DIVISION OF COMPUTER SCIENCE

An Evaluation of the HARP ORed Indexing Addressing Mechanism

F. L. Steven

Technical Report No.156

July 1993

ABSTRACT

Several different addressing mechanisms are evaluated within the context of VLIW and superscalar processor design. The results suggest that traditional RISC addressing mechanisms are less effective than the other simpler addressing mechanisms considered. In particular, the ORed indexing addressing mechanism significantly improves performance.

Key Words: RISC, VLIW, Superscalar, Addressing Mechanisms

1 INTRODUCTION

Traditional processors fetch and execute instructions one at a time, taking several cycles to complete each instruction. More recently, Reduced Instruction Set Computers (RISC) have approached single cycle execution rates by overlapping the instruction fetch and execute phases of successive instructions [Tabak 90]. However, to obtain an execution rate in excess of one instruction per cycle it is necessary to fetch and execute multiple instructions in parallel.

Execution of multiple instructions per cycle can be achieved by two different approaches. VLIW(Very Long Instruction Word) processors[Fisher 83] rely entirely on the compiler to schedule instructions into long instruction words so that they can be fetched and executed in parallel. In contrast, superscalar processors [Johnson 91] rely on instructions being scheduled dynamically at run-time.

The computer architecture group at the University of Hertfordshire has developed iHARP[Steven 89a, 92], a VLIW(Very Long Instruction Word) processor. The aim of the HARP project is to develop a processor which can execute instructions significantly in excess of one instruction per cycle. To date, rates in excess of two instructions per cycle have been achieved[Wang 93].

During each machine cycle iHARP fetches multiple instructions in the form of a long instruction word from an instruction cache and passes the component instructions directly to four separate pipelines for execution. The HARP compiler is responsible for making sure that instructions which can be executed in parallel are grouped together in a single LIW.

The computer architecture group also intend to develop a superscalar processor and the work outlined in this paper will help to decide the type of architectural features that will be incorporated into this processor.

This paper evaluates the performance of four different addressing mechanisms which are listed below:

- iHARP addressing mechanisms; offset(Ri), (Ri,Rj) with ORed indexing.
- Traditional RISC addressing modes; offset(Ri), (Ri,Rj).
- Register indirect and direct addressing only.
- Register indirect addressing, direct addressing and an additional offset(SP) addressing mode.

In particular, the performance of ORed indexing is examined to deduce whether its inclusion in the HARP architecture improves performance.

Section 2 describes a typical RISC processor pipeline and shows how this model has been altered in the iHARP processor. In section 3 the addressing mechanisms compared in this study are described. Section 4 presents the results. These results are then discussed in section 5 with concluding remarks in section 6.

2 iHARP PIPELINE

A typical RISC processor pipeline[Hennessy 90] requires five stages to execute a load instruction:

IF:	Instruction Fetch.
RF:	Register Fetch.
ALU:	ALU/Shift operation/Calculate memory address.
MEM:	Wait for data from memory on a load instruction or output data on a store instruction.
WB:	Write result to destination register.

A load instruction uses all five stages in the pipeline with the result of the load operation available at the end of the MEM stage. Consequently, a delay of one cycle is required before the result can be used in the next sequential instruction (Fig 1).

In contrast, iHARP uses a four stage pipeline:

IF: Instruction Fetch.
RF: Fetch register operands from the register file.
ALU/MEM: Perform operation or access data.
WB: Return results to register file.

Two changes have been made to the original five stage pipeline. First, address computation has been moved to the RF stage. Second, the ALU and MEM stages have been combined. This second change is possible because loads and stores no longer require the ALU stage to compute memory addresses and other operations do not use the MEM stage. The result of a load operation is therefore available in the ALU/MEM stage and can be bypassed directly to the next sequential instruction with no load delay (Fig 2).

The key to moving the address computation to the RF stage is to reduce the computation to a bitwise logical OR between the two address components. A logical OR is equivalent to an addition if no carries are generated. This condition is met if the address components never have a logical one in the same bit position.

