Instruction Trace Format

Instructor: Prof. Milo Martin, University of Pennsylvania
Last Updated: September 2011

Overview

Computer architects use simulation to quantitatively understand performance and inform design decisions. To allow you to perform similar experiments in your homework assignments, we have created an instruction trace format that describes the dynamic execution of the program. The intention is that this single trace format will be sufficient for analysing a variety of aspects of a microprocessor (for example: instruction mix, cache simulation, branch prediction simulation, pipeline simulation, etc.).

The trace format has one line per x86 micro-op executed by the program. Each line has all the same fields and each of those fields is described below. Not all fields are used by all instructions, but to keep the trace file format simple, those unused fields contain some default value. Fields are separated by one or more whitespace character.

To assist you in getting started, we have written a simple trace parser and provided you the source code (both a version written in C and one in Java). See below for more information. If you choose to use another language (not recommended), you will need to write your own parser.

Example Trace

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
Uop	**PC**	**Src1**	**Src2**	**Dest**	**Flags**	**Branch**	**Ld/St**	**Immediate**	**Mem. Address**	**Fallthrough PC**	**Target PC**	**Macro Opcode**	**Micro Opcode**
1	48d1de	-1	-1	13	-	-	-	0	0	48d1e2	0	SET	ADD
2	48d1de	-1	-1	13	R	-	-	1	0	48d1e2	0	SET	ADD_IMM
1	48d1e2	-1	5	45	-	-	L	-264	7fffe7ff048	48d1e9	0	CMP	LOAD
2	48d1e2	45	3	44	W	-	-	0	0	48d1e9	0	CMP	SUB
1	48d1e9	-1	5	3	-	-	L	-200	7fffe7ff088	48d1f0	0	MOV	LOAD
1	48d1f0	-1	-1	0	-	-	-	0	0	48d1f3	0	SET	ADD
2	48d1f0	-1	-1	0	R	-	-	1	0	48d1f3	0	SET	ADD_IMM
1	48d1f3	-1	-1	12	-	-	-	0	0	48d1f6	0	XOR	ADD
1	48d1f6	13	0	13	W	-	-	0	0	48d1f9	0	OR	OR
1	48d1f9	-1	-1	-1	-	T	-	54	0	48d1fb	48d231	JMP	JMP_IMM
1	48d231	-1	3	0	-	-	L	0	7fffe7fefd0	48d234	0	MOV	LOAD
1	48d234	0	0	44	W	-	-	0	0	48d237	0	TEST	AND
1	48d237	-1	-1	-1	R	N	-	-25	0	48d239	48d220	J	JMP_IMM
1	48d239	-1	0	7	-	-	L	8	12ff228	48d23d	0	MOV	LOAD
1	48d23d	-1	7	44	-	-	L	0	12ff200	48d240	0	MOVZX	LOAD

Field Descriptions

The fields are:

1. **Uop** is a counter for the micro-ops within a macro-op. If you look at the first two lines of the example trace above, you'll notice that they are part of the same macro-op. The first micro-op has value 1, the second is 2, etc. Use this information to decide how many micro-ops there are in a particular macro-op. [32-bit decimal]
2. **PC** is the address of the instruction being executed. [64-bit hexadecimal]
3. **Src1** is the first source register. A value of -1 indicates that no register is read for the source. [32-bit signed decimal]
4. **Src2** is the second source register. A value of -1 indicates that no register is read for the source. [32-bit signed decimal]
5. **Dest** is the destination registers. A value of -1 indicates that no register is written. [32-bit signed decimal]
6. **Flags** indicates if the instruction reads or writes the condition codes (also know as the flags in x86). This field is a single character that signifies if the micro-op reads the flags ('R'), writes the flags ('W'), or neither ('-'). [Single character]
7. **Branch** indicates if a micro-op is "taken" to the target PC ('T'), "not taken" and thus proceeds to the fall through PC ('N'), or a non-branch micro-operation ('-'). [Single character]
8. **Ld/St** indicates if a micro-op loads from memory ('L'), stores to memory ('S'), or neither ('-'). [Single character]
9. **Immediate** indicates the value of any immediate to the instruction. [64-bit signed decimal]
10. **Memory Address** indicates the address accessed by a load or store. If the micro-op doesn't access memory, this field will have the value zero. [64-bit hexadecimal]
11. **Fallthough PC** indicates the address of the instruction that immediately follows this instruction in memory; that is, if this instruction is a not-taken branch, this field holds the address of the instruction that will be executed. Note that the trace format is showing micro-ops; that is why you see the

same address multiple times in a row. All micro-ops that are part of the same instruction will share the same fall-through address. [64-bit hexadecimal]

