Computerarchitectuur

H&P Appendix A: Instruction Set Principles

Kristian Rietveld

http://ca.liacs.nl/

Universiteit Leiden
The Netherlands

Instruction Sets

- Een processor moet precies worden verteld wat deze moet doen.
- Dit staat “opgeschreven” in een programma.
- Een programma bestaat uit instructies.

- Hoe zien die instructies eruit?
- Op wat voor manieren kunnen we instructies vormgeven?

Algemeen computermodel

- CPU voert berekeningen uit mbv ALU.
- CPU heeft interne opslag, vaak registers.
- CPU staat in verbinding met grootschalig geheugen: RAM.
- Alles gedreven door instructies.

Instruction Set Architecture

- Kortweg: ISA.
- Dit is hoe de computer zich openbaart aan de buitenwereld.
  - De “interface” te gebruiken door programmeur en compiler writer.
- Bestaat in het algemeen uit:
  - Een zeer precieze beschrijving hoe de instructies moeten worden opgeslagen/geëncodeerd.
  - Semantiek: wat voor instructies zijn er en wat voor effect heeft het uitvoeren van die instructie?
- Omschreven in de ISA manual of architecture reference manual van de processor.

Evolutie van ISAs

- *Jaren 70*: kosten software ontwikkeling terugdringen.
  - High-level architecturen om het leven van software ontwikkelaars makkelijker te maken.
  - Leidde tot architecturen met zeer uitgebreide instructiesets, bijv. VAX.
- *Jaren 80*: terug naar simpele architecturen.
  - Compilertechnologie werd steeds beter.
  - Nadruk op verbeteren processor performance.
- *Jaren 90*:
  - Verdubbeling adresgrootte.
  - Verdere verbetering performance.
  - Multimedia extensies.

CISC vs. RISC

- **CISC**: Complex Instruction Set Computing
  - Complexe instructies die eigenlijk meerdere dingen tegelijk doen: zowel een waarde uit het geheugen halen als een optelling uitvoeren.
  - VAX, PDP-11, Intel x86, ...
- **RISC**: Reduced Instruction Set Computing
  - Eenvoudige instructies, kleine instructieset.
  - Betere performance door lagere CPI.
  - MIPS, SPARC, ARM, PowerPC, Alpha, ...
- Verschillen zullen gaandeweg duidelijker worden.

Hoe zit dat met x86?

