DNA
Processor Types And Instruction Sets

Alexandre David
1.2.05
adavid@cs.aau.dk

Disclaimer

- Not everything from chapter 5 really applies.
- Replace
  "the programmer"
  by
  "the compiler".

What Instructions Should A Processor Offer?

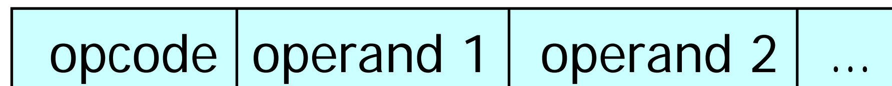
- Answer: It's a tradeoff.
  - Minimum: +1, -1, branch if != 0 ??
  - Very large – cost and performance issues.
  - Factors:
    - size
    - use (often/rare)
    - power
    - domain specific (simd,rsa)
  - → convenient for programmers, good for power and cost.

Representation Considerations

- Set of instructions
  - which type? (arithmetic/memory access/floating)
  - how to code them
  - semantics of the instructions
  - definition of the set must be precise
    See Intel's manuals.

Instructions

- Generally (all processors), the parts are:
  - opcode
    type of instruction – operation code (hexa)
  - operands
    arguments of the instruction
  - result location
    where to put the result
  - Example Intel's manual.



Instruction Length

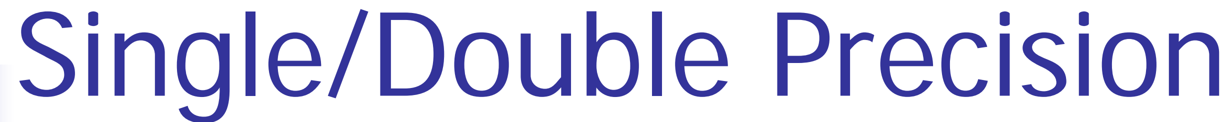
- Variable
  - longer/shorter, variable # of operands
  - better memory usage
- Fixed-length
  - same size for every instruction (opcode+operands)
  - simpler hardware
  - faster hardware (?)
  - the point: not all instructions need to use all the fields! (neg/not: 1 input, add/sub: 2 inputs)

Registers

- High speed storage device.
  - like local variables on chip
- Limited size – tight to architecture.
- Few of them – fast but expensive.
- Basic operations: fetch & store.
- Sometimes numbered from 0 to N-1.
  - x86 mess
- Operands must be in registers most of the time.
  - x86 mess

Floating Point Registers

- Separate set because purpose and type of operations are different.
  - Operations on a different execution unit too!
- Hold floating point number (single/double precision).
  - x86 mess
- Operands must be placed in registers.
  - x86 mess++.



Single/Double Precision

- Applies to floats & ints.
- Double: twice as large as “usual”.
  - Usually floats are on 32 bits.
  - In fact they were already double precision numbers on 16 bits processors.
  - 64 bits floats are considered double precision (double type).
  - x86 mess: internal 80 bits registers.
- For ints: long int (32 or 64 bits) – also **emulation** of long types.
  - Use pairs of registers (eax:edx).

Example

- Task: do Z = X + Y.
- Steps (hypothetical instruction sets):
  - load r3, @X
  - load r4, @Y
  - add r3, r4
  - store @Z,r3
  - (assuming r3 and r4 are *free*
    this is the job of the *compiler*).

