ISA Details

prepared for Programmable Logic Lessons / 1.5 by ~skmp

Kindly hosted by hackerspace.gr

What is defined in an ISA ?

- Register count, types, and addressing mores
- Operations, and how they get their data
- Operation encodings
- “user visible” cpu states (this doesn't include internals like bus states)
- Memory mapping (if any) and coprocessors that come with the ISA
  - Some ISAs can be extended via external coprocs

Traditional ISA Paradigms

- CISC
  - Complicated encoding, multiple encodings for each opcode, flexible addressing modes
    - Eg add eax,[ebx+edx*4+0x88712]; // compute address, read data and add it to eax
- RISC
  - Simpler encoding, each opcode does one “thing”
  - Typical traits are dedicated load/store and many registers
- VLIW
  - Can specify the instruction for multiple units at once
    - Lower level than both CISC and RISC
- Modern cpus implement a blend of all 3 :)

Registers

- There are many types of registers
  - Integer registers (~ r0.. rN, ...)
  - Floating registers (~ f0.. fN, fr0 frN, …)
  - Program counter, stack pointer, and others
    - pc/eip, sp, lr, etc
  - State registers
    - Current processor mode, etc
- They are the fastest-access memory that a cpu has

More on registers

- Program counter
  - Next address of the opcode to execute
  - Tracks “where” we are in the program
- Stack pointer
  - Stack is used as temporal local storage
- “General” registers
  - Used to perform ALU operations
  - In some cases, PC and SP are part of the general register file
- Link register
  - Used mostly in RISC, stores the return address

Instruction Types

- ALU/Computing
  - Add, sub, and, or, etc
  - Compares
- Branching
  - Changes the program execution order (by changing the pc)
  - Unconditional
  - Conditional (jump if equal and such) – thats how ifs are implemented !
- I/O
  - Memory reads & writes
  - Peripherals are typically connected to the memory bus, so you communicate with them using the memory interface

Data addressing

- There are many ways to read and write opcode results
  - Implicit data
    - pre-decided while designing the cpu. Eg, a branch opcode writes to the pc register
  - Immediate data
    - directly encoded on the opcode
  - Register
    - There is a register index encoded on the opcode
  - Memory
    - Intimidate - the address is encoded on the opcode
    - Via Register – the address comes from a register
  - Register
- Most ISAs provide immediate and register
  - Dedicated opcodes for memory access
    - X86 is an exception to that (CISC traits)

Architectural state

- … The entire state of the architecture
  - Consists of all the data that defines the current state
- Typically contains all registers
  - And then some more internal state
- Memory is not considered as part of the architectual state

General trends

- As cpus get more and more complex, and transistors get cheaper and cheaper, the cost metrics change and as a result the architecture implementation changes drastically
  - Most cpus have a “frontend” that translates the ISA to the internal state and is responsible to maintain coherency
  - Hugely different implementations exist for the same ISA
    - 8086 – typical simple mostly monolithic CISC ~ 30K transistors
    - Core i7 – Out of order, super scalar, register renaming,caching, VLIW/MIPS internal state, and MUCH MUCH more ~ 2,270,000K transistors (2.2 Bilion)
    - Both implement x86-16
- There is a trend to make things more 'generic' with fewer dedicated units
- Also, while opcodes become far more complex they usually implement only a single operation (so they can be combined in more elegant ways) with all addressing modes
  - Also known as orthogonal design
- Yet we still have transistors to spend, so we add more cores and more parallelism

Designing an ISA

- Mostly a trade-off of complexity and usability
  - There's a trend to move some of the complexity to compilers
- Decide on word size
- Decide on register count
  - X86 → 8, arm → 13, mips → 31, ppc → 31.5
- Decide addressing modes
- Decide on the instructions
  - Add, sub, neg, or, xor, etc.
    - Not all of them are required – eg, a-x = a+(-x)
- Come up with a compact format for the opcodes
  - Random googled example, MIPS32

Designing an ISA (2)

- Decide on a simple syntax for the opcodes
  - The assembly language of the cpu
- Decide how to handle illegal/unexpected states
  - (aka: Exceptions)
- Decide how to handle “notifications” from the hardware
  - (aka: Interrupts)
- Decide what peripherals should be built in
  - Usually: Timers, MMU, etc

Thanks !

Let's improve the design we worked on last time now !

Next week we'll talk more about integrating more peripherals and SoCs

Feel free to drop by #hsgr @ freenode

... or the hsgr mailing list

... and use the wiki !

(or, send direct feedback – skmp@emudev.org)