilled program. This allows the compiler architect to test potential enhancements by making changes at the assembly code level. The tool then reassembles the modified code into the distiller's intermediate representation. Finally, the rebuilt instruction stream is run on the MSSP simulator where the optimized performance impact can be gauged.

## Introduction
The MSSP distiller is an optimizing compiler that generates approximate code through profile-guided transformations. Approximate code is run on a high-performance master processor in an MSSP configuration [Zilles 2002]. Because the distiller operates on a program binary without access to its source code, a program's behavior must be determined empirically from the execution profile before good optimizations can be made. While some optimizations like efficient code layout have obvious performance benefits, the potential benefit of other enhancements is not as clear.

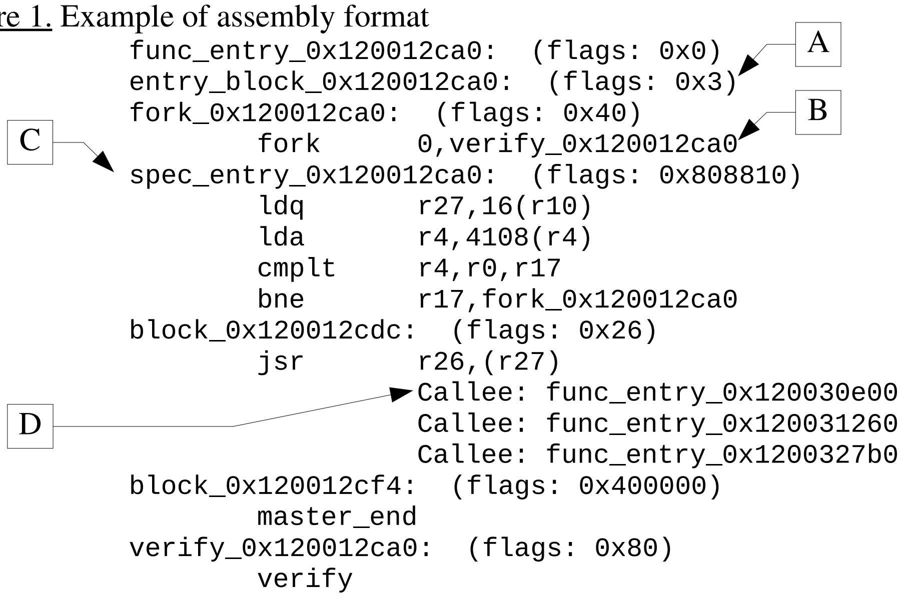
After the distilled code is generated, it is difficult to modify. Instruction opcodes, register references, and branch offsets are fixed and encoded in the 32-bit instruction words used by the MSSP simulation infrastructure. We have written a tool to simplify the process of modifying distilled code by allowing changes to be made at the assembly level. This will allow quicker prototyping and evaluation of optimization strategies in the distiller.

Our MSSP simulation infrastructure implements the HP Alpha Instruction Set Architecture (ISA). The Alpha is a 64-bit RISC architecture designed for high-speed implementations. The instructions are very simple, with all memory operations in the form of loads and stores, and all data manipulation done between registers [Witek, et al. 1998]. In many ways this tool is like a typical assembler targeting an Alpha processor although there are some key distinctions. An important property of MSSP programs is they exist in two versions: original and distilled. Various mappings are required to compare distilled execution on the master processor to original execution on the slaves; special care is required to preserve the mapping between these two versions.

## Assembly Format
The assembly format is similar to that of traditional Alpha assembly code. Several annotations are added to the code to preserve the internal attributes of the distilled program. For example, to represent the mapping between distilled program addresses and original program addresses, each distilled block has the corresponding program counter (PC) of the original program as part of its header label. It is also important to preserve information about which basic blocks serve as speculative entries, function entries, path stubs, task boundaries, and verification checkpoints.

This information is written in the header label tag and the hexadecimal flag field of the block header. A sample function listing is shown in Figure 1.