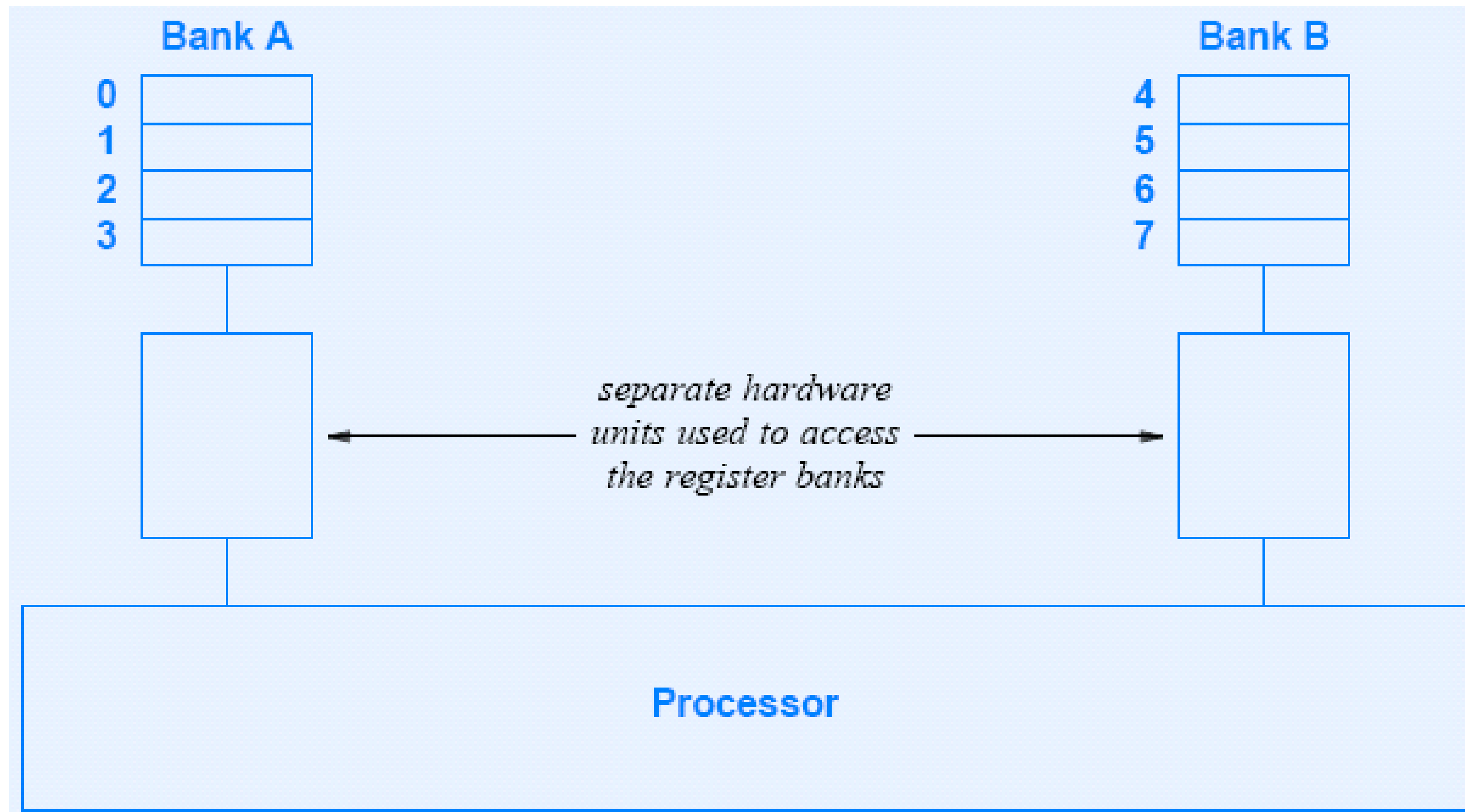
Terminology

- Register spilling
  - Saving values in memory for later use.
  - When no more registers available.
- Register allocation:
  - Choose what to keep when in **registers**.
    - Complex programs have many variables, constants etc.. but few registers are available.
    - Compiler decides.
    - What to do when you run out of registers? (There's a stack BTW).

Register Banks

- Sometimes registers are partitioned into banks.
  - Allows parallel access.
  - Restrict use.
- What's really relevant in fact: Groups of registers.
  - General purpose.
  - Floating point.
  - Reserved for memory accesses.
  - Extensions – MMX/SSE