In order to use a logical OR in address computation, the compiler ensures that the bottom n bits of the stack pointer are always zero by always aligning the stack pointer on a power of two (2^n) memory boundary. A second address component or offset can then be safely added to SP using a logical OR. Glew[Glew 89] suggested the term "ORed indexing" to describe this method.

3 ADDRESSING MECHANISMS CONSIDERED

To evaluate the alternative addressing mechanisms four GNUcc C compilers were produced[Stallman 88]. The first compiler was the standard GNUcc iHARP compiler[Wang 91]. This compiler was modified to produce three further compilers, each incorporating one of the alternative sets of addressing modes listed in section 1. Different epilogues and prologues were also generated for procedure entry and exit. Apart from the addressing modes the iHARP instruction set was used throughout.

3.1 iHARP ADDRESSING MECHANISMS

This compiler used the iHARP addressing mechanisms which comprised offset(Ri) and (Ri, Rj) where Ri and Rj are any registers. Since register R0 is always zero register indirect and direct addressing were also available. Two additional features included were:

1) ORed indexing.
2) Rearranged array accessing.

3.1.1 ORED INDEXING

To use ORed indexing, the code generated on procedure entry must align the stack pointer on a power of two memory boundary. Local data can then be safely accessed relative to the stack pointer using ORed indexing. The worst case procedure entry and exit code is shown below:

```
        {parameters placed in registers or on stack}
        BSR RA, _proc                 /*return address placed in return address register*/
        NOP

_proc:  MOV SP', SP                   /*save old stack pointer*/
        AND SP, SP, #mask             /*force SP to 2^n boundary*/
        SUB SP, SP, #frame_size       /*space allocated for new stack frame*/
        ST 4(SP), SP'                 /*save old SP*/
        ST 0(SP), RA                  /*save return address on stack*/
              .
              .
        LD RA, 0(SP)                  /*return address placed in register*/
        LD SP, 4(SP)                  /*restore old SP*/
        MOV PC, RA                    /*return from procedure*/
```

The stack frame size is rounded up to the next 2^n boundary by the compiler. On entry into the procedure body the value of the old SP is saved. The AND instruction then rounds down the stack pointer to the nearest 2^n boundary by forcing the bottom n bits to zero. The new stack frame is then allocated by subtracting the frame size from SP. On exit from the procedure the old SP is restored.

In the worst case all the above code is required. However, most iHARP procedures use a standard minimum stack frame size of 128 bytes. Since SP is always aligned on a 128 byte boundary, this removes the need to save the old stack pointer and to realign SP before allocating a new stack frame. As a result the overwhelming majority of iHARP procedure calls incur no additional overhead to support ORed indexing. The simplified procedure entry and exit code is shown below:

```
        {parameters placed in registers or on stack}
        BSR RA, _proc                 /*return address placed in return address register*/
        NOP

_proc:  SUB SP, SP, #128              /*space allocated for new stack frame*/
        ST 0(SP), RA                  /*save return address on stack*/
              .
              .
              .
        LD RA, 0(SP)                  /*return address placed in register*/
        ADD SP, SP, #128              /*stack frame space deallocated*/
        MOV PC, RA                    /*return from procedure*/
```

A particular case for a stack frame requiring 27 bytes of memory is shown in Fig 3.

3.1.2 REARRANGED ARRAY ACCESSING

To access the element of an array on the stack the iHARP compiler originally generated the following code sequence:

```
        ADD R5, R7(ASL #2), SP        /*R5 = SP + (Index*4)*/
        ADD R5, R5, #8                /*R5 = R5 + offset*/
        LD R5, (R0, R5)               /*Access element of array*/
```

In this sequence the stack pointer is used in the first stage of the address computation. However, in order to take advantage of the ORed indexing mechanism this sequence of instructions was altered in the final peephole pass so that the stack pointer was a component of the final stage of the address computation. This new code sequence is shown below:

```
ADD R5, R7(ASL #2), #8          /*R5 = (Index*4) + offset*/
LD R5, (R5, SP)                 /*Access element of array*/
```

In the rearranged code sequence, because SP is on a power of two boundary, a bitwise logical OR can be performed on SP and R5 to produce the address required. As a result, one less addition instruction is required and the overhead on array accessing is reduced.

