Contents

RISC vs CISC

- Reduced Instruction Set Computer *vs* Complex Instruction Set Computers
- For a given benchmark the performance of a partucular computer:
- $P = \frac{1}{I*C*\frac{1}{2}}$
  - where
    * P = time to execute

        * I = number of instructions executed
        * C = clock cycles per instruction
        * S = clock speed
- RISC approach attempts to reduce C
- CISC approach attempts to reduce I
- Assuming identical clock speeds:
  - $C_{RISC} < C_{CISC}$
  - A RISC will execute more instructions for a given benchmark than a CISC

RISC-I

- History
- RISC-1 designed by MSc students under the direction of David Patterson and Carlo H. Sequin at UCLA Berkeley
- Released in 1982
- First RISC now accepts to be in the IBM 801 (1980), but design not made public at the time
- John Cocke later won both the Turing award and the Presidential Medal of Science for his work on the 801
- RISC-1 similar to SPARC (Sun, Oracle) and BLX/MIPS (discussing its pipeline later)

Design Criteria

- For an effective single chip solution artificially placed the following design constraints
  - Execute one instruction per cycle
  - Make all instructions the same size (simplifies instruction decoding)
  - Access main memory with load and store instructions (load/store architecture)
  - ONLY one addressing mode (Indexed)
  - Limited support for high level languages (C and hence Unix)

Architecture

- 32 x 32 bit registers r0 … r31 (r0 always 0)
- PC and PSW (status word)
- 31 different instructions
- Instruction formats:

7	1	5	5	1	13
opcode	scc	dst	src1	imm	src2

7	1	5	19
opcode	scc	dst	Y

- Opcode: 128 possible opcodes
- scc: if set, instruction updates the condition codes in PSW
- dst: specifies one of 32 registers r0…r31
- src: specifies one of 32 registers r0…r31
- imm, src2:
  - if(imm==0) then 5 lower order bits of src2 specifies one of the 32 registers r0…r31
  - if(imm==1) then src2 is a sign extened 13 bit constant
- Y: 19 bit constant/offset used primarily by relative jumps and ldhi (load high immediate)

Arithmetic Instructions

- 12 arithmetic instructions which date the form
  - $R_{dst} = R_{src1} op S_2$
  - S_2 specifies a register of an immediate constant
- Operations
  - add, add with carry, subtract, subtract with carry, reverse subtractm reverse subtract with carry
  - and, or, xor
  - sll, srl, sra (shifts register by S_2 bits)
  - **NB**: NO mov, cmp, …

Synthesis of some IA32 Instructions

- mov R_D, R_M -> add R_0, R_M, R_D
- cmp R_N, R_M -> sub R_M, R_N, R_0, {C}
- test R_N, R_N -> and R_N, R_N, R_0, {C}
- mov R_N, R_0 -> add R_0, R_0, R_N
- neg R_N -> sub R_0, R_N, R_N

- not R_N -> xor R_N, #-1, R_N (invert bits)
- inc R_N -> add R_N, #1, R_N
- Loading constants $-2^{12} < N < 2^{12} - 1$ (constant fits into src2 field)
  - mov R_N, N -> add R_0 #N, R_N
- Loading constants $(N < -2^{12} || (N > 2^{12} - 1)$ (constant too large for src2 field)
  - construct large constants using two instructions
  - mov R_N, N ->
    * add R_0 #N<12:0>, R_N (load low 13 bits from src2 field)
    * ldhi #N<31:13>, R_N (load high 19 bits from Y field)

Load and Store Instructions

- 5 load and 3 store instructions

ldl $(R_{src1})S_2$, R_{dst}	R_{dst}=$[R_{src1}+S_2]$	load 32 long
ldsu $(R_{src1})S_2$, R_{dst}	R_{dst}=$[R_{src1}+S^2]$	load short unsigned
ldss $(R_{src1})S_2$, R_{dst}	R_{dst}=$[R_{src1}+S_2]$	load short signed
ldbu $(R_{src1})S_2$, R_{dst}	R_{dst}=$[R_{src1}+S_2]$	load byte unsigned
ldbs $(R_{src1})S_2$, R_{dst}	R_{dst}=$[R_{src1}+S_2]$	load byte signed
stl $(R_{src1})S_2$, R_{dst}	$[R_{src1}+S_2]$=R_{dst}	store long
sts $(R_{src1})S_2$, R_{dst}	$[R_{src1}+S_2]$=R_{dst}	store short
stb $(R_{src1})S_2$, R_{dst}	$[R_{src1}+S_2]$=R_{dst}	store byte

- Load unsigned clears most significant bit of register
- Load signed extends sign across most significant bits of register
- Indexed addressing $[R_{src1}+S_2]$
- S_2 must be a constant