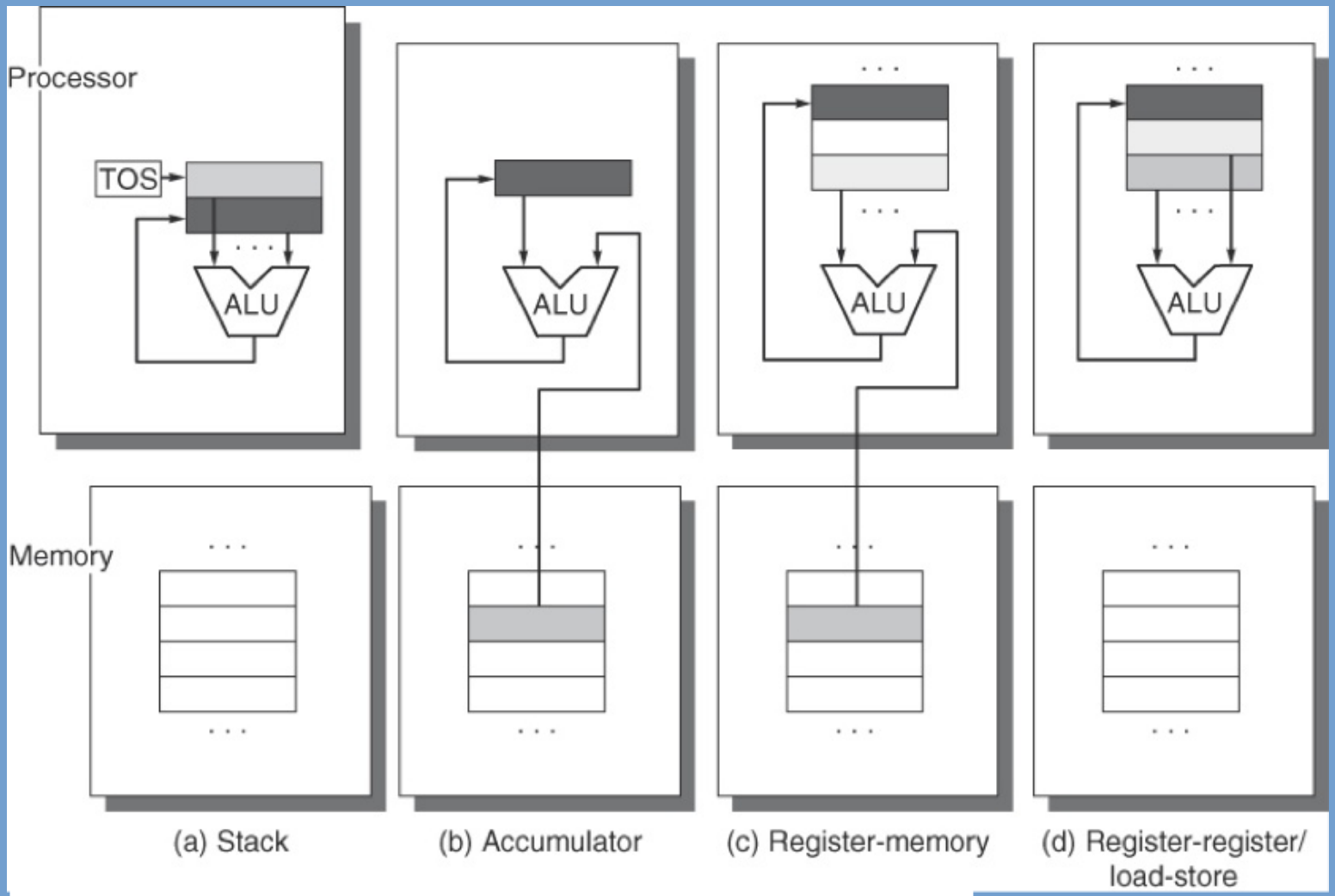
- We zagen x86 net geclassificeerd als CISC.
- Schijn bedriegt …

- x86 is CISC aan de buitenkant.
  - Backwards compatibility.
- Aan de binnenkant zit eigenlijk een RISC processor.
  - In de hardware worden alle CISC instructies omgezet in RISC instructies.

Verschillende architecturen

- Er bestaan verschillende manieren om data intern in een processor op te slaan.
- Dit leidt tot verschillende architecturen:
  - Stack architecture
    - Operanden impliciet (top of stack).
  - Accumulator architecture
    - Één van de operanden wordt impliciet de accumulator.
  - General-purpose register architecture
    - Operanden moeten expliciet worden gespecificeerd.

Verschillende architecturen (2)



Taken from Computer Architecture: A Quantitative Approach, fifth edition. Fig. A.1.

Register computers

- Als we naar de register computers kijken, zien we twee soorten:
  - *Register-memory:* elke instructie kan ook naar geheugen refereren.
    - x86 instructies kunnen dit bijvoorbeeld.
  - *Load-store*: geheugen kan alleen worden benaderd met expliciete load en store instructies.
    - RISC architecturen normaliter load-store.

- Je kunt een derde klasse bedenken, memory-memory, maar dit soort machines worden niet meer gemaakt.
  - Waarom niet? (Denk aan de trends van vorige week)

Load-store architectures

- De standaard tegenwoordig: load-store register architectures.
- General-purpose registers.
  - Vrijheid voor de compiler.
- Compiler probeert variabelen tijdelijk op te slaan in registers om geheugenverkeer te minderen en zo het programma te versnellen.

Voorbeelden

- C = A + B voor verschillende architecture classes:

Stack

```
Push A
Push B
Add
Pop C
```

Accumulator

```
Load A
Add B
Store C
```

Register-Memory

```
Load R1, A
Add R3, R1, B
Store R3, C
```

Register (load-store)

```
Load R1, A
Load R2, B
Add R3, R1, R2
Store R3, C
```

See also Computer Architecture: A Quantitative Approach, fifth edition.Fig. A.2..

