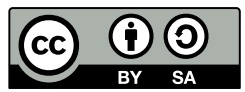
RISC architectures

Saulius Gražulis

Vilnius, 2020

Vilnius University, Faculty of Mathematics and Informatics
Institute of Informatics



x86

Advantages:

- Compatibility over 40 years
- Fastest possible simultaneous execution of all code, both old an new
- Fastest systems

x86

Drawbacks:

- Large, complicated chips
- Large transistor count – large power dissipation
- Some obsolete features present just for compatibility
- Some functions are duplicated
- A lot of features not used in any given application – ballast
- A lot of specialised with special behaviour instructions – difficulties for compiler writers

"AMD's 80x86 architect, Mike Johnson, famously quipped, "The x86 really isn't all that complex—it just doesn't make a lot of sense"

(Waterman 2016)

The case for RISC

- “Semiconductor memories are both fast and relatively inexpensive” (Patterson et al. 2003)
- “it is difficult to have ”rational” implementations” (ibid.)
  - IBM 370: Peuto and Shustek have discovered that a sequence of load instructions is faster than a load multiple instruction for fewer than 4 registers; this covers 40% of cases in typical programs (Patterson et al. 2003; Peuto et al. 1977)
  - Patterson found that for VAX 11/780 replacing complex INDEX instruction (calculates address of an array element, checks bounds) by several simple instructions (COMPARE, JUMP LESS UNSIGNED, ADD, MUL) can calculate the same function **45%** faster (in special cases, **60%** faster!) (Patterson et al. 2003)

The case for RISC

- “measurements of a particular IBM 360 compiler found that 10 instructions accounted for 80% of all instructions executed, 16 for 90%, 21 for 95%, and 30 for 99%”[1] (Patterson et al. 2003; Alexander et al. 1975).
- “pushing a register on the stack with **PUSHL R0** is slower than pushing it with the move instruction **MOVL R0,-(SP)** on the VAX 11/78” (Patterson et al. 2003)

[1] IBM 360 had 8 bit opcodes, so would have no more than 256 different instructions – with various operands (“IBM System/360 Principles of Operation”, p. 14)

Technical features:

	RISC	CISC
Instructions	Simple: Load/Store regs, operations **only** in regs	Complex tasks for multiple data types, both in regs and in memory
Instr. formats	Fixed length, two main types: load/store & `R := R op R`	Variable length, may types: load/store, `R := R op R`, `R := R op Mem`, `R := Mem op Mem`
Registers	16–32 general purpose	Specialised or 8–16 general purpose

Adapted from: (Jamil 1995)

RISC

Design decisions:

	RISC	CISC
Design objective	Trade off program length, minimise time to execute instruction	Minimise program length, maximise work/instruction
Implementation	Hard-wired	Microprogrammed
Caching	Essential (at least for code)	Useful ($\Rightarrow$ nowadays, essential)
Compiler design	Find best instruction ordering & register allocation	Find best/right instructions
Philosophy	Move all (complicated) functions to software	Move any useful software function into hardware

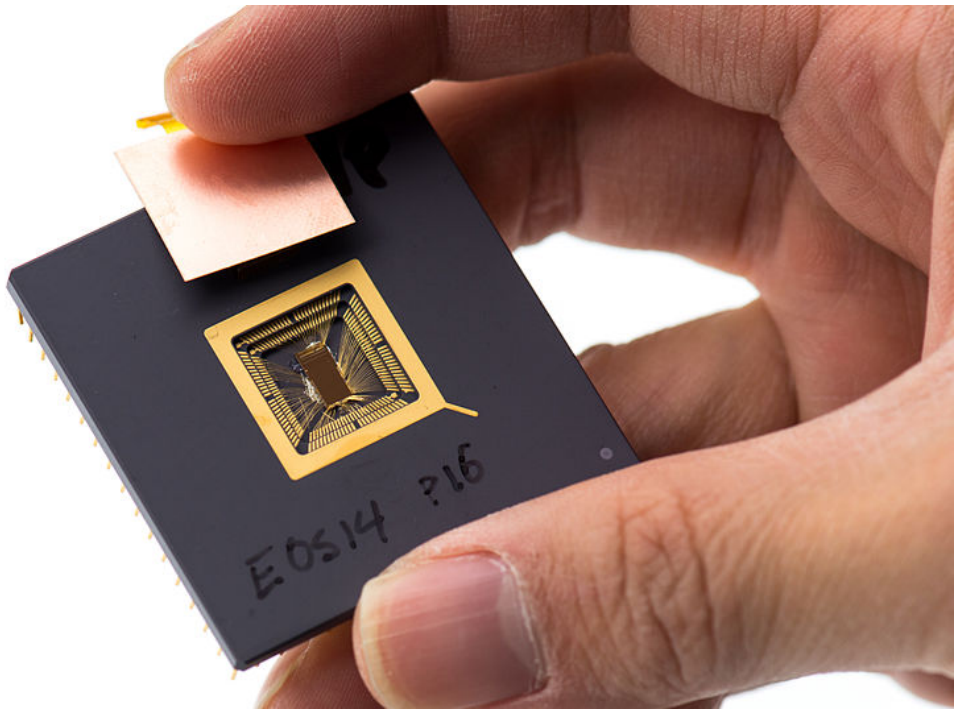
Adapted from: (Jamil 1995)