3.2 TRADITIONAL RISC ADDRESSING MODES

All the addressing modes found on a traditional RISC architecture are available.

```
LD Rk, offset(Ri)    ;   Ri is any register.
NOP                  ;   NOP indicates a load delay of one cycle.
ST offset(Ri), Rk    ;   Ri is any register including GP(Global Pointer),
                     ;   SP, and R0(always zero). No delay.
LD Rk, (Ri, Rj)      ;   Ri and Rj are any register.
NOP                  ;   NOP indicates a load delay of one cycle.
ST (Ri, Rj), Rk      ;   Ri and Rj are any register including GP, SP and R0. No delay.
```

The load delay shown above is required because address computations must be carried out during a separate ALU pipeline stage. As a result, the data cache access is delayed and the load operand is not available to the immediately succeeding instruction.

A traditional stack frame layout, which allocated only enough space for any locals and registers saved, was used for procedure entry and exit.

3.3 REGISTER INDIRECT ADDRESSING

It was seen in the previous section that a load delay of one is required to implement traditional RISC addressing mechanisms. One way of removing this load delay is to use a restricted set of addressing modes so that an address computation in the ALU stage is not required. As previously described one way is to use ORed indexing. An alternative is to restrict the addressing modes available to register indirect and direct addressing. A load delay is, therefore, not required as there are no offsets involved in the computation of a memory address. The same addressing modes are provided on the Am29000[Johnson 87] and on VIPER[Abnous 92].

The stack frame organisation used was identical to that used in the previous section.

3.4 DEDICATED SP ADDRESS ADDER

As with ORed indexing, another way to perform address computations in the RF stage is to treat the stack pointer as a separate register by removing it from the general register file, and providing a dedicated adder which operates in the RF stage to compute (SP) plus offset.

The addressing modes included were identical to those used by the previous option except an additional mode using offset(SP) is included. Therefore, the instructions LD Rk, offset(SP) and ST offset(SP),Rk are available to access operands relative to the stack frame.

4 PROCEDURE AND RESULTS

Section 4.1 outlines the procedure used to analyse the various addressing mechanisms described in section 3. The number of cycles required to execute eight Stanford benchmarks compiled by the separate compilers is shown in section 4.2.

4.1 PROCEDURE

Four separate C compilers were constructed from the original iHARP C compiler with each one incorporating one of the alternative addressing mechanisms described in section 3.

Eight Stanford benchmarks were then compiled by the four compilers. The branch and load delay slots of the resulting 32 HARP code programs produced by the four compilers were filled where appropriate[Wang 93]. On average 73% of the load and branch delay slots were filled by the scheduler.

The 32 HARP code programs were also scheduled[Wang 93] for the four pipeline iHARP processor. To avoid distorting the results, four register file write-backs per cycle were assumed.

The resulting scheduled code was then run on a HARP simulator[Whale 92]. The number of cycles required and the dynamic instruction count was recorded. These results are shown in section 4.2, Tables 1-4.

4.2 RESULTS

TABLE 1 SERIAL EXECUTION TIME (CYCLES)

Program	ORed Indexing	Trad. Modes	Register Indirect	SP Address Adder
Bubblesort	48946	44552	48962	48946
Multiply Matrix	235624	236232	238084	235624
Permute	16032	16940	19346	16046
Puzzle	75076	77492	75316	75076
Queens	47654	49858	52174	47648
Quick	48632	51242	49716	48632
Tower	33106	34610	37798	33106
Tree	72790	78894	74806	72790
Average	72232.5	73727.5	74525.3	72233.5
Harmonic mean	43042.9	44504.8	47422.3	43054.9

TABLE 2 PARALLEL EXECUTION TIME (CYCLES)

Program	ORed Indexing	Trad. Modes	Register Indirect	SP Address Adder
Bubblesort	26510	29562	26510	26510
Multiply Matrix	95570	96578	95570	95570
Permute	12010	13316	12010	12010
Puzzle	39302	41816	39302	39302
Queens	29278	31290	29502	29274
Quick	28698	31382	28876	28698
Tower	22952	25804	22952	22952
Tree	47818	54198	47818	47818
Average	37767.3	40493.3	37817.5	37766.8
Harmonic mean	27550.5	30277.5	27595.6	27550.1