Voorbeelden

- C = A + B voor verschillende architec

Stack

```
Push A
Push B
Add
Pop C
```

Accumulato.

```
Load A
Add B
Store C
```

Register-Memory

```
Load R1, A
Add R3, R1, B
Store R3, C
```

Register (load-store)

```
Load R1, A
Load R2, B
Add R3, R1, R2
Store R3, C
```

3-operand instruction

```
Load R1, A
Load R2, B
Add R2, R1
Store R2, C
```

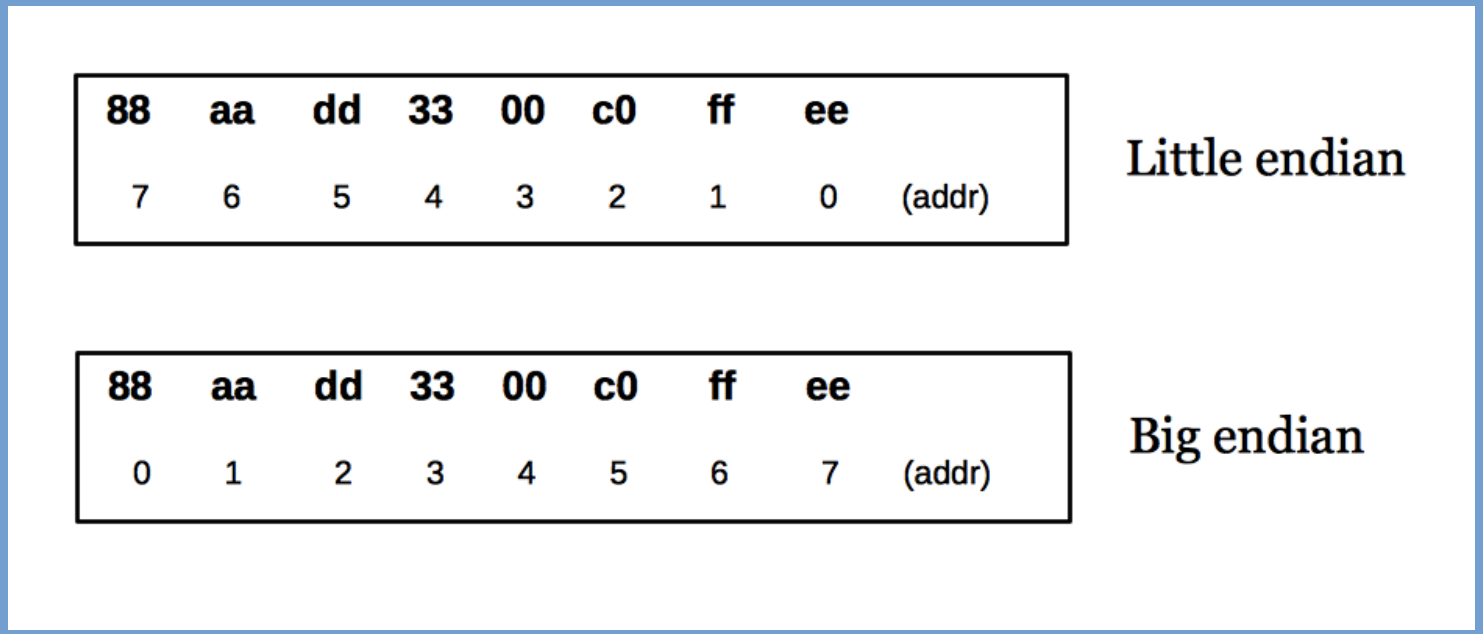
2-operand instruction

Memory Addressing

- Als we een geheugenadres opgeven, wat betekent dit?
- Meestal geven we een geheugenadres op en de grootte van een object om te benaderen.
  - byte (8 bits), half word (16 bit), word (32 bit), double word (64 bit).
- Er bestaan verschillende “byte orders”:
  - *Litte endian*: eerste byte op de least significant position (the little end).
  - *Big endian*: eerste byte op de most significant position (the big end).
- Meestal geen probleem, totdat je data gaat uitwisselen tussen verschillende computers.