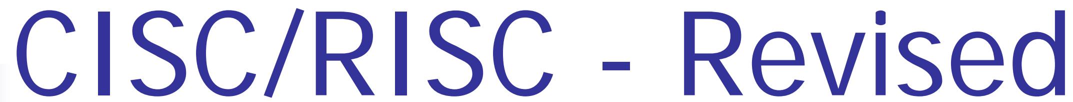
Register Banks



Types of Instruction Sets

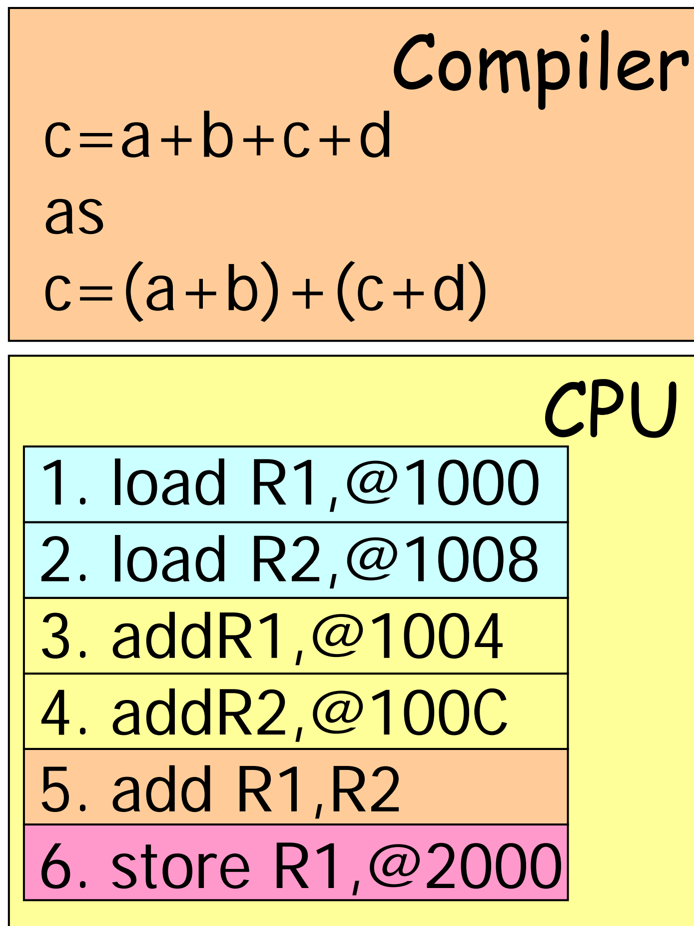
- CISC *(complex instruction set computer)*
  - many instructions
  - variable time
- RISC *(reduced instruction set computer)*
  - few instructions
  - single clock cycle
  - no floating point instructions
  - MIPS
- In fact we should revise that.

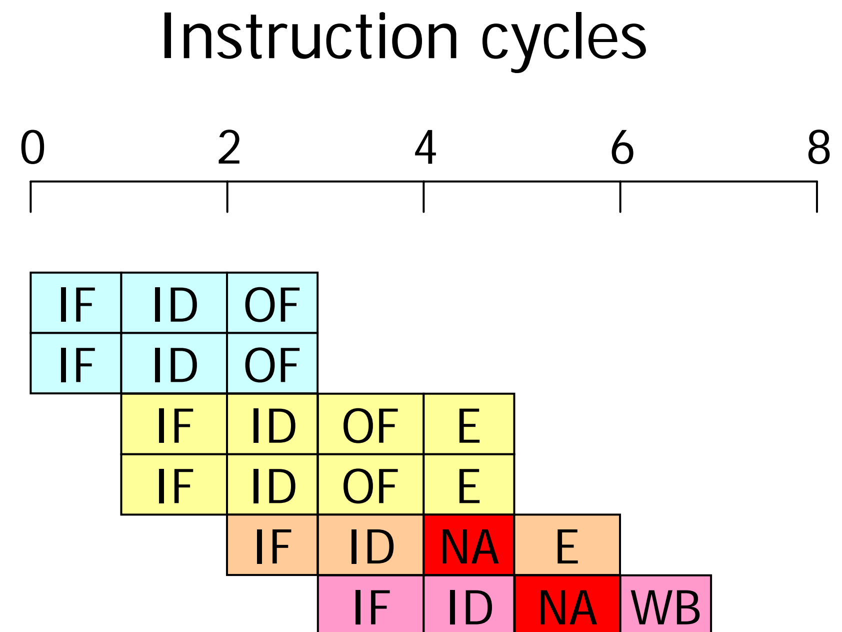
CISC/RISC - Revised



- CISC
  - complex set, variable size
  - x86
- RISC
  - reduced set, fixed-size
  - not necessarily on cycle/instruction
  - support for floats
  - PPC
- *The religious question of CISC vs. RISC is irrelevant now because all processors are RISC anyway.*

Pipelining and Superscalar Execution – Recall (MVP)