Some RISC machines

Past...

Some RISC machines

And possibly future...



RISC-V

- Open-standard RISC ISA :)
- Is provided under open source licenses
- Does not require royalty fees to use
- A number of companies are offering or have announced RISC-V hardware
- open source operating systems and tool-chains (compilers, assemblers) with RISC-V support are available

RISC-V variants

- 3 address widths:
  - RV32
  - RV64
  - RV128
- Multiple extensions:
  - I – base integer ISA
  - M – hardware multiply and divide
  - A – atomic synchronisation support
  - F – single precision floating point
  - D – double precision floating point

RV32G = RV32IMAFD

RISC-V variants

- 3 address widths:
  - RV32
  - RV64
  - RV128
- More exotic extensions
  - S – Supervisor mode is implemented
  - Q – Quad-precision (128 bit) floating point is supported
  - C – Compressed (i.e., 16 bit) instructions are supported
  - E – Embedded microprocessors, with only 16 registers

RV32G = RV32IMAFD

RISC-V variants

- 3 address widths:
  - RV32
  - RV64
  - RV128
- Future extensions:
  - L – Decimal arithmetic instructions
  - V – Vector arithmetic instructions
  - P – Packed SIMD instructions
  - B – Bit manipulation instructions
  - T – Transactional memory support

RV32G = RV32IMAFD

Main memory

- 32-bit, 64-bit or 128-bit (virtual) address widths
- Little-endian
- Supports unaligned access
- Virtual memory: paging
- I/O: memory mapped

RISC-V registers

- 32 general purpose registers
- if floating point is supported, there will be additional 32 floating point registers
- “it would be possible to define a non-standard subset integer RISC-V ISA with 16 registers” (Waterman et al. 2014)
- Register widths either 32, 64 or 128 bits

RISC-V register set

31	0
x0/zero	
x1	
x2	
x3	
x4	
x5	
x6	
x7	
x8	
x9	
x10	
x11	
x12	
x13	
x14	
x15	

32

31	0
x16	
x17	
x18	
x19	
x20	
x21	
x22	
x23	
x24	
x25	
x26	
x27	
x28	
x29	
x30	
x31	

32

31	0
pc	

32

RV32I Programmer-visible register set (Waterman 2016)

Instructions

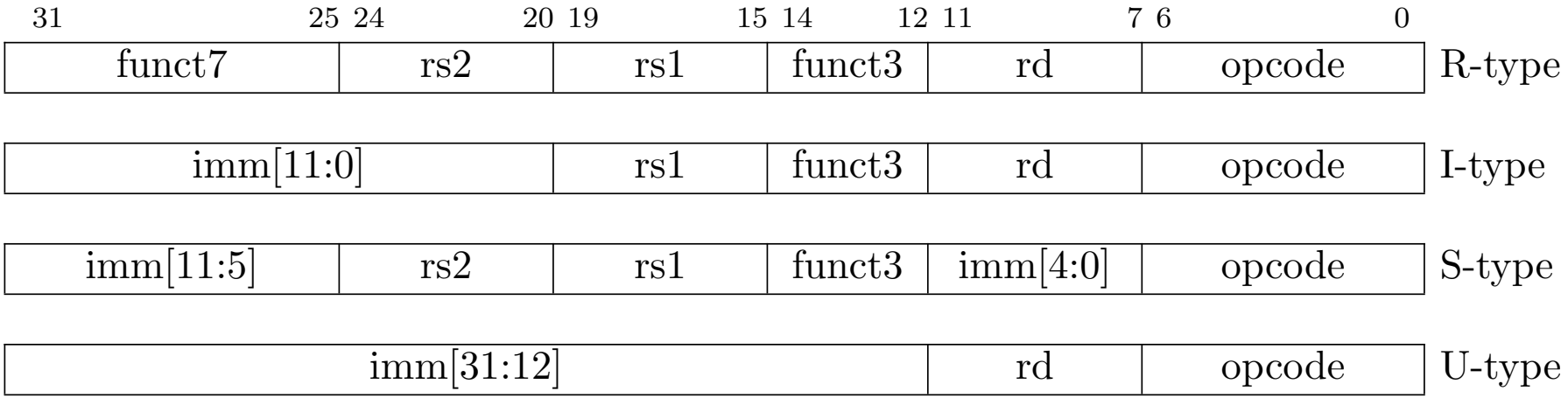
- all basic instructions are 32 bit wide
- must be stored at word-aligned memory locations
- ... but 16 bit compressed extensions relax alignment to 16 bits
- Instructions for the RV64 and RV128 variants are also 32 bits long
- RISC-V is a “three address” architecture. E.g.:

```
add x4,x5,x7 # x4 = x5 + x7
```