Byte ordering

- We slaan het getal *0x88aadd3300c0ffee* op.



Alignment

- Veel architecturen vereisen dat toegang tot objecten groter dan een byte “aligned” is.
- Een object s bytes groot opgeslagen op adres A is aligned als $A \bmod s = 0$.
- Anders gezegd: een object moet worden opgeslagen op een meervoud van de grootte.

- Waarom is dit? Geheugen vaak aligned op meervoud van word of double word.

Alignment (2)

- Wat gebeurt er in het geval van een misalignment?
  - Sommige architecturen ondersteunen dit helemaal niet en dit resulteert dan in een exception: Bus error.
  - Andere architecturen ondersteunen dit wel, maar een misaligned access is duurder dan een aligned.
- Misaligned access is op te lossen door meerdere aligned accesses te doen en hier de benodigde data uit te extraheren.

Addressing Modes

- Hoe duiden we nu aan op welke operanden we een operatie willen uitvoeren?
- Hier bestaan verschillende manieren voor – verschillende *addressing modes*.

- We zullen nu een aantal van deze modes bekijken.
  - (In het boek (Fig. A.6.) staan nog een aantal meer, exotische, modes).
- Verschillende architecturen implementeren verschillende subsets van deze modes.

Addressing Modes (2)

Mode	Example Instruction	Meaning	Use case
Register	`Add R4, R3`	`Regs[R4] <- Regs[R4] + Regs[R3]`	When values are in registers
Immediate	`Add R4, #3`	`Regs[R4] <- Regs[R4] + 3`	For constants
Direct or absolute	`Add R1,(1001)`	`Regs[R1] <- Regs[R1] + Mem[1001]`	E.g. accessing static data, address constant may be large.
Register indirect	`Add R4,(R1)`	`Regs[R4] <- Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or computed address.
Displacement	`Add R4,100(R1)`	`Regs[R4] <- Regs[R4] + Mem[100 + Regs[R1]]`	Accessing local variables.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] <- Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]`	Used to index arrays.

Addressing Modes (3)

- Met slimme addressing modes kan het aantal instructies worden verlaagd.
  - Nadeel: complexiteit wordt groter, dus risico dat gemiddelde CPI omhoog gaat.
- Voor displacements en literals moet er ruimte worden gereserveerd in de instructies. Grootte zit dus aan een maximum gebonden.

Operaties

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide.
Data transfer	Loads and stores (or move on register-memory)
Control flow	Branch, jump, function call
System	System calls (OS), virtual memory management
Floating point	Floating-point operations
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression

Taken from Computer Architecture: A Quantitative Approach, fifth edition. Fig. A.12.

Operaties (2)

- Welke operaties snel maken? “Common case”.
- Bijv. de top 10 instructies die volgen uit het draaien van de SPEC benchmark op x86:

Instruction	(% total executed)
load	22%
conditional branch	20%
compare	16%
store	12%
add	8%
and	6%
sub	5%
move register-register	4%
call	1%
return	1%
TOTAL	**96%**

Taken from Computer Architecture: A Quantitative Approach, fifth edition. Fig. A.13.

Control flow

- Nodig voor het implementeren van if, for, while, switch, function calls, etc ...

- *jump*: unconditional change in control flow
  - e.g. function call, return from function
- *branch*: conditional change in control flow
  - e.g. if, switch, break out of for loop.

Control flow (2)

- Waar springen we naartoe? Dit adres moet in de instructie worden vastgelegd.
  - Uitzondering: return from function call. Adres bij compileren niet bekend, wordt tijdens runtime bepaald.
  - Geldt bijvoorbeeld ook voor virtual functions, function pointers.
  - Oplossing: adres lezen uit register of stack.

- Vaak wordt gekozen om een *displacement* op te tellen bij de huidige waarde van de program counter.
  - PC-relative branches en jumps.
  - Het doel is vaak dichtbij, dus maar weinig bits nodig!
  - Deze code is “position independent”.