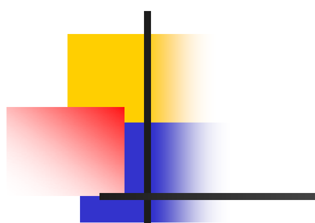




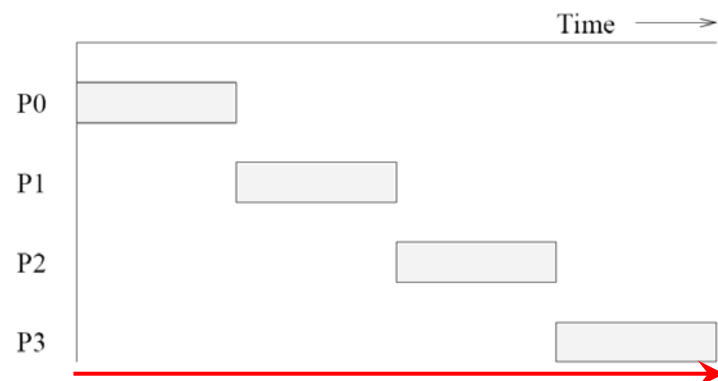
2x IF, ID, OF, … in the same cycle: **superscalar**.

Pipelines

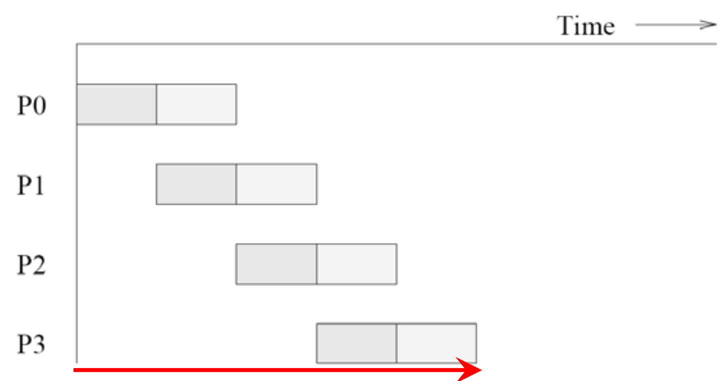
- Superscalar: if several execution units of the same type are present.
  - Exec 2 add at the same time.
- Depth of the pipeline: # of steps.
  - Pentium 4 – deep pipeline (20 states)
  - Pentium M – shorter (~15)
  - Interferes with branch prediction.
  - Interferes with out-of-order execution.
  - Amount of work @ each step influences maximum frequency.
- Used in all modern CPUs – RISC/CISC blurred.



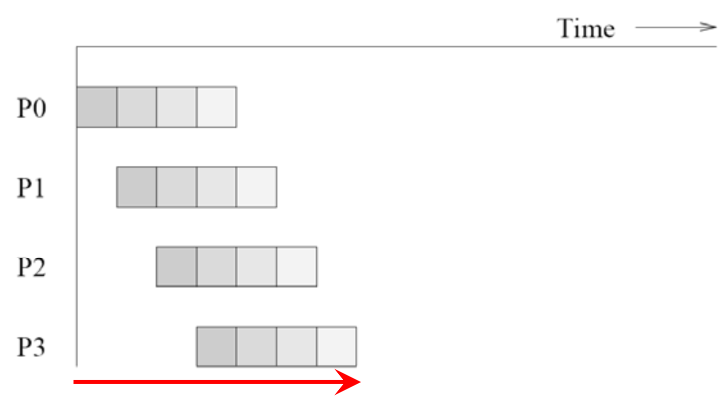
(Misuse of figure).



(a) A single message sent over a store-and-forward network



(b) The same message broken into two parts and sent over the network.



(c) The same message broken into four parts and sent over the network.

Figure 2.26 Passing a message from node P_0 to P_3 (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.

Basic Steps In Execution

- Fetch and execute cycle.
  - Fetch next instruction.
  - (Opcode tells # of operands.)
  - Fetch each operand.
  - Execute operation.
  - Store result.
  - Repeat.
- How to start?

How to Optimize?

- Ask Intel/AMD.
- Multiple dedicated units.
- Use the units in parallel.
- Out-of-order execution (deduce dependencies & reorder).
- Pipeline.
- Branch prediction.
  - Duplicate execution.
- Speculative execution.
- VLIW (Itanium's very large instruction word)

Pipeline cont.

- Use is automatic
  - it **stalls** automatically
    - dependencies not met
    - need operands from memory
    - need operands from other operations
    - call/jump/branch
    - wait for I/O or co-processor
  - effect of stalls: waste cycles
    - “bubble” passing through a pipeline
  - idea of hyper-threading: 2 execution threads share the same pipeline & units to fully use them.