Register Windows

- Single cycle function call and return
- Need to consider parameter passing, allocation of local variables, sabing of registers, etc.
- "Since the RISC-I microprocessor core is so simple, there's plenty of chip area left for multiple register sets"

- Each function call allocates a new "window" of registers from a circular on-chip register file
- Scheme based on the notion that the registers in a register window are used for specific purposes

Num	Registers	Use
6	R26…R31	For parameters passed to this function
10	R16…R25	For local variables and immedate result
6	R10…R15	Parameters for next function
10	R0…R9	Global registers common to all functions

Organisation

- Example shows function A calling function B
- `CWP` (current window pointer) points to current register window in circular on-chip register file
- On a function call `CWP` moved so that a new window of registers `r10…r25` (16 registers) allocated from the register file
  - `r10…r15` of the calling function are now mapped onto `r26…r31` of the called function (used to pass parameters)

Function Call and Return

- The CALL and CALLR instruction take the form
  - CALL $S_2(R_{src1})$, R_{dst}
  - CWP <- CWP-1 ; Move to next register window
  - R_{dst} <- PC ; Return address saved in R_{dst}
  - PC <- R_{src1} + S_2 ; function start address
  - CALLR R_{dst}, Y
  - CWP <- CWP-1 ; move to next register window
  - R_{dst} <- PC ; return address saved in R_{dst}
  - PC <- PC + Y ; relative jum to start address of function
- The RET instruction takes the form
  - RET $(R_{dst})S_2$
  - PC <- R_{dst}+S_2
  - CWP <- CWP+1
- CALL and RET must use the same register R_{dst}

- In most cases, functions cal be called in a "single cycle"
  - Parameters stored directly in r10…r15
  - No need to save registers as a new register window allocated
  - Use new registers for local variables

Register File Overflow/Underflow

- Can run out of register windows if functions nest deep enough
- Register window overflow can **only** over on a CALL/CALLR
  - No need to save oldest register window on stack in main memory
- Register window underflow can **only** occur on a RET
  - There *must* always be at least two valid register windows in register file (window CWP constains registers r10…r25 and window CWP-1 constains r26…r31)
  - Need to restore register window from stack

Register File Overflow

- Typical register file overflow sequence
- `SWP` = save window pointer (points to oldest register window in register file)
- `SWP++` performed using modulo arithmetic as register file is *circular*
- `r1` used as a stack pointer

1. Function calls already 8 deep (register windows 0 to 7)
2. `CWP` -> register window 7, `SWP` -> register window 2 (oldest window)
3. Two register windows already pushed onto stack (register windows 0 and 1)
4. Another call will result in a register file overflow
5. Register window 2 pused onto the stack (pointed to by `SWP`)
6. `CWP` and `SWP` *move down* one window (`CWP++` and `SWP++`)

- `PSW` contains `CWP` and `SWP`
- Before a CALL/CALLR instruction is executed, the following test is made

```
if(CWP+1 == SWP)
    TRAP(register file overflow
```

- The trap handler must save the registers pointed to by `SWP` onto a stack in main memory

- How might this be done
  1. Instruction to switch to `SWP` window so that `R10…R25` can be saved on stack using standard instructions
  2. An instruction to increment `SWP`
  3. An instruction to move back to the `CWP` window so that `CALL` can be re-executed with `TRAP`

Register File Underflow

- Always need 2 valid register windows in the register file (`SWP` -> oldest valid register window)
- `SWP` window contains `CWP`'s `r26...r31`
- Following test made when `RET` executed

```
if(CWP-1 == SWP)
    TRAP(register file underflow)
```

- Pop data from stack to restore window `SWP-1`
- `CWP` and `SWP` *move up* one window (`CWP--` and `SWP--`)

Problems with Multiple Register Sets

- Must save/restore 16 registers on an overflow/underflow even though only a few may be in use
- Saving multiple register sets on a context switch (between threads and processors)
  - If many windows in the registery file, forced to save all of them
- Referncing variables held in registers *by address* (a register does **not** normally have an address)

```
p(int i, int* j) {
    *j = ... // j passed by address
}

C++ p(int i, int& j) { j = ... // j passed by reference }

q() {
    int i, j;
    ...
    p(i, &j);
    ...
}
```

Solution

- Solution proposed in the original Computer IEEE paper
- "RISC-I solves tha problem by giving addresses to the window registers."
- Register file can be thought of as sitting on the top of stack in memory
- Can then assigned a notional address to each register in register file (where is would saved on stack if an overflow occured)
- Inside `Q`, `j` can be accessed as a register
- Address of `j` passed to `p()`, compiler able of generation instructions to calculae its address relative to `r1` of where the register would be stored if spilled onto the stack
- `*j` in `p()` will be mapped by load and store instructions onto a register if the address "map" to the register file otherwise memory will be accessed