Control flow (3)

- Drie manieren om branch conditie te specificeren.
- *Condition code*
  - ALU operaties (bijv. test instructie) zetten speciale condition flags/bits.
  - Branching gebeurt op basis van deze bits (is bit wel of niet gezet?)
- *Condition register*
  - Resultaat vergelijking wordt opgeslagen in register. Dit register wordt bekeken door branch instructie.
- *Compare and branch*
  - Vergelijking en branch in 1 instructie.

Encoding instruction sets

- Instructies moeten worden geëncodeerd in een binaire vorm.
- Vaak gebeurt dit op basis van *fields* met vaste grootte in bits. Een field voor source register, opcode enz.
- *opcode*: getal dat de uit te voeren operatie aanduidt.

- Vorm van encoderen belangrijk:
  - heeft invloed op de grootte van gecompileerde programma's,
  - heeft ook invloed op de implementatie van de processor, je wilt zo snel mogelijk instructies kunnen decoderen.

Encoding instruction sets (2)

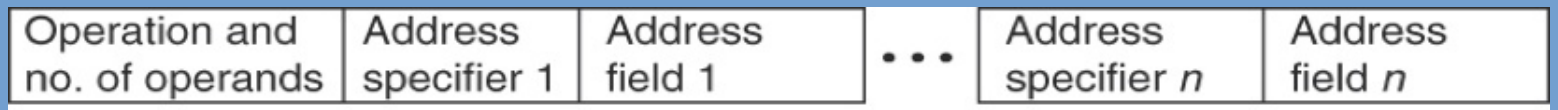
- Het belangrijkste om te bepalen is hoe de operanden en verschillende addressing modes worden vastgelegd.
- Hangt helemaal af van de architectuur!
  - Sommige oude architecturen: 1 tot 5 operanden, 10 mogelijke addressing modes voor elke operand.
    - Aparte address specifier nodig voor elke operand.
  - Load-store computers: load instructie heeft altijd 1 memory operand. Er worden maar 1 of 2 addressing modes ondersteund.
    - Addressing mode kan worden opgeslagen in opcode door beperkte aantal mogelijkheden.

Encoding instruction sets (3)

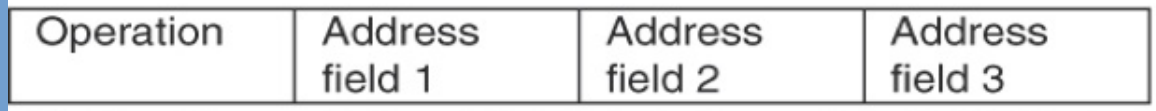
- Balancing act:
  - Graag zo veel mogelijk registers en addressing modes.
  - Meer registers -> meer bits nodig in instructie.
  - Gemiddelde grootte instructie groeit en daarmee ook gemiddelde grootte van een programma.

  - Instructies met vaste grootte is makkelijker implementeren (en belangrijk voor pipelining), maar laat weer de gemiddelde grootte toenemen.

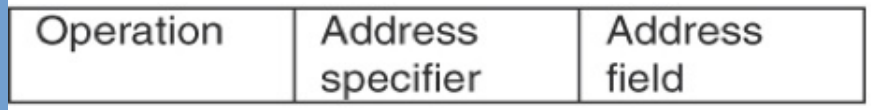
Encoding instruction sets (4)

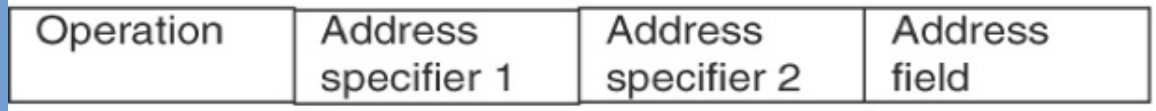


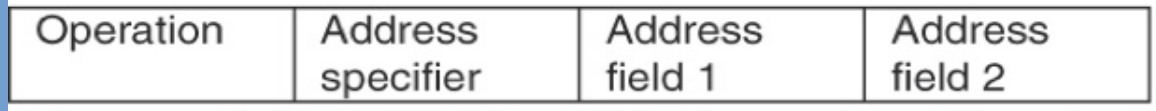
(a) Variable (e.g., Intel 80x86, VAX)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)







(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

Taken from Computer Architecture: A Quantitative Approach, fifth edition. Fig. A.18.