Optimizing Programs

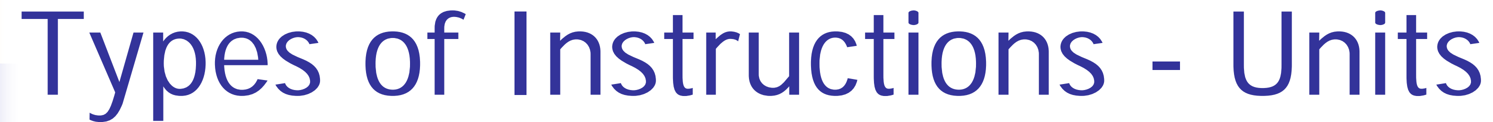
- Compiler's job.
- Pipeline friendly.
  - Move instructions around to reduce stalls.
  - Reduce branches.
  - Note: some special instructions (conditional moves).
  - Use *forwarding* – avoid store and use result for next instruction.

Example

(a)					(b)				
C	$\leftarrow$	add	A	B	C	$\leftarrow$	add	A	B
D	$\leftarrow$	subtract	E	C	F	$\leftarrow$	add	G	H
F	$\leftarrow$	add	G	H	M	$\leftarrow$	add	K	L
J	$\leftarrow$	subtract	I	F	D	$\leftarrow$	subtract	E	C
M	$\leftarrow$	add	K	L	J	$\leftarrow$	subtract	I	F
P	$\leftarrow$	subtract	M	N	P	$\leftarrow$	subtract	M	N

No-Op Instructions

- Book is wrong here.
- Have an effect on program counter.
- Have no effect on
  - registers
  - condition registers (flags)
- Used by compilers to *align* code.
  - Different types/sizes.



Types of Instructions - Units

- Arithmetic operations – ALU
- Logical operations – ALU
- Data access, transfers – MMU
- Jumps (conditional or not) – Main control unit (special)
- Floating point instructions – FPU
- Processor control instructions – special, e.g., cpuid.

Some Special Registers

- Program counter – instruction pointer register
  - @ of next instruction to execute
- Stack registers – work together with push/pop instructions.
  - Programs have a statically allocated stack.
- Condition registers – condition code.
  - ZF, CF, PF, OF…
- Extensions (SSE – xmm*/MMX – mm*/Altivec…)

Branches

- Absolute jumps – rarely used.
  - Interrupts – bios.
- Relative jumps
  - not conditional
  - conditional – after an instruction that changes the condition register.
    - test/cmp/add/sub…
- Calls – a branch in a sense
  - before a call: load arguments
  - push IP on the stack – jump
  - return: pop IP from the stack
  - after a call: use results
  - before/after: use registers or the stack (check 32/64 bits)

Condition Jump Example

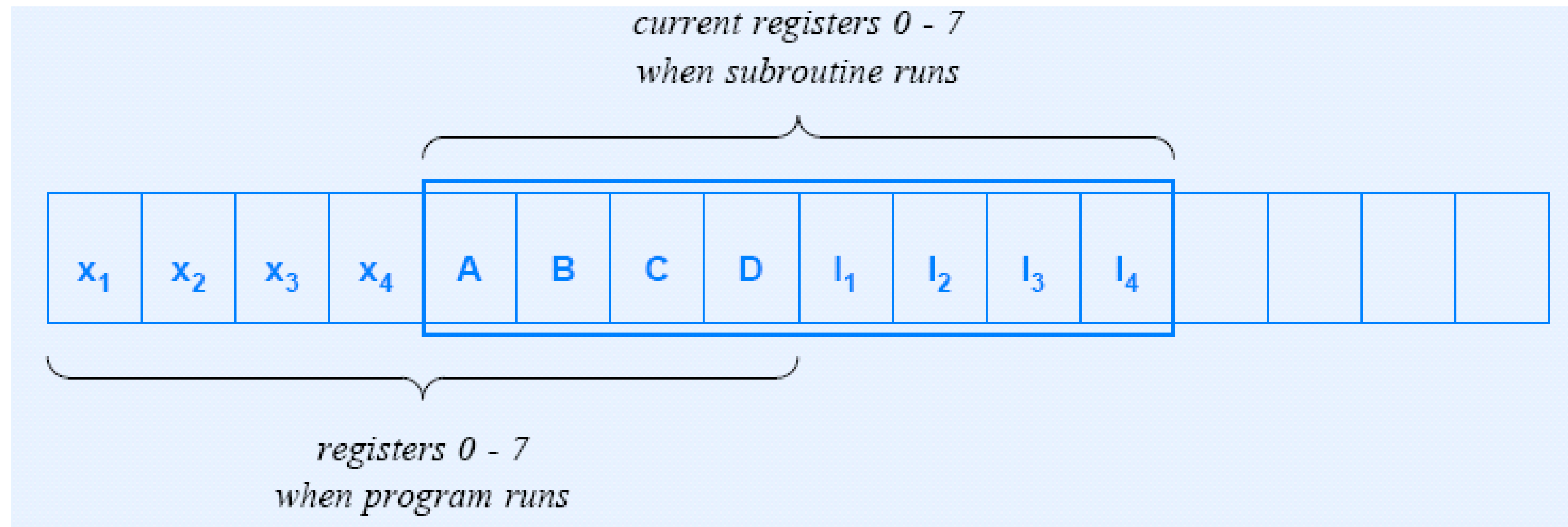
```
cmp    r4, r5    # compare regs. 4 & 5, and set condition code
be     lab1      # branch to lab1 if cond. code specifies equal
mov    r3, 0     # place a zero in register 3
```

