Instruction Set Architectures (ISAs)

Simon Hollis, COMS12200

What's in an ISA?

- An Instruction Set Architecture (ISA) is a definition of the instructions supported by a particular processor.
- The ISA specifies the exact behaviour of instructions fed to it.
- The ISA should be designed sympathetically for the programming language that will write code for it.

The ‘I’

- What exactly is an ‘instruction’?
  - It’s a definition of a unique operation code PLUS some contextual information.
  - We say there is an operation plus zero or more arguments.

The ‘I’

- We’ve seen this before with `1 01` being *decodable*.
  - `1` is the operation code *(op-code)*
    - *Here 1 means ‘move’*
  - `01` is the argument. Here, it is the literal/constant/immediate value 1

Decoding

- All instructions must be *uniquely decodable*.
  - One-to-one mapping of bit-string to meaning.
- Some ISAs do not define all possible bit-strings.
  - This is allowed, but can cause problems if an undefined instruction is encountered.

Creating instructions

- So, to create an ISA we need to produce a list of instructions and their possible arguments.
- What instructions make it to the set are based on experience and need.

Instruction classes

- Most instructions fall naturally into one of a few classes:
  - Arithmetic
    - *e.g. Add, subtract, multiply.*
  - Comparison
    - *e.g. Compare, If, set flags*
  - Memory
    - *e.g. Loads, stores, swaps.*
  - Control flow
    - *e.g. Branches, state changes*

Instruction classes

- Simple processors need only the previous classes, but complex, high-performance ones may employ:
  - Vector processing
    - *e.g. Array manipulation, block operations.*
  - Digital Signal Processing (DSP)
    - *e.g. Fast Fourier Transform (FFT)*

Instruction selection

- How do you go about selecting what instructions should be part of an ISA?
- #1: Think about the tasks in hand
  - Common operations
  - Representative applications
  - Future-proofing?

ISAs

ISA CATEGORISATION: RISC AND CISC

RISC and CISC

- We often categorise ISAs by how close they are to one of two opposing paradigms.
- RISC (Reduced Instruction Set Computer)
- CISC (Complex Instruction Set Computer)

RISC

- RISC (Reduced Instruction Set Computer)
  - The aim is to produce a minimal set of instructions that are ***lightweight*** and ***fast*** to execute.
  - Each instruction should have a ***similar run time***.
  - Complex operations must be **synthesised** from simple ones.
  - RISC machines run fast but may need lots of instructions to do a complex job.

CISC

- CISC machines are the opposite:
  - If there is a specialised operation to be done, add a ***dedicated instruction*** for it (up to a limit!)
  - Rich instruction set means ***smaller programs***.
  - Mixed-length instructions means ***unpredictability*** and compiler headache.
  - Processor ends up ***complex*** and ***slow***.

ISA OPTIONS

Instruction choices

- For many operations, you may have a set of choices
  - e.g. `r0 <- r0 + r1`
  - How many ways can we do this?

`r0 <- r0 + r1`

- Can have multiple accumulators: `ACC1, ACC2`

1. `ACC1 <- r0`
2. `ACC2 <- r1`
3. `ACC1 <- ACC1 + ACC2`
4. `r0 <- ACC1`

- This takes 4, very simple instructions.

`r0 <- r0 + r1`

- or, using a stack
  - `PUSH [r0]`
  - `PUSH [r1]`
  - `ADD`
  - `POP [r0]`
- (r0 here assumed to be a memory address here)

r0 <- r0 + r1

- We can get it down to 3 instructions
  - `ACC <- r0    :   MOV ACC, r0`
  - `ACC <- ACC + r1   :   ADD r1`
  - `r0 <- ACC   : MOV r0, ACC`

`r0 <- r0 + r1`

- ...or in 2 instructions
  - `MOV ACC, r2`
  - `ADD r0, r0, ACC`

`r0 <- r0 + r1`

- We can do better
- Set r0 to always be the destination.
  - r0 <- r0 + r1 : (ADD r0, r1)
- Or, even better -- r0 also always the argument:
  - r0 <- r0 + r1 : (ADD r1)

Instruction set choice

- So, how do I choose between all the possibilities?
- It’s a design choice, taking into account all the tradeoffs

ISA tradeoffs

- Tradeoff 1: more instructions -> more bits
    - More instructions -> greater information
    - But: more space, more energy
- Tradeoff 2: more operands -> more bits
    - More bits -> more work
    - More space, more energy

ISA implications

- Choices of instruction specificity impact how a machine may be programmed.
- Can imply a choice of paradigm:
  - Implicit source & destination: stack machine
  - Implicit destination: accumulator machine
  - Explicit source and destinations: register machine

Picking a set

- We want to cover the useful space of execution needs.
  - ***No repetition.***
  - ***No coverage where it is not needed.***
  - ***Minimum size*** to do the job.
  - Check if instruction addition is harmful.
    - Recall Amdahl’s Law

Amdahl’s Law

- Amdahl’s Law is a measure of utility of adding an instruction to an ISA.
- What it says is don’t add an instruction if it slows down an application overall.

Amdahl's Law

If a new instruction is added to speed up an operation by **s** times, the law gives the maximum overall speedup:

speedup = s / 1 + (f * (s - 1))

where **0 <= f <= 1** is the fraction of time spent *NOT* running the operation in question in the original design

ISAs

ENCODING INSTRUCTIONS

Instruction encodings

- Once we have selected our set of instructions, we need to encode them ***unambiguously***.
- A common way is to separate them into code ***prefixes*** and ***operand*** areas.

Instruction encodings

- ***Space*** is a major factor in deciding instruction encodings.
    - Space -> memory size
    - Space -> no of instructions loadable/second
    - Space -> decode stage complexity

Instruction lengths

- Instruction lengths can, therefore vary
  - Some instructions need to provide more information than others -> longer instruction
  - If instructions can be shorter, that's good.
- All this implies that ***fixed length*** instructions may be sub-optimal.

Instruction lengths

- Consider these two instructions:
- `ADD r0, r0, r1`
  - Op-code + 3 operands
  - 16 registers -> 12 bits + op-code
- `HALT`
  - Only an op-code.
- ***Variable length*** instructions can save space.

Variable or fixed length?

- Variable length instructions:
  - Size efficient
  - Can be extended to support more operands
  - Optimal in some sense.
- BUT: almost infinite amount of possibilities causes decoding to be very complex
  - -> pipeline slowdown

Fixed length instructions

- May be sub-optimal in length
- Can get around this by allocating op-codes as part of a ***prefix*** code.
  - e.g. `100` - *Branch*
  - `110` - *Branch equal to*
  - `111` - *Branch not equal to*
- Major advantage in simplified decode stage.
- Tends to be a winner in most modern designs.

Conclusions

- Creating an ISA is a complicated task
- There are almost infinite ways to do it
- It’s a very creative thing
- A choice of ISA has big implications for users of a processor. BUT:
    - History shows that bad ISAs can flourish (e.g. x86)
    - Good ISAs can become bad over time (or vice-versa (e.g. VAX, MIPS)