TABLE 3 PARALLEL INSTRUCTION COUNT (BYTES)

Program	ORed Indexing	Trad. Modes	Register Indirect	SP Address Adder
Bubblesort	25871	19505	25879	25871
Multiply Matrix	37709	35709	38939	37709
Permute	7994	7994	9644	7994
Puzzle	48877	46825	48997	48997
Queens	37251	37695	39189	37248
Quick	18704	18704	19346	18704
Tower	17873	17189	19156	17873
Tree	31733	31631	32741	31733
Average	28251.5	26906.5	29236.4	28254.9
Harmonic mean	21136.2	20227.2	22963.6	21138.8

TABLE 4 RELATIVE PERFORMANCE OF ADDRESSING MODES

RANK	CYCLE COUNT SERIAL CODE (% increase*)		CYCLE COUNT PARALLEL CODE (% increase*)		DYNAMIC INSTRUCTION COUNT(% increase*)	
1	ORed Indexing	(1.00)	SP Address Adder	(1.00)	Trad. modes	(0.96)
2	SP Address Adder	(1.00)	ORed indexing	(1.00)	ORed indexing	(1.00)
3	Trad. modes	(1.03)	Reg indirect	(1.00)	SP Address Adder	(1.00)
4	Reg indirect	(1.10)	Trad. modes	(1.10)	Reg indirect	(1.09)

*ORed indexing = 1.00.

5 DISCUSSION

The cycle count for the serial code shows that ORed indexing and the use of a dedicated SP address adder produced the best performances. Traditional RISC addressing modes degrade performance by 3% while using register indirect addressing degrades performance by 10%.

The move from serial to parallel code significantly changes the relative performance. With parallel code, ORed indexing, register indirect addressing and a dedicated stack pointer address adder perform equally well, while using traditional addressing mechanisms degrades performance by 10%. The VIPER group[Abnous 92] also found that using register indirect addressing in a VLIW environment yielded a similar 8.4% performance advantage over traditional addressing modes.

In all cases the performance advantage over the traditional addressing modes is achieved by executing more instructions. Using ORed indexing or a dedicated SP address adder requires 4% more instructions, while with register indirect addressing 14% more instructions are executed.

Traditional addressing modes perform relatively well in a single pipeline because the compiler is usually able to hide the load delay by scheduling useful instructions in the load delay slot. In contrast, in parallel code any instruction which can be used to fill load delay slots can now be executed in parallel with the load instructions. As a result increasing the load instruction latency has a greater impact on execution time in parallel code.

In contrast register indirect addressing performs significantly better in a parallel environment. This improved performance is a direct result of a VLIW processor's ability to precompute addresses in parallel with other instructions. While these address computations increase the instruction count, the impact on performance is minimal.

Having discussed the performances of the addressing mechanisms, the implementation considerations are now examined.

ORed indexing removes the requirement for a load delay and since both loads and ALU operations use four pipeline stages the instruction scheduler can easily compute the number of write-backs required in each cycle. However, the major disadvantage of ORed indexing is the wasted space on the stack frame and this overhead does not impact directly on cycle and instruction counts.

One of the disadvantages of using traditional RISC addressing modes is the implementation cost in a VLIW or superscalar processor. Implementing this mechanism could be achieved in one of two ways. First, the ALU write-back operation could be delayed for one cycle. This is done in both DLX[Hennessy 90] and MIPS-X[Chow 87]. However, using this scheme results in the bypassing requirements for a four pipeline model similar to iHARP increasing from four ALU results bypassed to eight ALU inputs[Steven 92] to eight, ALU results being bypassed to eight ALU inputs. This extra routing would be very costly in terms of silicon area. Second, only the write-back for the load instruction could be delayed. As a result, the number of write ports required would have to be increased to accommodate the possibility of two write-backs coinciding, for example, one write back from a load and one from a succeeding ALU operation which still only requires four stages. Alternatively, if the number of write-backs was restricted it would be more difficult for the compiler to predict the number of write-backs occurring in each cycle.