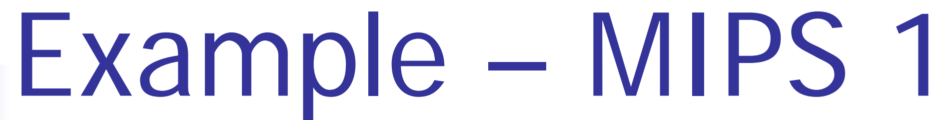
Note: condition moves also available on x86.

Register Window

- Hardware optimization for argument passing.
- Show only a subset of the real internal registers – at a time.
  - Slide the window upon calls.
  - Arguments can be loaded into these special registers.
  - Slide → keep arguments visible + expose new registers for future calls.

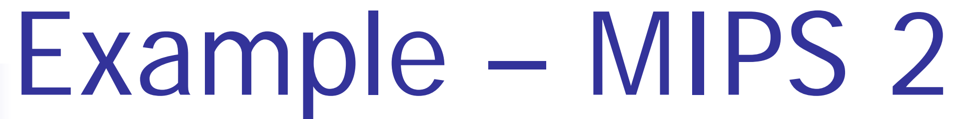
Register Window





Example – MIPS 1

Instruction	Meaning
Arithmetic	
add	integer addition
subtract	integer subtraction
add immediate	integer addition (register + constant)
add unsigned	unsigned integer addition
subtract unsigned	unsigned integer subtraction
add immediate unsigned	unsigned addition with a constant
move from coprocessor	access coprocessor register
multiply	integer multiplication
multiply unsigned	unsigned integer multiplication
divide	integer division
divide unsigned	unsigned integer division
move from Hi	access high-order register
move from Lo	access low-order register
Logical (Boolean)	
and	logical *and* (two registers)
or	logical *or* (two registers)
and immediate	*and* of register and constant
or immediate	*or* of register and constant
shift left logical	Shift register left N bits
shift right logical	Shift register right N bits



Example – MIPS 2

Instruction	Meaning
Data Transfer	
load word	load register from memory
store word	store register into memory
load upper immediate	place constant in upper sixteen bits of register
move from coproc. register	obtain a value from a coprocessor
Conditional Branch	
branch equal	branch if two registers equal
branch not equal	branch if two registers unequal
set on less than	compare two registers
set less than immediate	compare register and constant
set less than unsigned	compare unsigned registers
set less than immediate	compare unsigned register and constant
Unconditional Branch	
jump	go to target address
jump register	go to address in register
jump and link	procedure call

Instruction	Meaning
Arithmetic	
FP add	floating point addition
FP subtract	floating point subtraction
FP multiply	floating point multiplication
FP divide	floating point division
FP add double	double-precision addition
FP subtract double	double-precision subtraction
FP multiply double	double-precision multiplication
FP divide double	double-precision division
Data Transfer	
load word coprocessor	load value into FP register
store word coprocessor	store FP register to memory
Conditional Branch	
branch FP true	branch if FP condition is true
branch FP false	branch if FP condition is false
FP compare single	compare two FP registers
FP compare double	compare two double precision values

Comments

- No “mov reg, reg” ?
  - Use add r1,r2,r3 with a special register for 0.
- Minimum set of instruction.
- Orthogonality.
  - No overlap/duplication.
- But also several instructions for some operations.