8. Instruction set architectures

Instruction and memory cycles

- The fetch and execution of a machine code instruction by a CPU is called an instruction cycle.
- The reading or writing of one word to one memory location (over the data bus) is a memory cycle.
- A memory cycle will take a fixed amount of time depending on the speed of the CPU and its me
  - In general a mem ternal operation like accessing a regist
- An instruction cycle may involve several
  - To fetch a machine code word takes a me
  - To read or write a data word takes a mem
- In Sigma16 an RRR instruction uses 1 memory cycle (to fetch the operation word).
  - How many memory cycles does a LOAD take?
  - How many memory cycles does a JUMP take?

Hardware Registers for Instructions

- The computer needs information about the current and next instruction
- This information is maintained automatically by hardware, via several special registers associated with the Control Unit. All CPUs have
  - The IR (Instruction Register) contains the operation word of the instruction that is being executed
  - The PC (Program instruction that will be executed.

- There are also machine specific contro in Sigma16:
  - The Address Register (ADR) contains the t ied in an RX or X instruction.
  - The Data Register (DAT) holds temporary data

- The IR, ADR and DAT are not part of the programmer’s model since they do not carry information between instruction cycles.
  - They are sometimes referred to as temporary registers.
  - However, they are exposed in the Sigma16 emulator to help with debugging.

Example: executing a JUMP

- When the Sigma16 instruction `JUMP x[R1]` is executed, the following events occur:
  - First word of instruction is fetched, PC increased by 1
    `IR:=Memory[PC]; PC:=PC+1`;
  - CPU discovers this is a JUMP, and fetches second word.
    `ADR:= Memory[PC]; PC:=PC+1`; (The result: `ADR:= x`)

  - Effective address
  - Execution of the

- The next instruction will be fetched fied PC value: `IR:= Memory[PC]`

- Notice how instruction execution can be described by means of assignment statements to CPU registers.
- Summary: a `JUMP` is really just an assignment to the PC register.

Instruction types

- Different CPU designs have different approaches to machine code design.
- Examine a machine code (assembly) instruction from any CPU with the following questions:
  1. How many memory cycles are needed to fetch the whole instruction?
  2. How many memory cycles are needed to fetch or store data?
  3. How many independent memory locations can the instruction access?

- Memory cycles are re tions and instructions requiring many mem ill take relatively longer.

- One could design a *Sigma16-like* machine on like
  - `ADD x[R1],y[R2],z[R3].`

- This is what is called a 3-address instructio
  - How long would such an instruction be?
  - How many memory cycles would it need to fetch or store data?
  - To perform this operation in Sigma16 how many instructions are needed?
  - Which would be faster?
  - Is this an argument against Sigma16?

RISC and CISC

- Why not have instructions of multiple types in a CPU?
  - Problem is that many complex instructions mean a long operation word and a complex control unit.
  - Complex control unit with many instruction types is slower than a simple one with few types.
  - Yet by the 1970s this was the way most CPUs were evolving.
- As a counter to this, in the 1980s to the idea of a reduced instruction set computer (RISC)

  - eliminate comple ions accessing memory.
  - All memory acces . Sigma16)

  - Sometimes called a LOAD-STORE archite
  - Use internal registers for all arithmetic and Make these as fast as possible.

  - Need lots of internal registers.
  - Simple fast control unit
- Conventional machines with complex instructions were called CISC.
- CISC vs RISC was a computing "holy war" in the 1980s and 1990s (comparable to Mac vs PC?)
- Outcome was a compromise. Most successful architecture, Intel's x86 (aka Pentium), started as CISC but was modified to include RISC ideas.

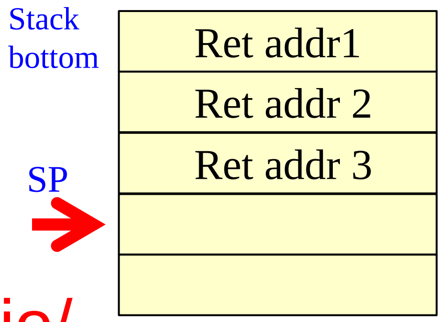
Virtual Machines

- A virtual machine (VM) is a model of a machine that does not really exist
  - Implemented in software rather than hardware.
  - Implementation (guest) is called an emulator and can be run on chosen real hardware (host).
  - Examples: Sigma16, Microsoft Hyper V, Java.
- Raw machine code all
  - Very dangerous
  - A good HLL will try to spot accidental m ammer and an operating system will block an application program thing that looks dangerous. But maliciously written code vent such safety measures.
  - On a system like Sigma 16 programming is in machine code directly and there is no OS protection. Fortunately Sigma 16 is a VM and typically a VM can confine disasters to itself. The VM may crash but the host is OK.
  - At cost of performance a VM can perform checks and generate reports on operations before they are performed (good for debugging).

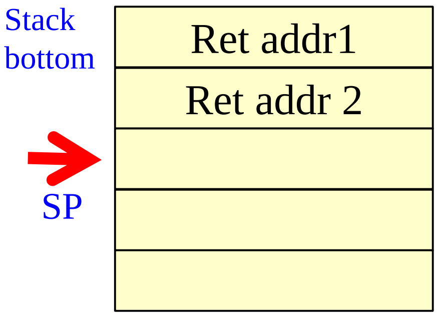
Subroutines and stacks

- A JAL instruction saves return address for subroutine call in an internal reg.
- But one subroutine can call another and so on (nested calls). When there are many calls, the register file will not have enough space for all return addresses.
- Instead return addresses are usually stored in data memory in a data structure called a stack
  - Stack is setup and maintained by the programmer
  - Unlike an array a stack can grow and shrink.
  - First address is called the st
  - In Sigma 16 the stack botto ent. It must be placed after all other DATA statements as the stack ca overwrite anything above it.
  - The stack top is tracked by a register, the stack pointer (S current address. In Sigma 16 the programmer uses one of the R registers to do this.
  - Initially the stack top and bottom are the same so SP is set to the address of stack bottom. At this point the stack is empty.
  - After a subroutine call, programmer stores return address on stack and increments SP.
  - After a return the address at the stack top is retrieved and the SP is decremented

Stack after 3 nested subroutine calls…



And after first subroutine returns…



More on Stacks

- A stack is a last in first out (LIFO) data structure.
- Storing an item at the stack top is called a PUSH operation (e.g. save return address)
- Retrieving an item from the stack top is called a POP or PULL operation (e.g. retrieve return address)
- The item popped is always the last one pushed.
- Stack can be used to push and pop items other than return addresses but must be done with care by any program d

- In Sigma 16 the stack upwards.

- Many CPUs help maintain the stack by provid
  - A dedicated register for the SP which auto-incr when subroutine calls are made
  - Special PUSH and POP instructions for manua ving data items on the stack

- Note that in many CPU designs the stack grows downwards in memory so the stack top is at a lower address than the stack bottom.
- Programmer is responsible for making sure maximum stack size does not overwrite other data or code or require unpopulated addresses. Failure is stack overflow.