Register indirect addressing requires no load delay, an advantage it shares with ORed indexing. However, as it is a subset of the iHARP addressing modes it can never do quite so well in terms of both performance and flexibility.

Although the use of an additional dedicated adder produces excellent results it would, nevertheless, scale very badly, especially if using multiple data cache ports. For example, if two loads per cycle were required a dedicated adder would be required for each load to add an offset to the SP. Therefore, a dedicated adder would be required for every port. In general, this mechanism is the least flexible of the alternatives examined. It is very dependent on the most common addressing mode being SP plus offset. It would be less flexible if many address pointers were required. The other addressing mechanisms are more consistent since they allow every address register to be treated in exactly the same way in address calculations. Therefore, on balance, this scheme is less attractive than the performance figures suggest.

6 CONCLUSIONS

The results support the HARP decision to use ORed indexing. It has produced the best performance with the cheapest implementation. Its other advantages include no load delay, ease for scheduling and a simpler mechanism to compute addresses. The one overhead that is incurred, in procedure entry and exit, does not significantly degrade performance. The dedicated SP address adder has done as well as ORed indexing but at the expense of additional hardware and reduced address functionality. Register indirect has done particularly well in terms of cycle count in parallel code, and especially so when considering its simple implementation. Therefore, in a VLIW or superscalar processor, the results suggest that ORed indexing or register indirect addressing will give a superior performance when compared to traditional addressing mechanisms.

REFERENCES

[Tabak 90]	Tabak D "RISC Systems", Wiley, New York, 1990
[Fisher 83]	Fisher J A "Very Long Instruction Word Architectures and the ELI 512" 10th Ann. Int. Symp. on Computer Architecture, June 1983, pp 140-150
[Johnson 91]	Johnson M "Superscalar Microprocessor Design", Prentice Hall, 1991
[Steven 89a]	Steven G B, Gray S M and Adams R G "HARP: A Parallel Pipelined RISC Processor" Microprocessors and Microsystems, Vol 13, No. 9, November 1989, pp 579-587
[Steven 92]	Steven G B, Adams R G, Findlay P A and Trainis S A "iHARP: A Multiple Instruction Issue Processor", IEE Proceedings, Part E, Computers and Digital Techniques, Vol 139, No. 5, September 1992, pp 439-449
[Wang 93]	Wang L "Instruction Scheduling for a Family of Multiple Instruction Issue Architectures", Division of Computer Science, PhD Thesis, University of Hertfordshire, Expected September 1993
[Hennessy 90]	Hennessy J L and Patterson D A "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 1990
[Steven 88]	Steven G B "A Novel Effective Address Calculation Mechanism for RISC Microprocessors", SIGARCH, Vol 16, No. 4, September 1988, pp 150-156
[Glew 89]	Glew A "ORed Indexing", personal communication, April 1989
[Stallman 88]	Stallman R M "Using and Porting GNU CC", version 1.36, Free Software Foundation Inc, 1989
[Wang 91]	Wang L "Crafting a C Compiler for the iHARP Chip using the GNU Compiler Compiler", Division of Computer Science, Technical Report No. 121, University of Hertfordshire, April 1991
[Steven 89b]	Steven F L "The Impact of Instruction Set Orthogonality and Complexity on Compiler Code Generation", Division of Computer Science, PhD Thesis, University of Hertfordshire, August 1989
[Johnson 87]	Johnson M "System Considerations in the Design of the AM29000", IEEE Micro, Vol 20, No. 8, 1987, pp 28-41
[Abnous 92]	Abnous A and Bagherzadeh N "Architectural Design and Analysis of a VLIW Processor", University of California, Irvine, Technical Report No. 92-79, October 1992
[Whale 92]	Whale D J "Development of a Processor Simulation for iHARP" Division of Computer Science, University of Hertfordshire, April 1992
[Chow 87]	Chow P and Horowitz M "Architectural Tradeoffs in the Design of MIPS-X" 14th Ann. Int. Symp. on Computer Architecture, Pittsburgh, June 1987, pp 300-307