Figure 1. Example of assembly format



```
func_entry_0x120012ca0:  (flags: 0x0)
entry_block_0x120012ca0:  (flags: 0x3)
fork_0x120012ca0:  (flags: 0x40)
        fork      0,verify_0x120012ca0
spec_entry_0x120012ca0:  (flags: 0x808810)
        ldq       r27,16(r10)
        lda       r4,4108(r4)
        cmplt     r4,r0,r17
        bne       r17,fork_0x120012ca0
block_0x120012cdc:  (flags: 0x26)
        jsr       r26,(r27)
                  Callee: func_entry_0x120030e00
                  Callee: func_entry_0x120031260
                  Callee: func_entry_0x1200327b0
block_0x120012cf4:  (flags: 0x400000)
        master_end
verify_0x120012ca0:  (flags: 0x80)
        verify
```

A. Block properties are represented in a 32-bit hexadecimal flag field. Block header names serve as a more readable indicator of the block type (e.g. entry_block). Also, the address of this block in the original program is included in the label.

B. The target of the fork instruction is the verify block below. Control flow references are resolved during the assembly and code layout steps.

C. Empty blocks are skipped during code layout, so their block flags are "pushed" to the next non-empty block. In this instance, the block appears to hold speculative entry code, when in fact the spec_entry block is an empty predecessor that has forwarded its block flags to a normal block. The user can add instructions to a speculative entry block by creating a block outside the control flow path. Flags must set for speculative entry and the block must be terminated with an unconditional branch to the successor.

D. Indirect jumps and calls are annotated with a list of potential targets in order to restore control flow edges in the IR.

Most instructions are disassembled using the `md_print_insn` function from the SimpleScalar supporting library [Burger and Austin 1997]. Branch target offsets are replaced with their corresponding block label, leaving the assembler to perform the work of recalculating instruction offsets.

A special set of control instructions is the group of indirect jumps and calls. Since the target offset is based on the contents of a register, the jump destination is not clear from the assembly itself. Instead the distiller performs an analysis to generate a list of possible jump destinations. During disassembly, we iterate through this list, placing each target block label on a line below the indirect jump or call instruction.

Reading in Assembly

Once the desired changes are made to the disassembled code, the assembler must read in the file and reconstruct the distiller's internal representation (IR) of the modified trace. This procedure is outlined in Figure 2.

Figure 2. Reconstructing IR from assembly file.

```
For each function in assembly file:
      For each basic block in function:
            Build a new block with correct flags
            Add block to parent function
            Encode instructions for block according to machine
                  definition
            If needed, add a fall-through edge to succeeding block
            If last instruction has a control flow target, record
                  target label to be resolved later

Match control flow references with target blocks, adding CFG edges
For each function read in:
      Perform code layout on internal representation
```

The Alpha instructions are defined and implemented in the MSSP simulator's machine definition files. In order to parse the disassembled instructions, we used this information to record each instruction name, opcode, assembly format string, and input/output register set in a structure. When reading in the assembly file, the scanner searches this structure for the current instruction name. After locating the name, the assembler can find the format of the current instruction to properly encode the input and output register and immediate parameters.

The last important role of the assembler is to order the basic blocks into the correct control flow arrangement. Because the assembly code has each branch and function call offset replaced with a string referencing its target, the assembler must provide a way to resolve each target label reference to its block object in the internal representation. This is done by storing for each basic block a `<string, basic block>` pair that associates the target block's text label with its C++ class object. These pairs are stored in an STL map indexed by the target label string.

Each block that terminates with a control instruction also encodes a `<string, basic block>` pair associating the source block object with the text label of the target block. The source block pairs are stored in an STL multimap since there can be multiple source edges to a given target. Branch edges are resolved by iterating through the map of target blocks and adding an edge for each source block in the source multimap that references that target. Call edges are inserted in a similar manner.

