10
6-20-90 1
32

UCID-21568

DIAGNOSTICS ON THE S-1 ADVANCED ARCHITECTURE PROCESSOR (AAP)

Michael M. Murray

October, 1988

Lawrence Livermore National Laboratory

Price Code	Page Range
A01	Microfiche
Papercopy Prices	
A02	1- 10
A03	11- 50
A04	51- 75
A05	76-100
A06	101-125
A07	126-150
A08	151-175
A09	176-200
A10	201-225
A11	226-250
A12	251-275
A13	276-300
A14	301-325
A15	326-350
A16	351-375
A17	376-400
A18	401-425
A19	426-450
A20	451-475
A21	476-500
A22	501-525
A23	526-550
A24	551-575
A25	576-600
A99	601 & UP

UCID--21568

DE90 011762

Diagnostics on the S-1 Advanced Architecture Processor (AAP)

Abstract

The logical design of the AAP was extensively tested using diagnostic programs before construction of the wire-wrapped AAP was begun. This paper describes how the diagnostics were written and used, and some of the reasoning behind the more interesting diagnostics. A diagnostic generating program is also described.

[illegible]

Each diagnostic was an AAP assembly language program which had been designed to exercise some feature(s) of the hardware design. The diagnostics were written in AAP assembly language because of the high degree of control assembly language programming gives to the diagnostic programmer. In many diagnostics, a hardware feature could be tested only if the machine executed a unique sequence of AAP instructions, or addressed specific locations in memory. Such control would not have been possible if a high level language were used.

Each diagnostic was initially executed on the AAP architecture simulator, or archsim. Archsim served as the standard of correctness for AAP programs; given a program and data, archsim and a debugged hardware AAP should produce identical results. Thus, the first step in using a diagnostic was to verify that the diagnostic itself was debugged, i.e., the diagnostic would find no errors in a correctly operating machine. The debugged diagnostic was then executed on the SCALD simulator which had been loaded with the AAP hardware design. A diagnostic would terminate "normally" if an error was found. Normal and abnormal terminations were determined by the final value of the program counter. When a diagnostic terminated abnormally, the machine register contents were preserved, which allowed a preliminary diagnosis of the problem to be made. The diagnostic could then be rerun with the SCALD simulator in single step mode, and the exact malfunction could be identified.

The diagnostics were divided into two categories: custom diagnostics and pseudo-random diagnostics. Each custom diagnostic was a carefully chosen instruction sequence designed to exercise a particular feature of the hardware design more rigorously than a typical high level language program would. Pseudo-random diagnostics tested the machine as a whole. A pseudo-random diagnostic instruction sequence resembled a HLL program instruction sequence, but included fine granularity checking for a machine malfunction.

The STORE and GOTO diagnostics are the best examples of the custom diagnostics. The STORE diagnostic did basic testing of the load and store instructions. It exemplified the principle of testing the design with a demanding, but legal instruction stream. High level language programs typically perform a load or store instruction once every 4 instructions. The STORE diagnostic performed 54 consecutive load and store instructions, including a sequence which caused 9 consecutive data cache misses. It also performed stores to a location X followed by reloads from X, with varying numbers and types of intervening instructions. STORE verified the design of the write register, write buffer, and data cache miss logic.

The GOTO diagnostic tested instruction sequencing: HLL language programs typically execute a goto assembly language instruction once every 6 instructions. The GOTO diagnostic consisted of 24 instructions, 18 of which were gotos. Nine of the gotos were consecutive and caused 9 consecutive instruction cache misses. GOTO thus subjected the instruction branching logic to much more frequent use than HHL language program would require. GOTO tested the immediate and delayed forms of the goto instruction. The 6 non-goto instructions in GOTO were placed in the delay slots of delayed gotos, and caused side effects which verified that they had been executed.

Other custom diagnostics were written to test the ALU, tags, condition testing, and data and instruction cache flushing.

The pseudo-random diagnostics complemented the custom diagnostics by taking a Monte Carlo approach to testing. Rather than searching for a specific design weakness, the Monte Carlo approach postulates the existence of an unspecified flaw. By definition, this flaw must cause an incorrect result to be generated by some instruction. Hence, the flaw can be disclosed by executing a sufficiently long instruction sequence. The goal of the diagnostic programmer becomes the generation of that instruction sequence.

Long instruction sequences are better generated by computers than by humans. Consequently, a Diagnostic Generating Program (DGP) was designed, and an initial prototype was written. The goal of the DGP was to generate diagnostics consisting of tens of thousands of instructions in several minutes. The instructions in each diagnostic would be pseudo-random, i.e., the DGP user could specify certain parameters that the instruction stream would satisfy overall, but the actual instruction sequence would be generated randomly by the DGP. The user controlled parameters would include instruction mix, instruction and data cache hit rate, memory reference address mix, and instruction result interlocking. DGP diagnostics would also be suitable for use in benchmarking and performance tuning experiments.

A DGP generated diagnostic consisted of two sections. The first section consisted of the pseudo-random sequence specified by the user. The second section consisted of code that verified that the first section had executed correctly. While generating the first section, the DGP performed the same calculations as would be performed when the first section was executed. Thus, the second section consisted of the results of these calculations, plus instructions to compare these predicted results with the actual results from the first section.

The prototype DGP allowed the user to specify cache and memory reference parameters such as data cache hit rate, and frequency of access to the same word, same cache line, or next cache line as the previous access. The prototype DGP produced diagnostics with an instruction of 25% loads, 35% moves, and 40% stores. One DGP generated diagnostic was run on the SCALD simulator. It found an error in one of the SCALD library entries.

Conclusion

The logical design of the AAP was extensively tested using assembly language diagnostic programs before construction of the wire-wrapped AAP was begun. Consequently, the wire-wrapped AAP was relatively free of design bugs, and debugging consisted primarly of locating defective chips and interconnections.