- 16-bit instructions are optionally supported to compress code (the “C” extension), but they are *synonyms* of the standard 32-bit instructions

(Porter 2018)

Instruction formats



(Waterman et al. 2014)

Instruction types

- R-type instructions:

```
add x3,x5,x6 # x3 = x5 + x6
```

Instruction types

- R-type instructions:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-type instructions:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

Instruction types

- R-type instructions:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-type instructions:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

- S-type instructions:

```
sw x4,8(x6) # Mem[8+r6] = x4 (word)
```

Instruction types

- R-type instructions:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-type instructions:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

- S-type instructions:

```
sw x4,8(x6) # Mem[8+r6] = x4 (word)
```

- B-type instructions (a variant of S-type):

```
blt x4,x6,loop # if x4<x6, goto offset + pc
```

Instruction types

- R-type instructions:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-type instructions:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

- S-type instructions:

```
sw x4,8(x6) # Mem[8+r6] = x4 (word)
```

- B-type instructions (a variant of S-type):

```
blt x4,x6,loop # if x4<x6, goto offset + pc
```

- U-type instructions:

```
lui x4,0x12AB7 # x4 = value<<12
auipc x4,0x12AB7 # x4 = (value<<12) + pc
```

Instruction types

- R-type instructions:

```
add x3,x5,x6 # x3 = x5 + x6
```

- I-type instructions:

```
addi x4,x6,123 # x4 = x6+123
lw x4,8(x6) # x4 = Mem[8+x6]
```

- S-type instructions:

```
sw x4,8(x6) # Mem[8+r6] = x4 (word)
```

- B-type instructions (a variant of S-type):

```
blt x4,x6,loop # if x4<x6, goto offset + pc
```

- U-type instructions:

```
lui x4,0x12AB7 # x4 = value<<12
auipc x4,0x12AB7 # x4 = (value<<12) + pc
```

- J-type instructions (a variant of U-type):

```
jal # call: pc = offset + pc; x4 = ret add
```

Addressing modes

Just 2 addressing modes!

- immediate (operand in the instruction)
- register indirect + offset

Immediate values:

```
addi x5,x0,21

lui x6,x0,0x1234
addi x6,x0,0x5678 # x6 = 0x12345678
```

Memory access instructions

Instruction	Format	Meaning
`lb rd, imm[11:0](rs1)`	I	Load byte, signed
`lbu rd, imm[11:0](rs1)`	I	Load byte, unsigned
`lh rd, imm[11:0](rs1)`	I	Load half-word, signed
`lhu rd, imm[11:0](rs1)`	I	Load half-word, unsigned
`lw rd, imm[11:0](rs1)`	I	Load word
`sb rs2, imm[11:0](rs1)`	S	Store byte
`sh rs2, imm[11:0](rs1)`	S	Store half-word
`sw rs2, imm[11:0](rs1)`	S	Store word
`fence pred, succ`	I	Memory ordering fence
`fence.i`	I	Instruction memory ordering fence

(Waterman 2016)

Register x0

Register `x0` always contains `0` (and writes to it are ignored).
Interregister `MOV` instruction not necessary!

```
add x6,x7,x0
```

is used instead of

```
mov x6,x7
```

Register x1

Register `x1` is used to store return address during subroutine calls, **by convention** (any other register could do!).

The return is then just `jr x1` (Jump using Register)

For recursive calls x1 needs to be saved!