Encoding instruction sets (5)

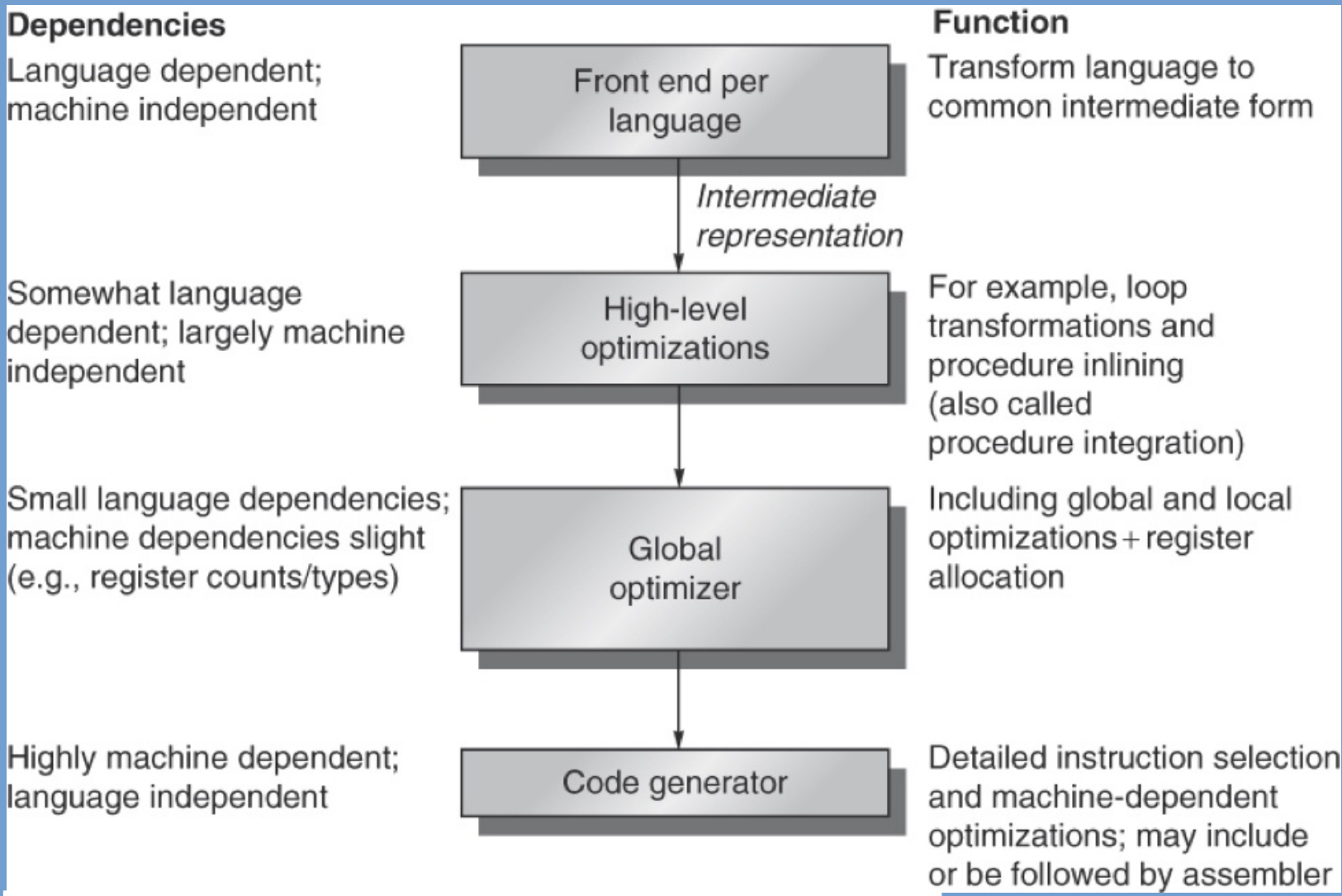
- RISC architecturen zoals MIPS, ARM hebben 32-bit instructies.
- Voor embedded toepassingen telt de grootte van de code en 32-bit instructies zijn een probleem.
  - Fabrikanten hebben hierdoor ook een 16-bit code geïntroduceerd: ARM Thumb, MIPS MIPS16.
  - Er wordt een code size reductie geclaimd van 40%.