Most basic blocks need a fall-through control flow edge to the succeeding block. Exceptions to this include jump, tail call, `master end`, and `verify` instructions. Unconditional branches are treated uniquely, since they are inserted only when the code layout routine cannot place the fall-through block immediately after its predecessor. Upon encountering an unconditional branch instruction, the assembler does not insert the instruction in the basic block, but rather adds a control flow edge to its target. Inserting the branch instruction is left to the code layout phase.

After basic blocks have been created, instructions encoded, and control flow set, the code layout phase can write the new distilled trace to memory. Finally, the MSSP simulator executes the trace and reports the performance statistics. To demonstrate the integrity of the reconstructed trace, we tested the code and were able to achieve equivalent results when tasks were not modified.

## Experimental Results

To illustrate a situation where the assembler is useful, we identified a candidate function `sub_penal` from the *SPEC CPU2000 twolf* benchmark (Figure 3). This function has a loop that takes the same inner branch on all but the first and last iteration. Because of the inner branch's unique structure, we were able to "peel" the initial branch above the loop and the final branch below the loop by modifying the disassembled code of the distilled trace. To stay consistent with the original code, we replicated the fork instruction at the head of each peeled iteration. We took advantage of this peeled loop structure by inserting a check in the `spec_entry` block to branch to the current iteration's code. This avoids needlessly repeating cycles of the loop when a task misspeculation occurs on the final branch, for example.

Another performance bottleneck we identified was that every iteration applied a pair of absolute value operations. For each absolute value, the compiler inserted conditional branches to determine the sign of the operand. Our profile data show these branches suffer frequent mispredictions. We modified the disassembled loop to replace the absolute value computations with conditional move instructions. Finally, we assembled these changes and tested them in the simulator. In our test run of one hundred million instructions, `sub_penal` accounted for only 4.3% of the dynamic function calls; however, the performance impact of these changes was a 1.4% speedup overall.

When used with good profile data, the assembler can be used to quickly evaluate the effectiveness of a performance hypothesis. This will be useful as the distiller is extended to handle more aggressive optimizations for approximate code.

Figure 3. C code listing of function `sub_penal` from *SPEC CPU2000 twolf* benchmark.

```
sub_penal( startx , endx , block , LoBin , HiBin )
int startx , endx , block , LoBin , HiBin ;
{

BINPTR bptr ;
int bin ;

if( LoBin == HiBin ) {
    bptr = binptr[block][LoBin] ;
    newbinpenal     -= ABS( bptr->nupenalty ) ;
    bptr->nupenalty -= endx - startx          ;
    newbinpenal     += ABS( bptr->nupenalty ) ;
} else {
    for( bin = LoBin ; bin <= HiBin ; bin++ ) {
        bptr = binptr[block][bin] ;
        if( bin == LoBin ) {
            newbinpenal     -= ABS( bptr->nupenalty ) ;
            bptr->nupenalty -= bptr->right - startx   ;
            newbinpenal     += ABS( bptr->nupenalty ) ;
        } else if( bin == HiBin ) {
            newbinpenal     -= ABS( bptr->nupenalty ) ;
            bptr->nupenalty -= endx - bptr->left      ;
            newbinpenal     += ABS( bptr->nupenalty ) ;
        } else {
            newbinpenal     -= ABS( bptr->nupenalty ) ;
            bptr->nupenalty -= binWidth               ;
            newbinpenal     += ABS( bptr->nupenalty ) ;
        }
    }
}
}
```

---

## References

Burger, D. C. and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0". Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

Systems Performance Evaluation Corporation. SPEC benchmarks. http://www.spec.org.

Witek, Richard T. and Alpha Architecture Committee. Alpha Architecture Reference Manual. 3rd edition. Maynard: Digital Press, 1998.

Zilles, Craig. "Master/slave speculative parallelization and approximate code," Ph.D. dissertation, Computer Sciences Department, University of Wisconsin–Madison, Aug. 2002.