Fig 1 DLX Pipeline Stages

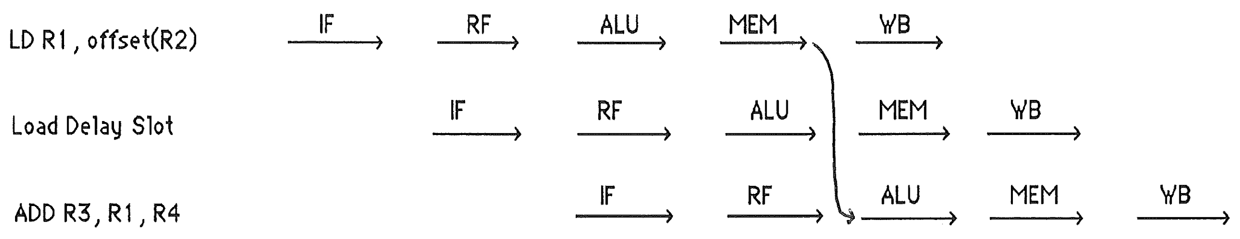


Fig 2 iHARP Pipeline Stages

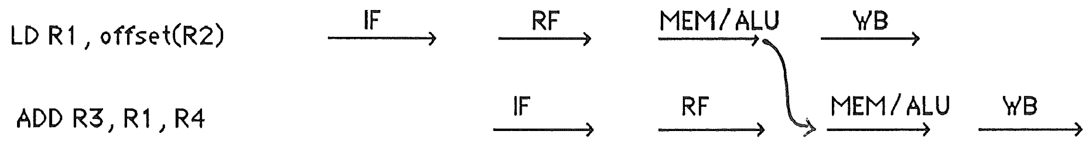
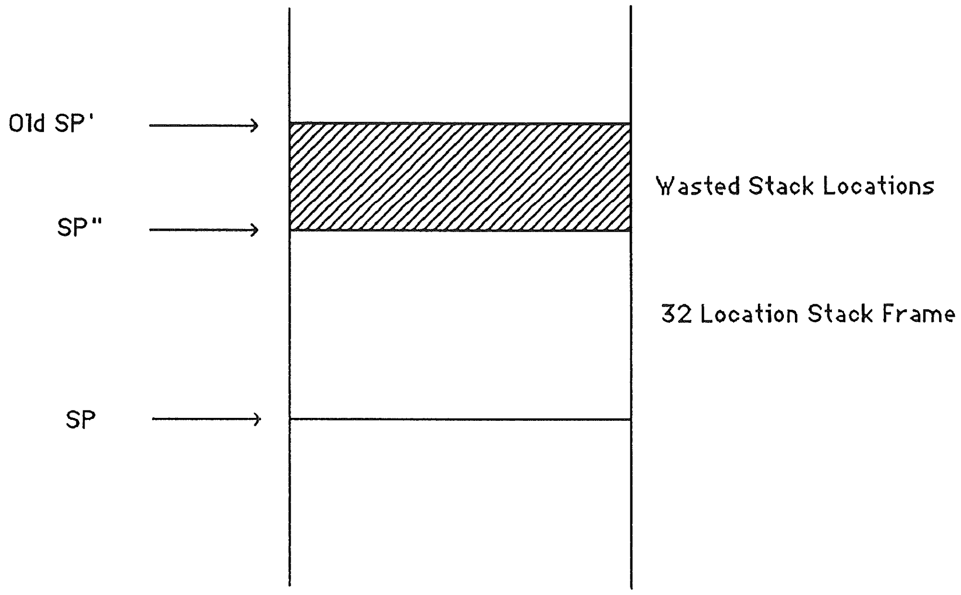


Fig 3 iHARP Stack Frame Allocation



1) SP' points to old stack frame.

2) AND SP, SP, #$FFFFFFE0 aligns SP on a 32 byte boundary wasting some stack locations in the procedure stack frame.

3) SUB SP, SP, #$20 allocates a 32 byte stack frame.