12. **Target PC** indicates the address that a branch will go to if the branch is taken. This field is set to zero for non-control instructions. [64-bit hexadecimal]
13. **Macro opcode** is the printable string that corresponds to the x86 assembly instruction. This is the same for all the micro-ops in the macro-op. [String]
14. **Micro opcode** is the printable string that describes the specific micro-ops used in our simulation. [String]

Parser Code

We've provided simple parser for you in both C and Java.

- simulator.c
- Simulator.java

In both cases, the parser expects the trace file on the standard input stream.

To compile and run the C code, you can log into minus.seas.upenn.edu via ssh and run the following commands:

```
cd ~
mkdir trace-simulator
cd trace-simulator
cp ~cis501/html/traces/simulator.c .
gcc -Wall -Wextra -pedantic -std=c99 -O2 -lm simulator.c -o simulator
zcat ~cis501/html/traces/sjeng-10M.trace.gz | ./simulator
```

To compile and run the Java code, you can log into minus.seas.upenn.edu via ssh and run the following commands:

```
cd ~
mkdir trace-simulator
cd trace-simulator
cp ~cis501/html/traces/Simulator.java .
javac Simulator.java
zcat ~cis501/html/traces/sjeng-10M.trace.gz | java Simulator
```

In either case, after 30 seconds or so, the following should be displayed:

```
Processing trace...
Processed 10000000 trace records.
Micro-ops: 10000000
Macro-ops: 7569542
```

If you want to work on the code on your personal machine, you can download the trace file (sjeng-10M.trace.gz) to your machine. If your system doesn't support "zcat", you'll need to un-gzip the file. The parser code above also supports reading from a non-compressed file by giving it the file name as a command line parameter.

Trace Files

The trace file we'll be using is 87MB compressed (and 847MB uncompressed). Any of the engineering school's CETS-supported machines can access this file using the following path: ~cis501/html/traces/sjeng-10M.trace.gz

For debugging, we've also provided a version of the trace with just the first 1000 lines of the trace: ~cis501/html/traces/sjeng-1K.trace.gz

If you'd like to download the traces to your personal machine, you can access them at:

- sjeng-10M.trace.gz
- sjeng-1K.trace.gz

Longer Trace Files

The following trace files are from SPEC benchmarks. They are 50 million or 100 million micro-ops long. They are up to 1GB compressed (3GBs total for the six traces) and even larger uncompressed. You can also access them directly from an CETS-supported system at: ~cis501/html/traces/

- gcc-50M.trace.gz (The GCC compiler)
- go-100M.trace.gz (Plays the game of Go)
- hmmer-100M.trace.gz (Biosequence analysis using profile hidden Markov models) - http://www.spec.org/cpu2006/Docs/456.hmmer.html
- libquantum-100M.trace.gz (Simulation of quantum mechanics) - http://www.spec.org/cpu2006/Docs/462.libquantum.html
- sjeng-100M.trace.gz (Plays the game of Chess) - http://www.spec.org/cpu2006/Docs/458.sjeng.html
- sphinx3-100M.trace.gz (Speech to text program from CMU) - http://www.spec.org/cpu2006/Docs/482.sphinx3.html
- art-100M.trace.gz (Image recognition using neural networks) - http://www.spec.org/cpu2000/CFP2000/179.art/docs/179.art.html
- mcf-100M.trace.gz (Vehicle scheduling) - http://www.spec.org/cpu2006/Docs/429.mcf.html

Acknowledgments

The x86 traces were generated by a simulator written by Drew Hilton and Prof. Amir Roth with modifications by Jim Anderson. The trace parsers were written by Cem Karan, Jim Anderson, and Prof. Milo Martin.

Addendum

- None, yet

Generated on: 2011-09-18 20:49 UTC.