Rol van de compiler

- Vroeger werden er nog wel eens instructies toegevoegd om het schrijven van assembly code makkelijker te maken.
- Tegenwoordig is dit niet meer aan de orde, de meeste code wordt gegenereerd door een compiler.

- Het is van belang rekening te houden met compilers bij ontwerp van instructie sets:
  - zorg voor regulariteit,
  - maak trade-offs zo eenvoudig mogelijk.

Rol van de compiler (2)



Taken from Computer Architecture: A Quantitative Approach, fifth edition. Fig. A.19.

MIPS architectuur

- We bespreken kort de ISA van de MIPS architectuur
- RISC, load-store architectuur.

- Kan zowel als Big en Little Endian opereren.
- 32 general-purpose registers: R0, R1, ..., R31.
  - R0 is altijd 0.
  - In assembly language schrijven we $0, $1, ..., $31.
  - Er zijn ook floating-point registers.

MIPS architectuur (2)

- Twee addressing modes:
  - immediate, 16-bit veld.
  - displacement, 16-bit veld.
- Nog twee addressing modes worden als volgt bereikt:
  - register indirect: plaats een 0 in het displacement veld.
  - absolute: gebruik R0 als base register.

- Instructies altijd 32-bits, er bestaan 3 instructieformaten of types.

Instruction formats

I-type instruction

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd=0, rs=destination, immediate=0)

R-type instruction

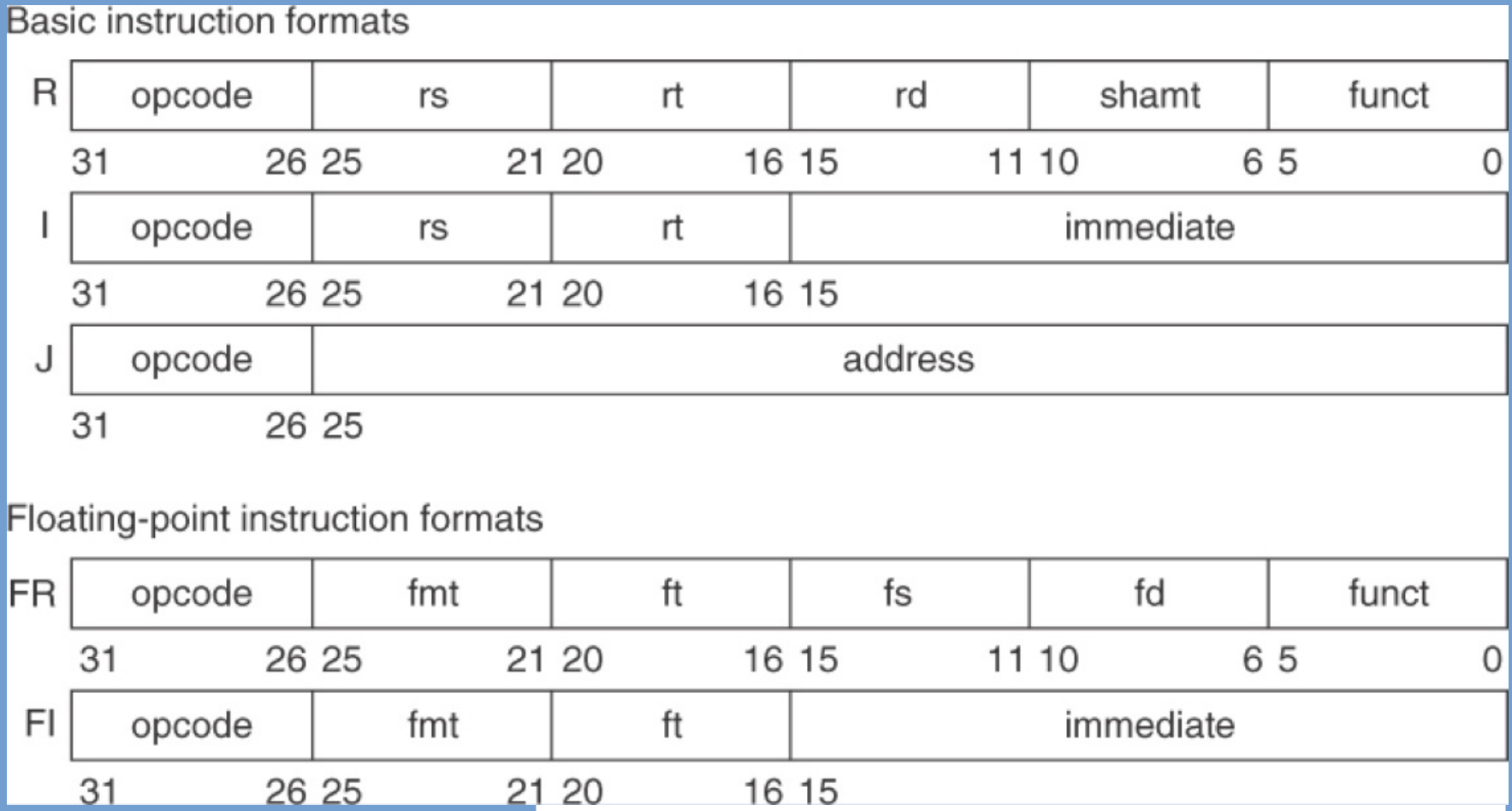
Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction

Jump and jump and link
Trap and return from exception

Taken from Computer Architecture: A Quantitative Approach, fifth edition.Fig. A.22.

Instruction formats (2)



Taken from Computer Architecture: A Quantitative Approach, fifth edition.Fig. 1.6.

Voorbeelden

```
LD R1, 30(R2)          # Regs[R1] <- Mem[30+Regs[R2]]

ADDU R1, R2, R3        # Add unsigned
                       # Regs[R1] <- Regs[R2] + Regs[R3]

ADDIU R1, R2, #3       # Add immediate unsigned
                       # Regs[R1] <- Regs[R2] + 3

SLT R1, R2, R3         # Set less than
           # Regs[R1] <- Regs[R2] < Regs[R3] ? 1 : 0
```

Voorbeeld decodering

- Instructie: 0x27bd0018
  - Bekijk opcode (bovenste 6-bits): 0x9 => addiu
  - addiu is een I-format instructie:

opcode	**rs**	**rt**	**imm**
31 … 26	25 … 21	20 … 16	15 … 0

  - rs: 0x1d => $29 => $sp
  - rt: 0x1d => $29 => $sp
  - IMM: 0x18 => 24

  - Dus: `addiu sp,sp,24`

```
        .rdata
        .align 2
$LC0:
        .ascii   "hello\000"
        .text
        .align   2
        .globl   main
        .ent     main
        .type    main, @function
main:
        .frame   $fp,24,$31
        addiu    $sp,$sp,-24       # Verlaag stack pointer met 24 bytes
        sw       $31,20($sp)       # Sla register $31 op (return addr).
        sw       $fp,16($sp)       # Sla frame pointer op.
        move     $fp,$sp           # $fp = $sp
        lui      $2,%hi($LC0)      # Zet hoge bits adres $LC0
                                   # in $2.
        addiu    $4,$2,%lo($LC0)   # Tel $2 en lage bits op.
                                   # $4 bevat nu adres $LC0.
                                   # $4 heet ook wel $a0.
        jal      puts              # Jump-and-link puts().
        nop                        # Delay slot.


        move     $2,$0             # Zet 0 in $2 (return value).
        move     $sp,$fp           # $sp = $fp
        lw       $31,20($sp)       # Restore $31
        lw       $fp,16($sp)       # Restore $fp
        addiu    $sp,$sp,24        # Restore $sp
        j        $31               # Jump $31 (return)
        nop
```