RISC-I Pipeline

- Two stage pipline - fetch unit and execute unit
- Normal instructions

fetch i1	execute i1		
	fetch i2	execute i2	
		fetch i3	execute i3

- Load/store instructions

fetch load	compute addr	mem access	
	fetch i2	stall	
			execute 2
			fetch i3

- Pipeline stall arises because it is **not** possible to access memory twice in the same clock cycle (fetch the next instruction and read/write target of load/store)
  - Harvard Architecture
    * CPU
      · Information Memory

        - Data Memory
    - This is not Harvard Memory

Delayed Jumps

- RISC-I cycle long enough to
    1. Read registers, perform ALU operation, and store result back in a register **OR**
    2. Read instruction from memory, **BUT** not both sequentially
- What about `jmp`, `call` and `ret`

fetch jmp		execute jmp
	fetch i_{next}	execute i_{next}

execute jmp calculates the address of the next instruction

fetch instruction needs to fetch next instruction, but don't know its address yet

- RISC-I solution is to use "delayed jmps*
- `jmp`/`call`/`ret` effectively take lace *after* the following instruction (in the code) is executed

```
1  sub R16, #1, R16{C}
2  jne L               ; Conditional jump
3  xor R0, R0, R16
4  sub R17, #1, R17

10  L: sll R16, 2, R16
```

- If conditional `jmp` taken
    - effective execution order 1, 3, 2, 10, …
- If **not** taken
    - effective execution order 1, 3, 2, 4
- **NB:** `jmp` condition evaluated at the *normal time* (condition codes set by instruction 1 in this case)

Example

- Consider the RISC-I code for the following code segment

```
i = 0;          // Assume i in r16
while(i < j)    // Assume j in r17
    i += f(i); // Assume parameter and result in r10
k = 0;          // Assume k in r18
```

- Unoptimised
- Place `nop` (`xor r0, r0, r0`) after eacg `jmp/call/ret` (in the delay slot)

```
    add R0, R0, R16     ; i=0
L0: sub R16, R17, R0{C} ; i < j
    jge L1
    xor R0, R0, R0      ; nop
    add R0, R16, R10    ; set up parameter in r10
    callr f
    xor R0, R0, R0      ; nop
    add R16, R16, R16   ; i += f(i)
    jmp L0
    xor R0, R0, R0      ; nop
L1: add R0, R0, R18     ; k=0
```

- Reorganised and optimised

```
    add R0, R0, R16     ; i=0
L0: sub R16, R17, R0{C} ; i < j
    jge L1
    add R0, R0, R18     ; k=0
    callr f
    add R0, R16, R10    ; set up parameter in r10
    jmp L0
    add R16, R16, R16   ; i += f(i)
L1:
```

- Managed to place useful instructions in the delay slot

fetch i1	execute i1		
	fetch jmp	compute addr	
[delay slot]		fetch i3	execture i3
			fetch i4

- Destination of jmp instruction is i4 (if jump not taken this will be the instruction after the delay slot)
- i3 executed in the delay slot
- Better to execute an instructions in the delay slot than leaving execution unit idle
- 60% of delay slots can be filled with useful instructions

Pipelining

- Key implementation technique for speeding up CPUs
- Break each instruction into a series of small steps and execute them in parallel (steps from different intructions)
- Clock rate set by the time needed for the longest step - ideally time for each step should be equal
- Consider a 5 stage instruction pipeline for the hypothetical DLX microprocessor (after Knuth's MIX)
  - IF - Instruction Fetch
  - ID - Instruction Decode and Register Fetch (operands)
  - EX - Execution and Effective Address Calculation
  - MA - Memory Access
  - WB - Write Back (into register)
- Execution time of each individual instruction remains the same but throughput increased by the depth of the pipeline (five in this case)
- Clock frequency five times faster than non-pipelined implementation
- Good performance if it runs without stalling
- For example, pipeline stalled whilst data is read from memory, if memory access causes a cache miss
- Instruction *issued* when it an be executed without stalling
- **Also** note that a non-pipelined DLX requires 2 memory access every 5 clock cycles *while* a pipelined DLX requires 2 memory accesses per clock cycle

Data Hazards

- Consider the execution of the following

```
R1=R2+R3     ; ADD
R4=R1-R5     ; SUB
```

- ADD instructions writes R1 in the WB state, but SUB reads R1 in the ID state

- Problem is solved in DLX by
  - Pipeline forwarding (or bypassing)
  - Two phase access to the register file
- Alternative approach is to expose the pipeline to programmers
- Programmers would need to insert three instructions between ADD and SUB to get the *expected* result (three pipeline instructions between ID and WB)