MUL implementation

Lower part bits of multiplication are the same for signed and unsigned multiplication $\Rightarrow$ only one `MUL` instruction is needed:

```
mul x4,x9,x13 # x4 = x9*x13
```

MUL implementation

Lower part bits of multiplication are the same for signed and unsigned multiplication $\Rightarrow$ only one `MUL` instruction is needed:

```
mul x4,x9,x13 # x4 = x9*x13
```

However the higher portion can be different for signed and unsigned operands, thus 3 instructions produce high bits:

```
MULH - signed operands
MULHU - unsigned operands
MULHSU - one signed operand and one unsigned operand
```

MUL implementation

Lower part bits of multiplication are the same for signed and unsigned multiplication $\Rightarrow$ only one `MUL` instruction is needed:

```
mul x4,x9,x13 # x4 = x9*x13
```

However the higher portion can be different for signed and unsigned operands, thus 3 instructions produce high bits:

```
MULH - signed operands
MULHU - unsigned operands
MULHSU - one signed operand and one unsigned operand
```

Operations should be performed in this order and can then be (optionally) fused by hardware:

```
mulh x4,x9,x13 # compute upper half
mul  x5,x9,x13 # compute lower half
# The product is now in the register pair x4:x5
```

JMP implementation

Instruction for `JMP` is *always* jump-and-link: `jal` or `jalr`.

But, if you do not need the return address, use `x0` as a link register :)

```
jal x0, loop
```

Stack. x2

- A function that requires stack storage will grow the stack (always by a multiple of 16) by subtracting from the stack top pointer sp.
- Variables within the stack frame can be addressed using positive offsets from register x2.

GP: x3

- The RISC-V calling convention is to place the global variables together and initialise a register to point to this area.
- By convention, register x3 is used for this.
- The individual variables can be conveniently addressed by using a small offset from the global pointer.
- The global pointer is typically initialised early in the program and never changed. So, in some sense, it is “callee-saved”

Other registers

- Register `x4` – The Thread Base Pointer (“tp”)
- Register x8,x9,x18-x27 – Saved Registers (“s0-s11”) (*Callee saved*)
- Register x5-x7,x28-x31 – Temporary Registers (“t0-t6”) (*Caller saved*)
- Register x10-x17 - Argument Registers (“a0-a7”) [2]

[2] Floating point arguments are passed in the floating point, `fn` registers if they exist.

Example: multiple precision add

```
li t0,0x12345678
li t1,0xFFEDCBA9
li t2,0x1F
li t3,0x44
add t5,t1,t0
sltu t4,t5,t0 # Determine the carry bit.
add t6,t2,t3
add t6,t6,t4 # Add the carry bit
# t6 now contains the double-machine-word (64 bit) sum.
# Exit to the caller:
li a0,0
li a7,93
ecall
ebreak
```

Example: multiple precision add

```
li t0,0x12345678
li t1,0xFFEDCBA9
li t2,0x1F
li t3,0x44
add t5,t1,t0
add t6,t2,t3
bgeu t5,t0,nocarry # Determine the carry bit.
addi t6,t6,1 # Add the carry bit
nocarry:
# t6 now contains the double-machine-word (64 bit) sum.
# Exit to the caller:
li a0,0
li a7,93
ecall
ebreak
```

References

Alexander, W. C. et al. (1975). “Static and Dynamic characteristics of XPL programs”. In: *Computer* 8.11, pp. 41–46. DOI: 10.1109/c-m.1975.218804.

Jamil, T. (1995). “RISC versus CISC”. In: *IEEE Potentials* 14.3, pp. 13–16. DOI: 10.1109/45.464688.

Patterson, David A. et al. (2003). *The case for the reduced instruction set computer*. Tech. rep. Computer Science Division, University of California; Bell Laboratories, Science Research Center, pp. 25–33. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.68.9623.

Peuto, B. L. et al. (1977). “An instruction timing model of CPU performance”. In: *Fourth Annual Symposium on Computer Architecture*, pp. 165–178. DOI: 10.1145/800255.810667.

Porter III, Harry H. (2018). *RISC-V: an overview of the instruction set architecture*. URL: https://web.cecs.pdx.edu/~harry/riscv/RISCV-Summary.pdf.

Waterman, Andrew (Jan. 2016). “Design of the RISC-V instruction set architecture”. PhD thesis. Electrical Engineering and Computer Sciences, University of California at Berkeley, pp. 1–117. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

Waterman, Andrew et al. (2014). *The RISC-V instruction set manual, volume I: user-level ISA, version 2.0*. Tech. rep. UCB/EECS-2014-54. Electrical Engineering and Computer Sciences, University of California at Berkeley, pp. 1–102. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf.