NTNU

Det skapende universitet

TDT4255 Computer Design

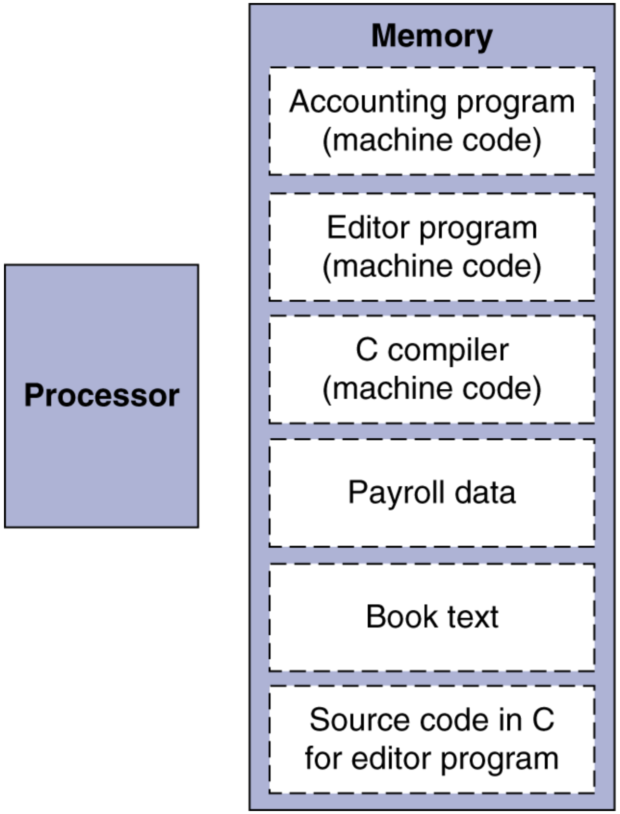
Lecture 2: Instruction Set Architecture (ISA)

Magnus Jahre

Appendix A

Instruction Set Principles

Stored Program Computers

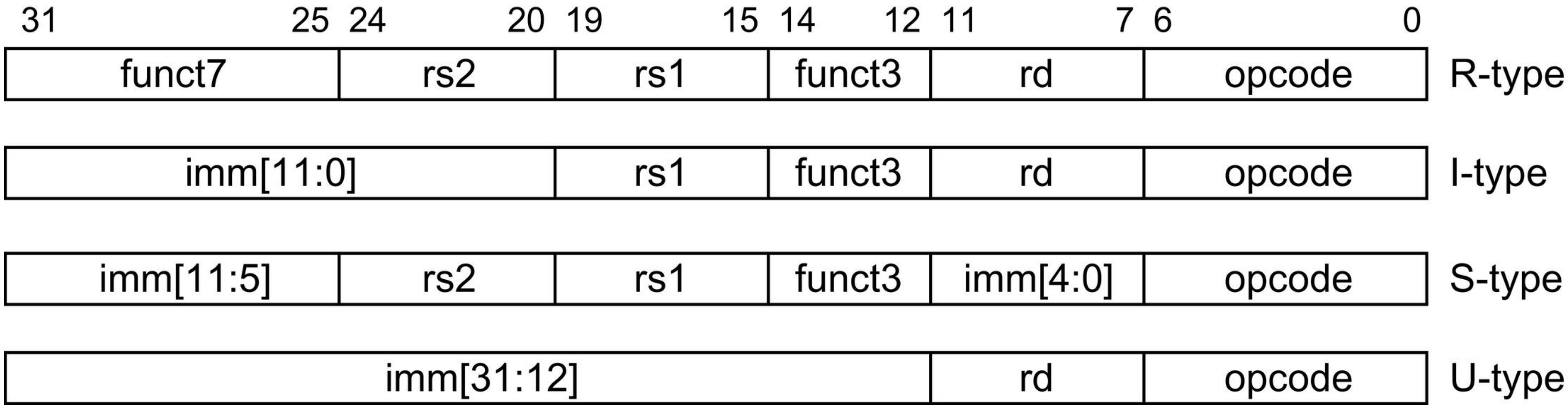


- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

RISC-V ISA Example



31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

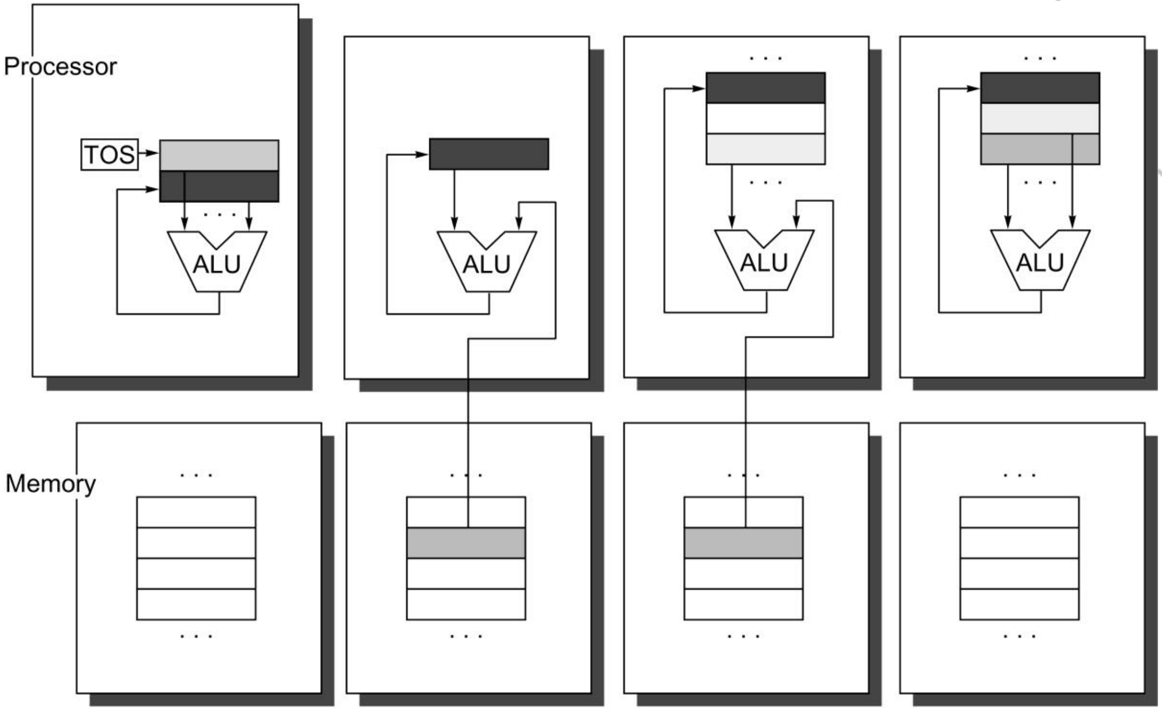
- A fixed (small) number of instruction types
- All types have an operation code (opcode)
- Format is regular across instruction types

High-level View on ISA Design

- ISA design is focused on the “common case”
  - Figure out what the common case is
  - Make sure that the common case can be efficiently expressed in the ISA
  - Make sure that there are ways to express less common cases as well
- We will systematically go through instruction selection trade-offs
  - Lots of data – need to focus on the insight
  - The analysis leads to the RISC-V ISA definition
- There is no such thing as a perfect ISA – the data could equally well be used to argue for other choices (although likely not significantly different)
  - … and you can get very good performance (but possibly not the same efficiency) out of ISAs that were not designed using this data-driven approach (e.g., x86)

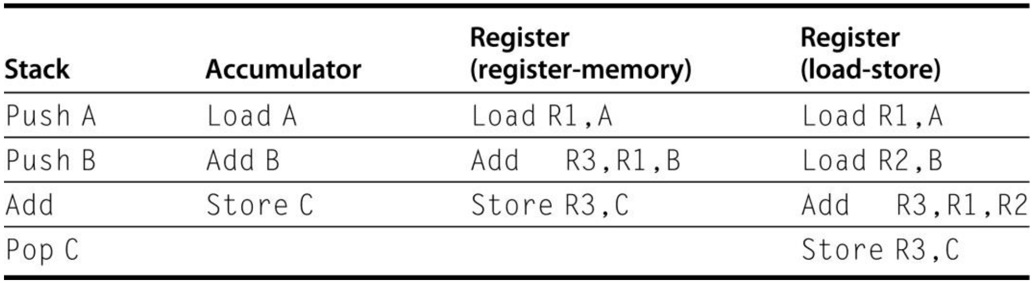
CLASSIFYING INSTRUCTION SET ARCHITECTURES

ISA Types



(A) Stack (B) Accumulator (C) Register-memory (D) Register-register/ load-store

C = A+B



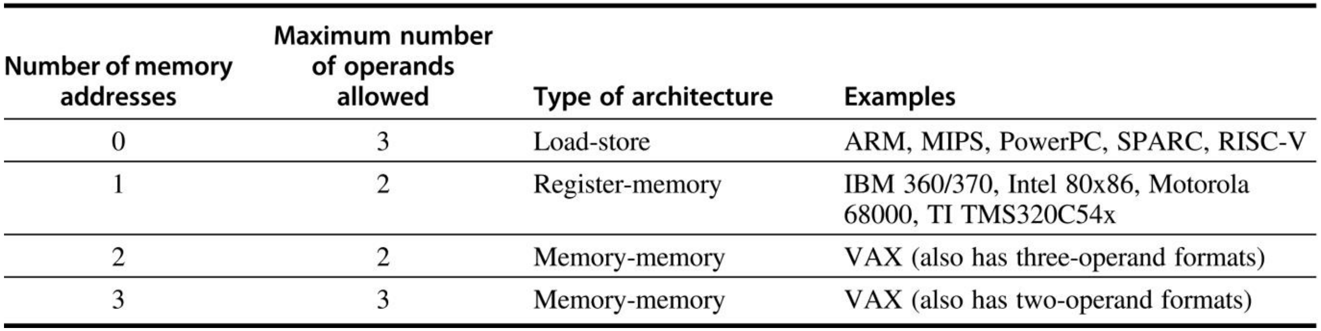
Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Register-based ISAs

- ISAs from the 80s onwards are commonly load-store register ISAs
  - Advantage 1: Accessing registers is faster than accessing memory
  - Advantage 2: Registers introduce fewer unnecessary dependencies than accumulator and stack ISAs

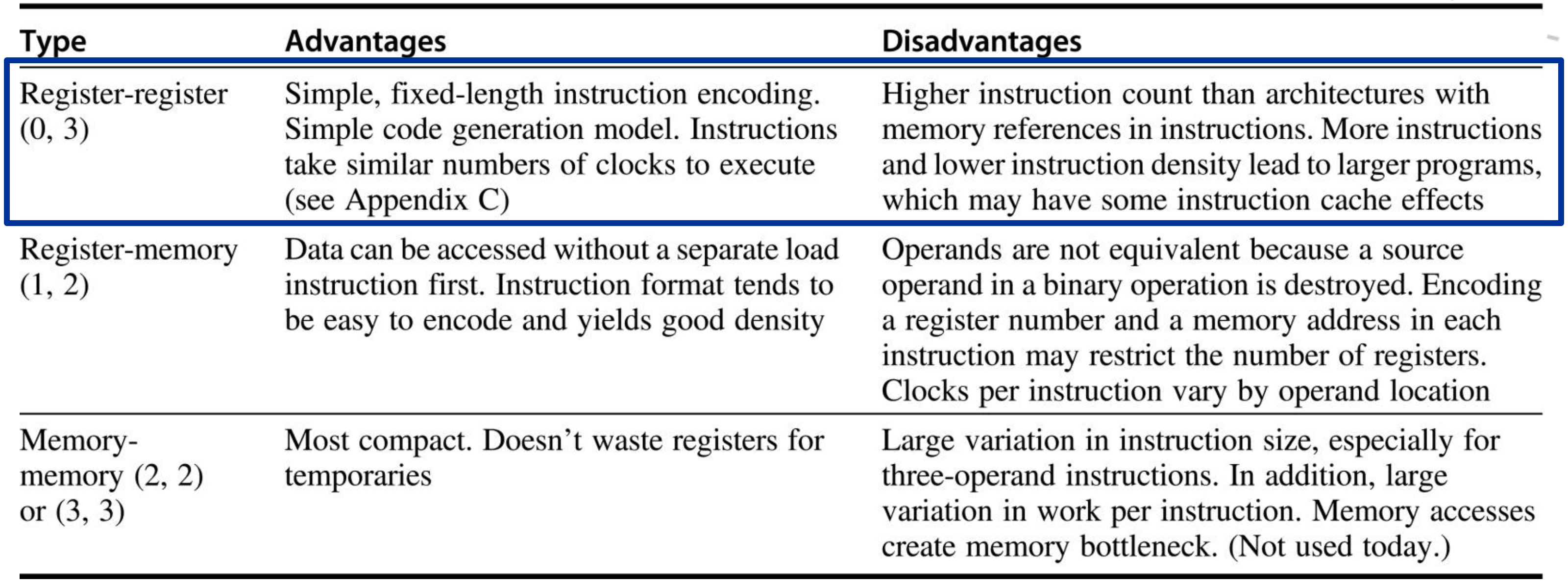
- How many registers?
  - Depends on the implementation, quite a few in general.
  - More registers give longer instructions because more bits are required for the register address
  - More registers makes compiler register allocation heuristics more efficient

- Memory vs. register operands:



Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Classifying ISAs



Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C)	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density	Operands are not equivalent because a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

"Best" choice

(memory operands, total operands)

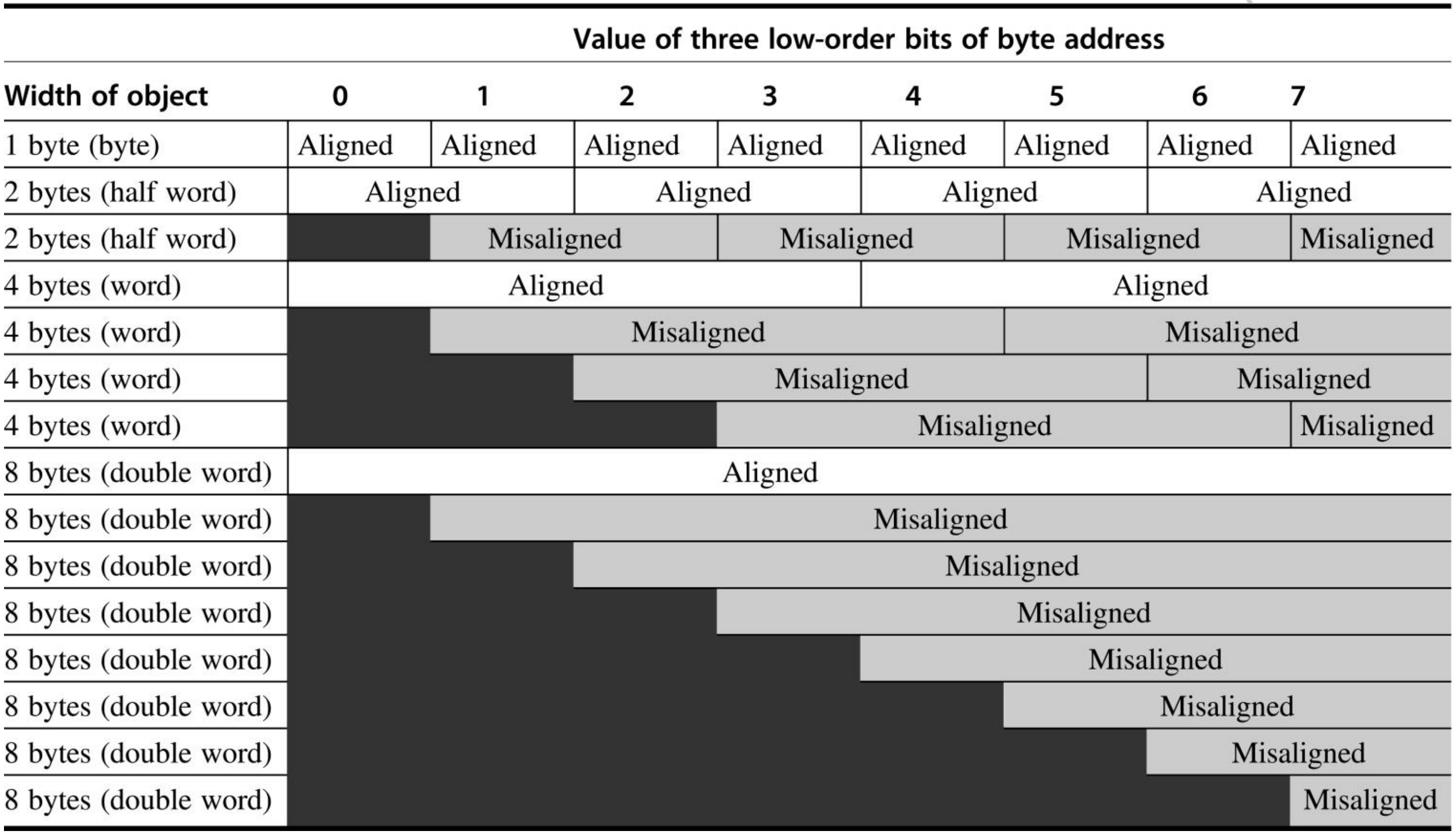
MEMORY ADDRESSING

Interpreting Memory

- Big vs. little endian
  - Ordering of bytes within a double word (0 is LSB and 7 in MSB, 8 bytes in total)
  - Little endian: 7 6 5 4 3 2 1 0
  - Big endian: 0 1 2 3 4 5 6 7

- Alignment
  - An address *A* of an object is aligned to size *s* if *A mod s == 0*
  - Many computers expect accesses larger than a byte to be aligned since this simplifies the implementation
  - Most computers do not support addressing items smaller than a single byte

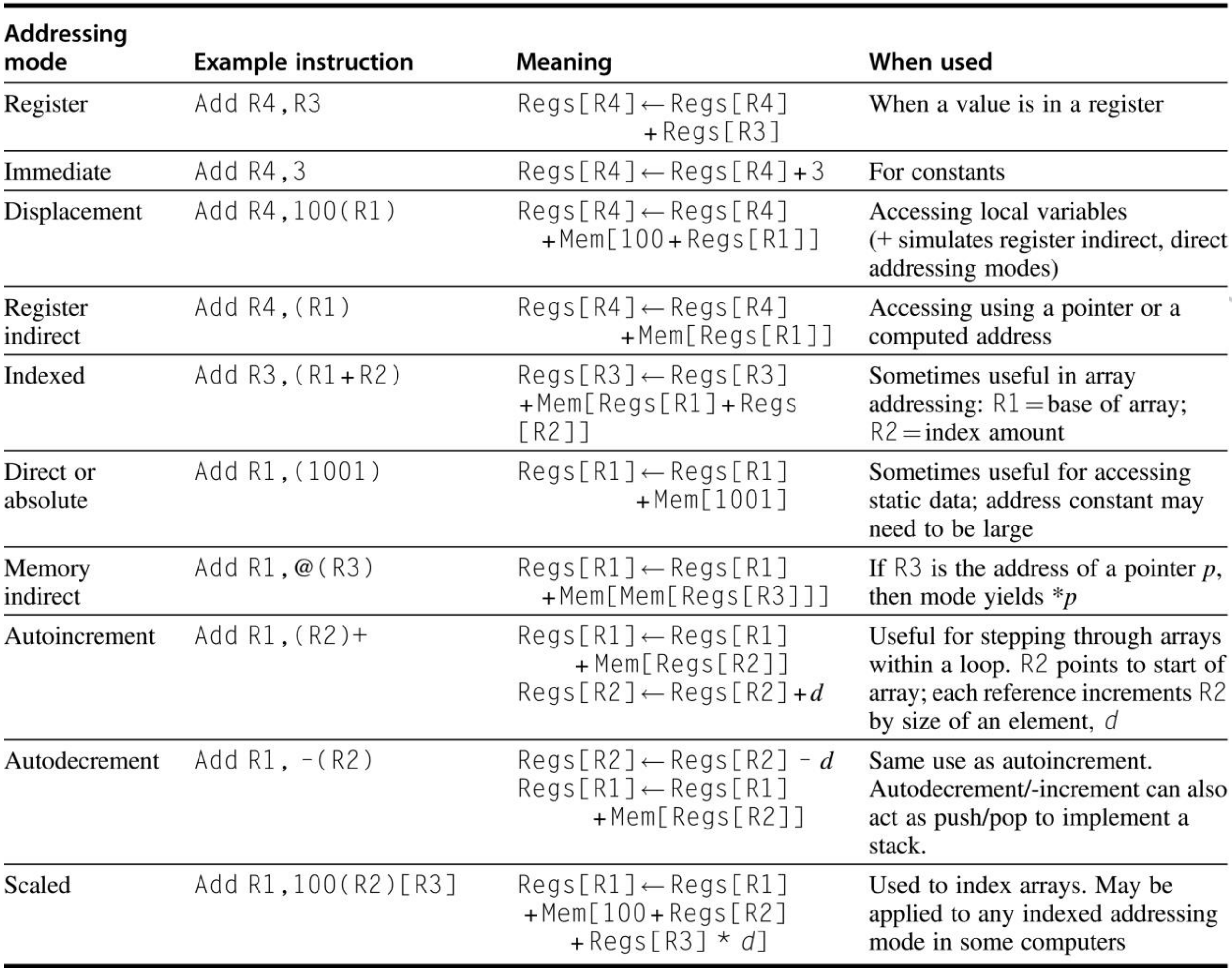
Alignment Examples



	Value of three low-order bits of byte address							
Width of object	0	1	2	3	4	5	6	7
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

Note the names of each object size: byte, half word, etc.

Addressing Modes



Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	Regs[R4]←Regs[R4] +Regs[R3]	When a value is in a register
Immediate	Add R4,3	Regs[R4]←Regs[R4]+3	For constants
Displacement	Add R4,100(R1)	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	Add R4,(R1)	Regs[R4]←Regs[R4] +Mem[Regs[R1]]	Accessing using a pointer or a computed address
Indexed	Add R3,(R1+R2)	Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs [R2]]	Sometimes useful in array addressing: R1=base of array; R2=index amount
Direct or absolute	Add R1,(1001)	Regs[R1]←Regs[R1] +Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	Add R1,@(R3)	Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer *p*, then mode yields **p*

Autoincrement	Add R1,(R2)+	Regs[R1]←Regs[R1] +Mem[Regs[R2]]
		Regs[R2]←Regs[R2]+*d*
			Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*

Autodecrement	Add R1, -(R2)	Regs[R2]←Regs[R2] - *d*
		Regs[R1]←Regs[R1] +Mem[Regs[R2]]
			Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.

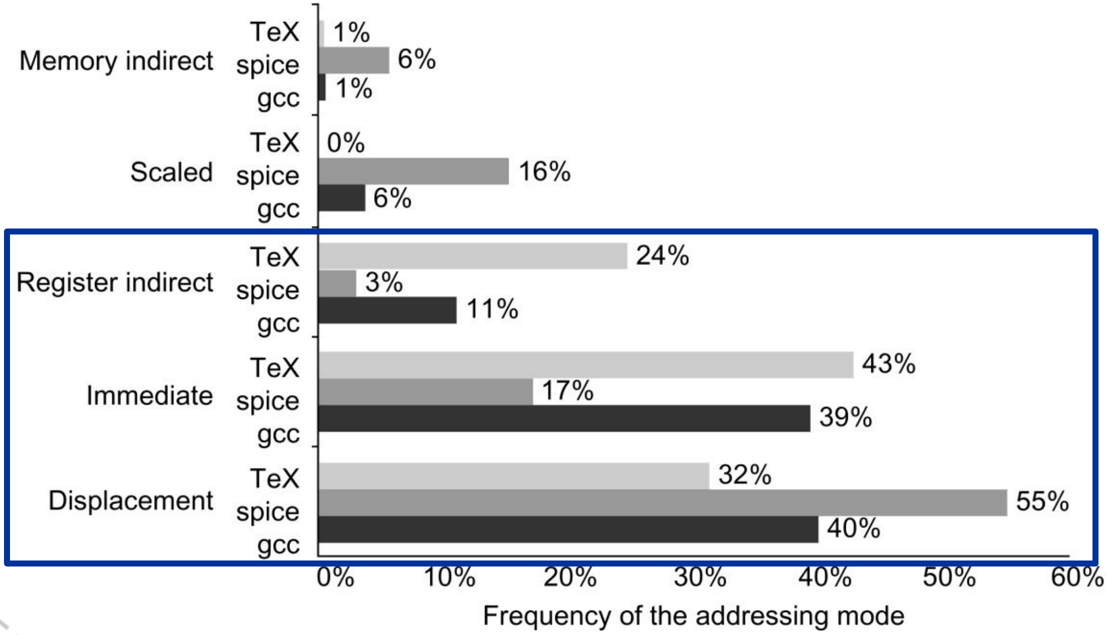
Scaled	Add R1,100(R2)[R3]	Regs[R1]←Regs[R1] +Mem[100+Regs[R2] +Regs[R3] * *d*]	Used to index arrays. May be applied to any indexed addressing mode in some computers

Effective address == the address that addressing mode specifies

Also: PC-relative addressing is not mentioned

Addressing Mode Selection

- Which addressing modes should we support:
  - Advantage: More addressing modes reduce instruction count significantly
  - Disadvantage: More addressing modes complicate the CPU – possibly to the point of increasing average cycles per instruction (CPI)

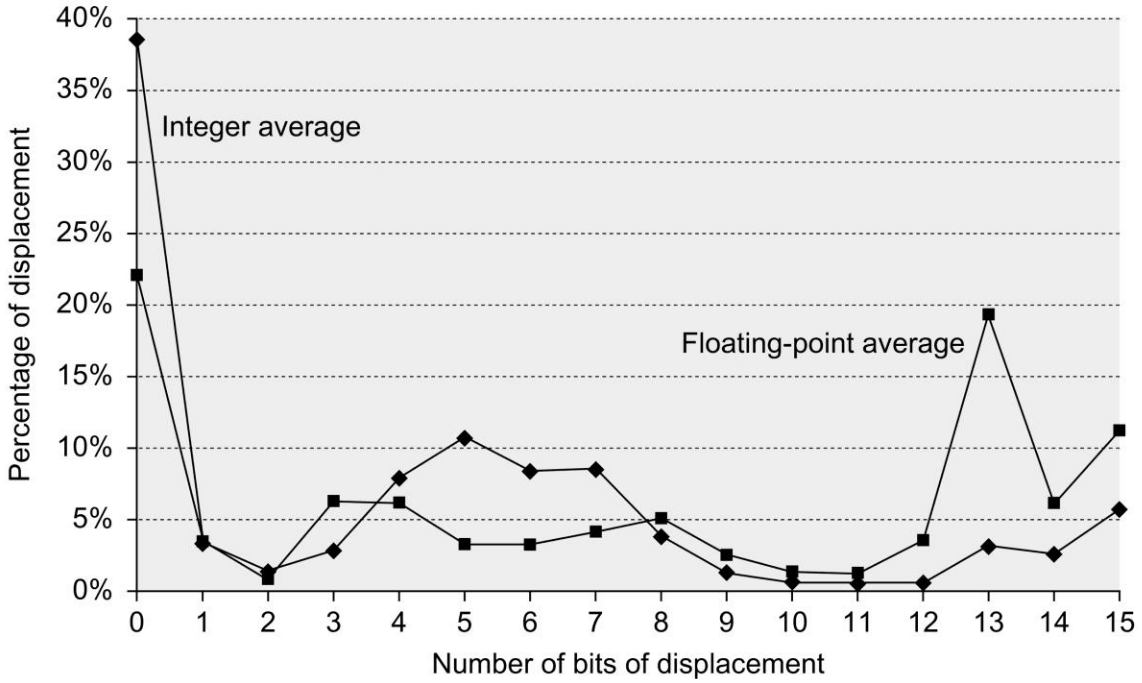


Most common

Note: Evaluation is a bit tricky since the use of addressing modes depends heavily on the choices of the compiler (workaround: use the VAX)

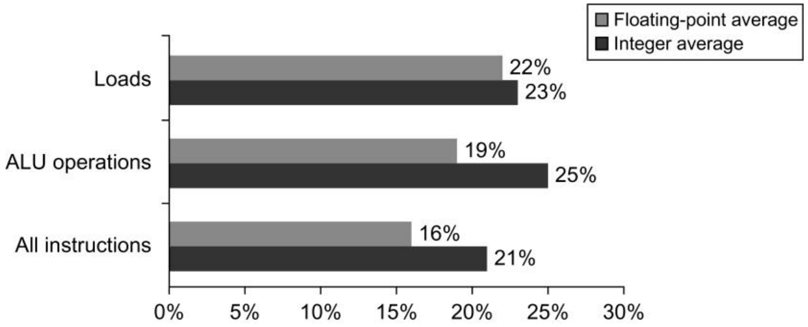
Displacement Size Histogram

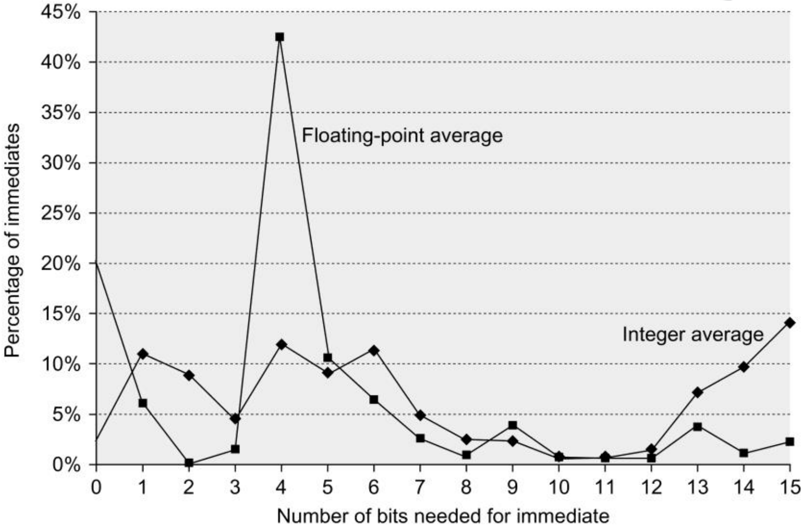
Note: From here on the analyses assume the Alpha ISA with full compiler optimization across SPEC2000 (both integer and floating point)



Key takeaway: Displacement sizes are widely distributed

Immediates





Immediates are common

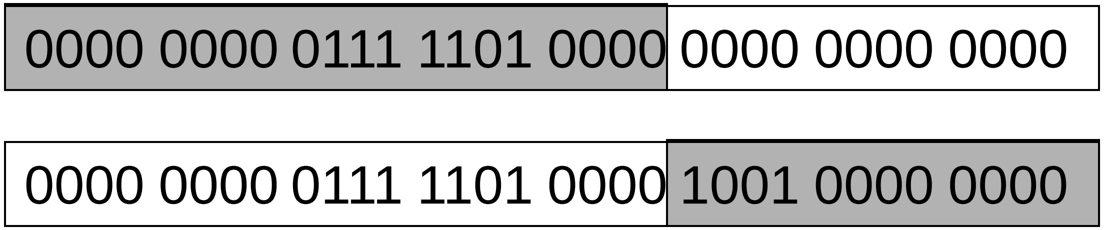
Immediates are mostly small, but some are larger

Large Immediates

- Analysis shows that most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

  `lui rd, immediate`
  - Copies 20-bit constant to left 20 bits of rd
  - Clears right 12 bits of rr to 0

```
lui x0, 61

ori x0, x0, 2304
```



0000 0000 0111 1101 0000 0000 0000 0000

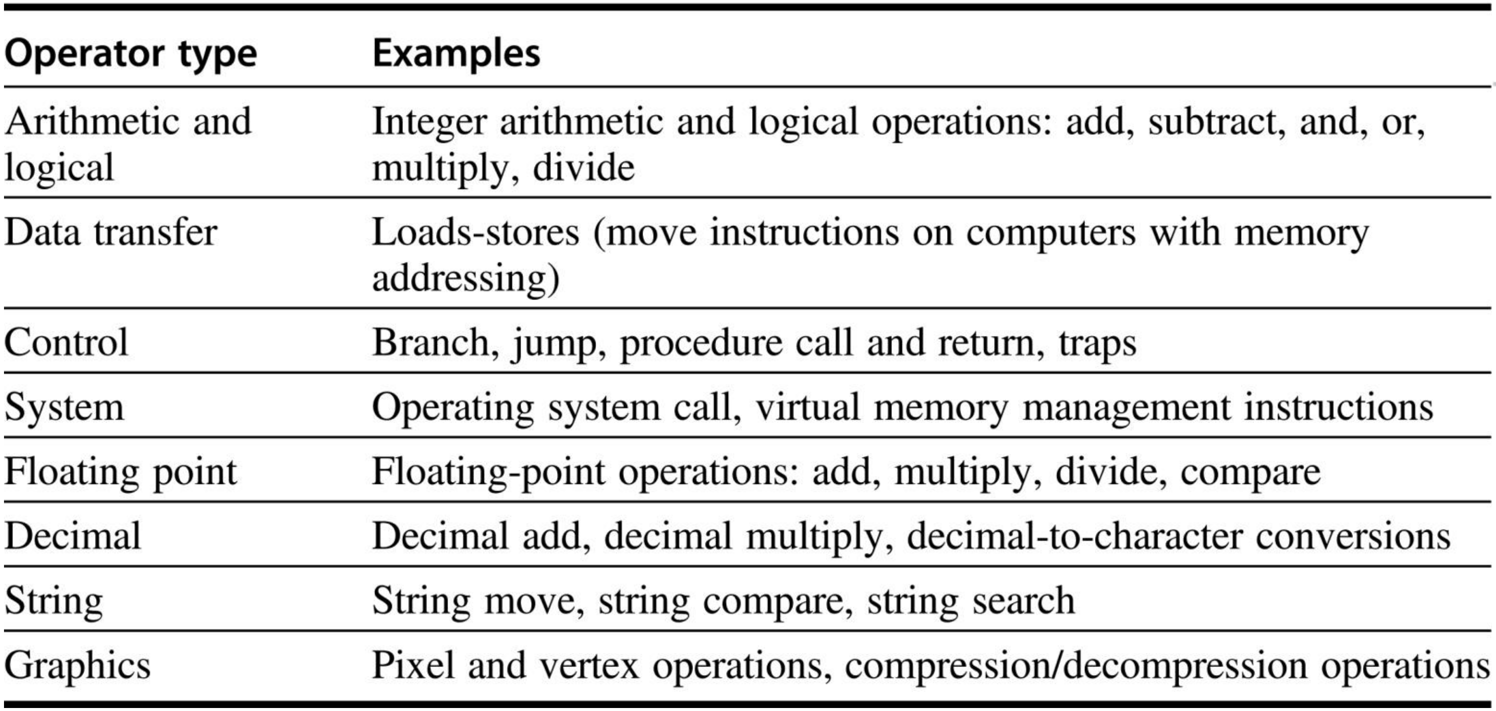
0000 0000 0111 1101 0000 1001 0000 0000

Addressing Mode Summary

- A new ISA should support displacement, immediate and register indirect addressing
  - These capture 75% - 99% of the addressing modes used in the textbook experiments
- Displacement address should be 12 to 16 bits
  - Captures 75% - 99% of displacements
- Immediate fields should be between 8 and 16 bits
  - 8 bit immediate captures around 85% (floating point) and 65% (integer) of all immediates
  - That the *lui* (load upper immediate) instruction can support 32 bit immediates favor a 16 bit immediate field

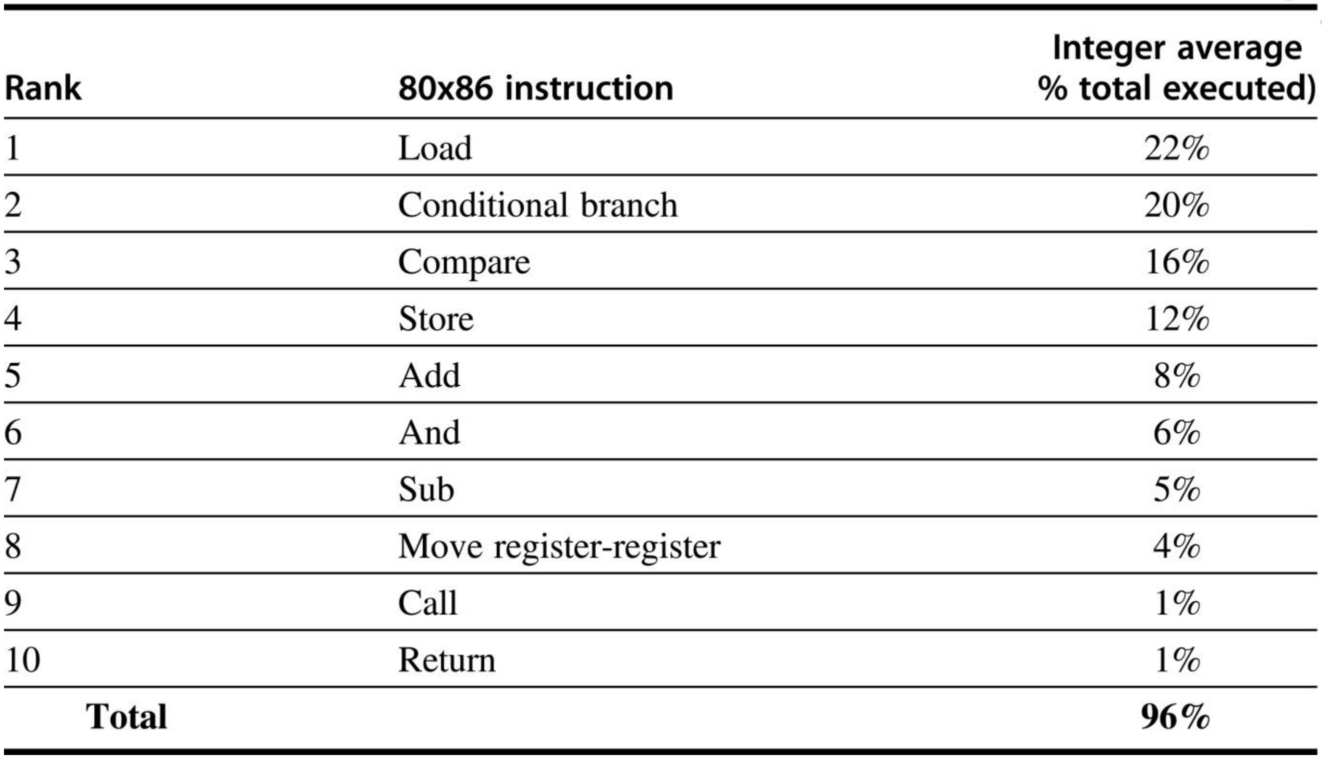
ISA OPERATIONS

ISA Operations



Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

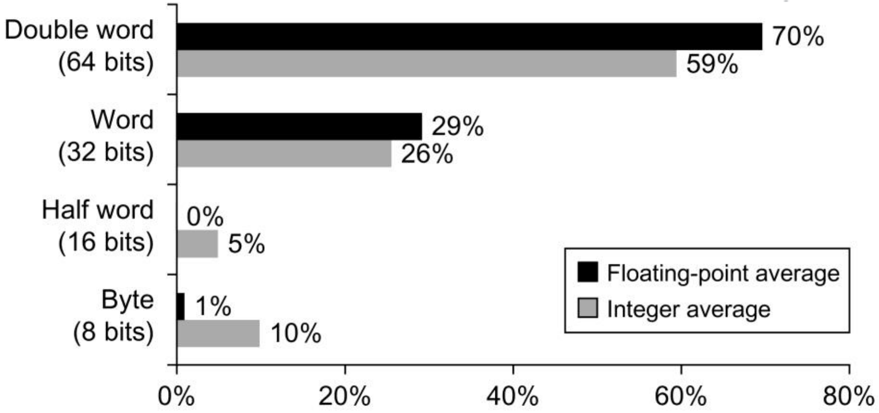
Which Operations are Most Common?



Rank	80x86 instruction	Integer average % total executed)
1	Load	22%
2	Conditional branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move register-register	4%
9	Call	1%
10	Return	1%
Total		**96%**

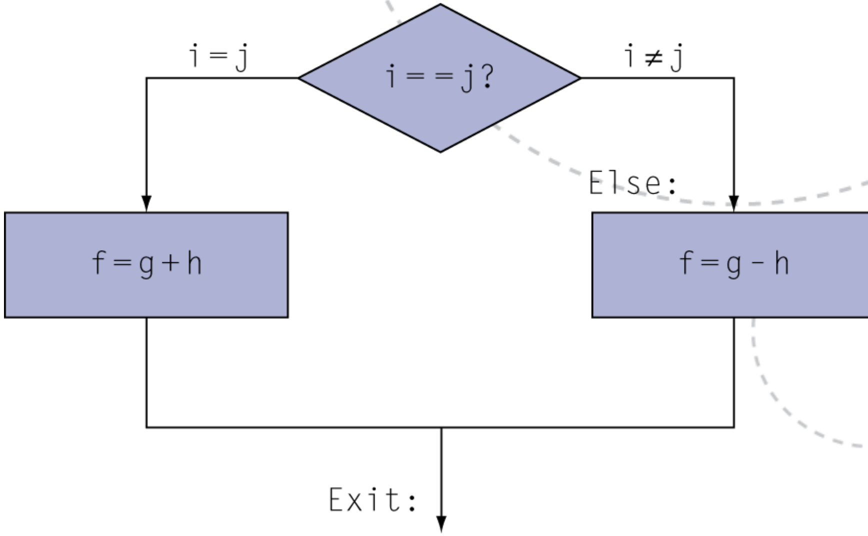
Operand Type and Size

- How to specify the type of operands:
  - Commonly: Opcode of the instruction decides
  - Also, data can be annotated with tags that choose the operation (not used anymore)
- Operand size is typically determined by the operation:
  - Integer arithmetic: 32- or 64-bit words (the word size of the machine)
  - Floating point: Single or double precision IEEE (32 or 64 bit)
  - Characters: 8 bit ASCII or 16+ bit Unicode
- Business applications: Binary code decimal
  - Floating point may not accurately represent a given fraction (e.g., 0.1)



INSTRUCTIONS FOR CONTROL FLOW

Compiling If Statements



- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, h, i, j in x1, x2, x3, x4, x5

- Compiled RISC-V code:

```
      bne x4, x5, Else
      add x1, x2, x3
      j   Exit
Else: sub x1, x2, x3
Exit: …
```

Assembler calculates addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - *i* in x3, *k* in x4, base address of the *save* array in x5

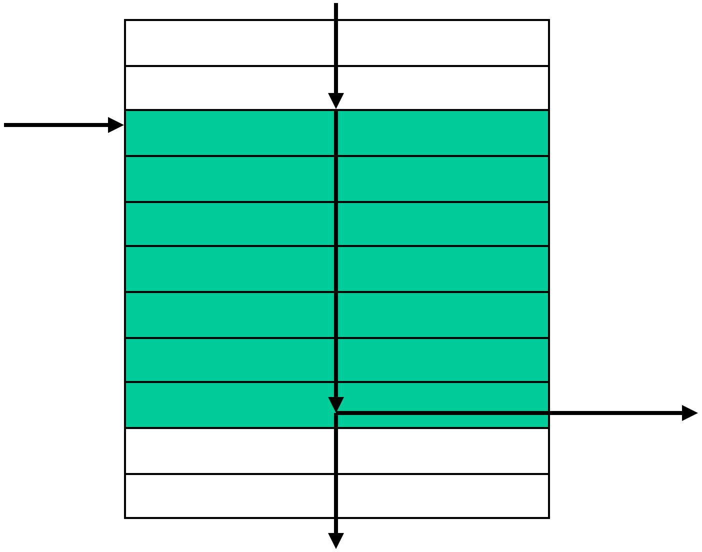
- Compiled RISC-V code:

```
Loop: sll  x6, x3, 2
      add  x6, x6, x5
      lw   x7, 0(x6)
      bne  x7, x4, Exit
      addi x3, x3, 1
      j    Loop
Exit: ...
```

Remember: Byte addresses

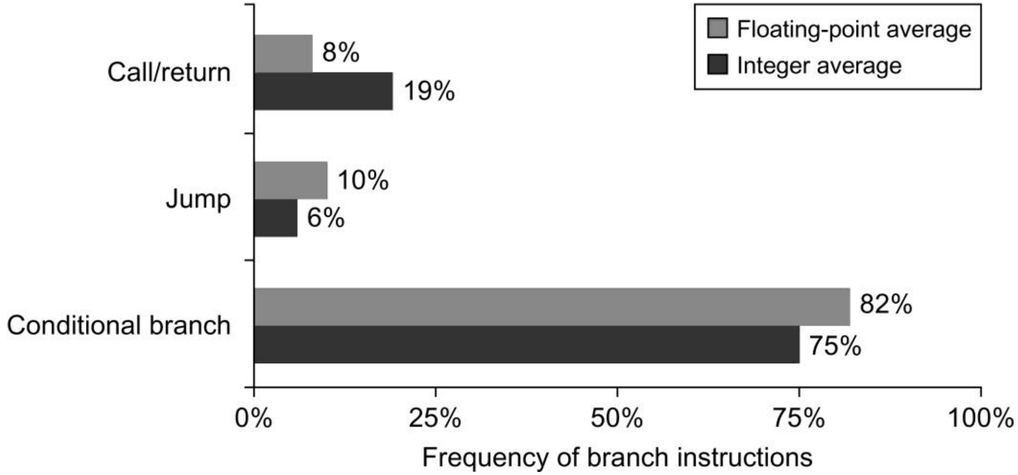
Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



Types of Control Flow

- Types of control flow
  - Conditional branches
  - Jumps (or unconditional branches)
  - Procedure calls
  - Procedure returns



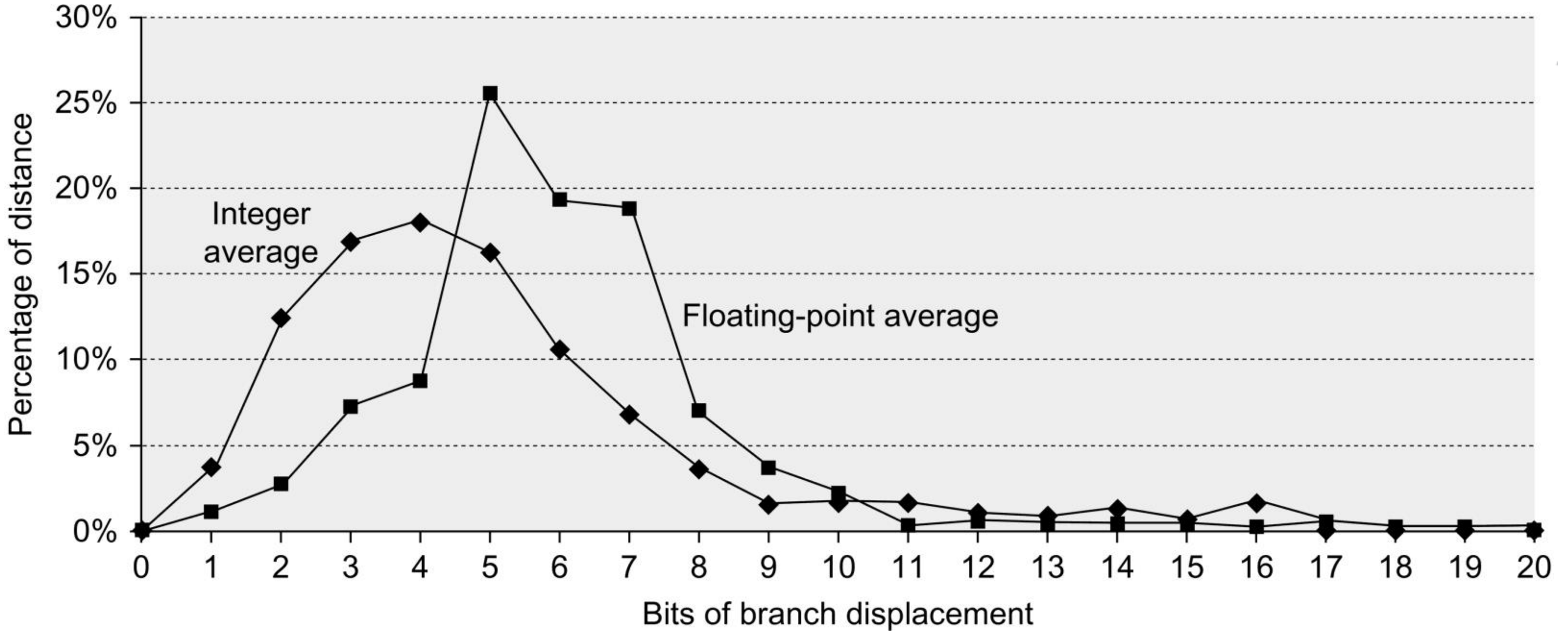
Specifying Branch Addresses

- Can be explicitly or implicitly specified

- Explicit: Commonly an offset to the PC (PC-relative addressing)
  - Advantage: The branch target is often close to the current location
  - Advantage: Branch is independent of where the program was loaded in memory (position independence property). Useful when loading and linking (especially dynamically).

- Implicit: Useful when the branch target is not known at compile time
  - Use a register or any other addressing mode to specify the location
  - Also useful in case/switch statements and for virtual functions, function pointers or dynamically shared libraries.

How Long are Branches?



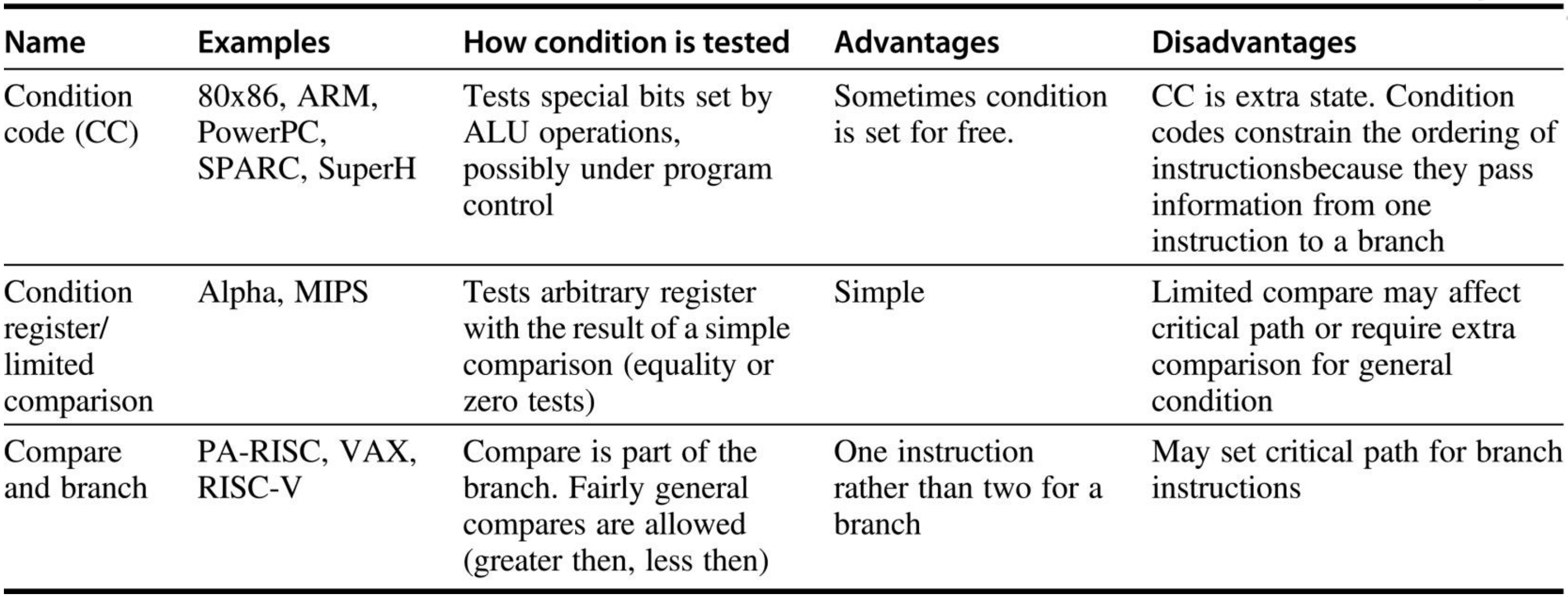
Key takeaway: Most branches are fairly short (< 10 bits)

Branching Far Away

- If branch target is too far to encode with a 13-bit signed offset you cannot use the branch instruction directly
- Why 13 bits?
  - Branches have a 12 bit immediate and the 13th bit is a 0
  - The minimum RISC-V instruction size is 16 bit, so the least significant bit is always 0

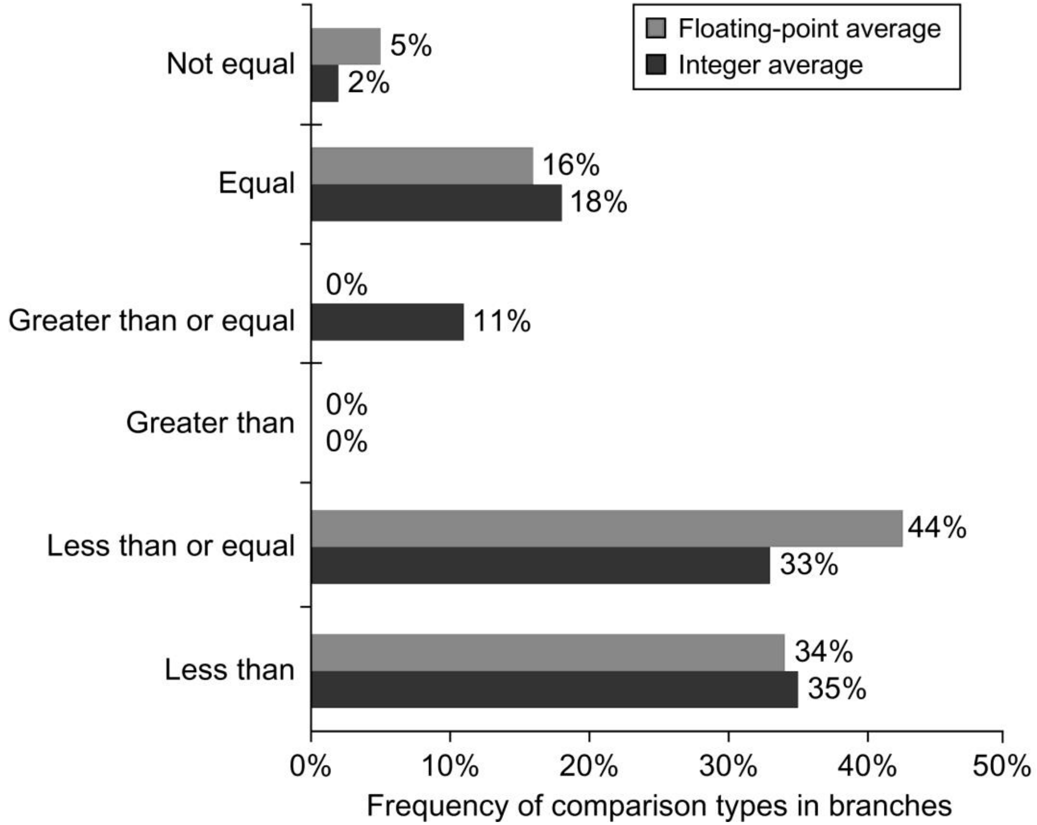
```
# Call function at any 32-bit absolute address
lui x1, <20 most significant bits>
jalr x1, x1, <12 least significant bits>

# Jump PC-relative with 32-bit offset:
auipc x1, <20 most significant bits>
jalr x1, x1, <12 least significant bits>
```

How to Specify the Branch Target?



Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructionsbecause they pass information from one instruction to a branch
Condition register/ limited comparison	Alpha, MIPS	Tests arbitrary register with the result of a simple comparison (equality or zero tests)	Simple	Limited compare may affect critical path or require extra comparison for general condition
Compare and branch	PA-RISC, VAX, RISC-V	Compare is part of the branch. Fairly general compares are allowed (greater then, less then)	One instruction rather than two for a branch	May set critical path for branch instructions

Which Comparisons are Common?



Key takeaway: Less than (or equal) dominate for this compiler and architecture

Procedure Calls

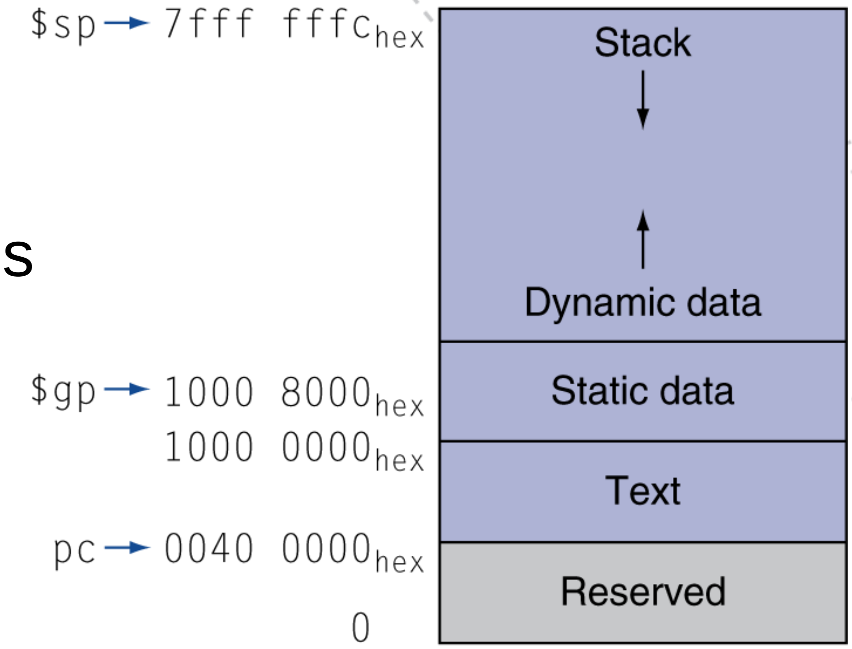
- Register state needs to be saved before calling a procedure

- Caller/callee convention
  - Specifies whether the caller or the callee should save state and which state needs to be saved.
  - Saving is done by pushing registers onto the stack
  - Specified for the architecture in the Application Binary Interface (ABI)

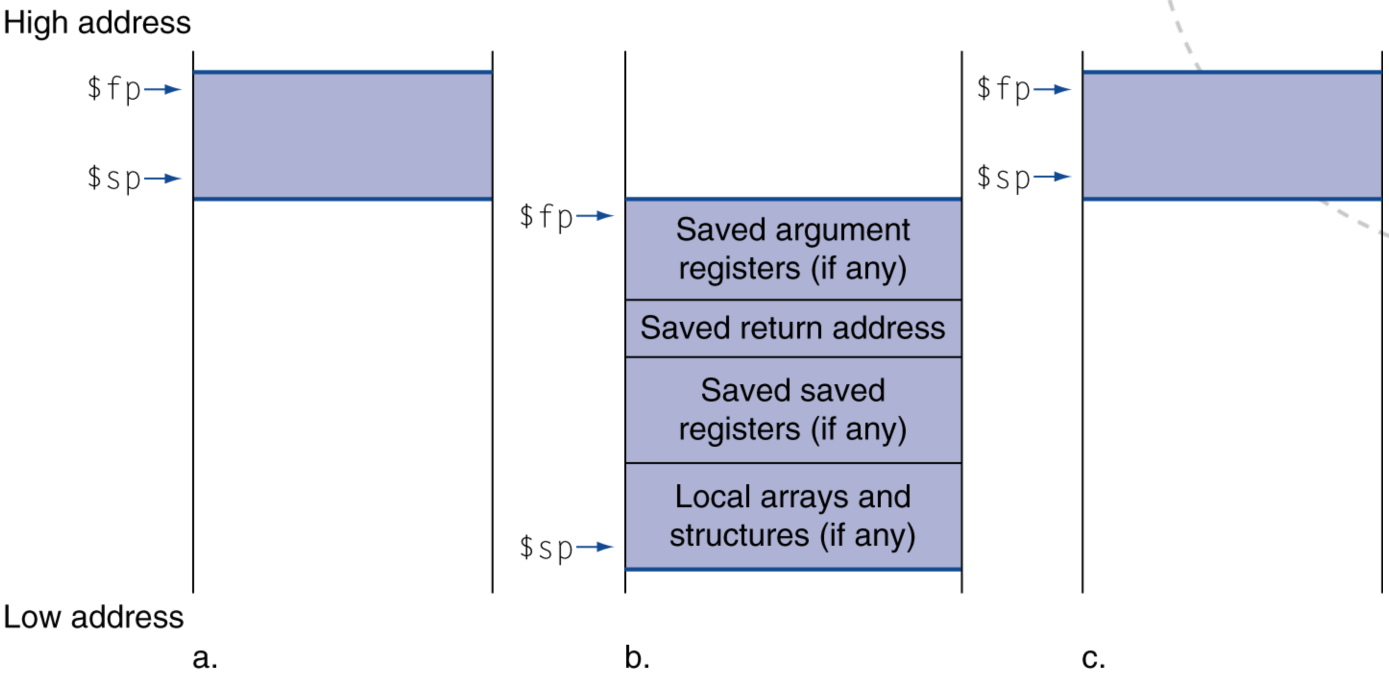
- Commonly some registers are temporaries (not saved) to reduce the amount of state copying necessary

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Local Data on the Stack


High address
$fp
$sp
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
Low address
a.
b.
c.

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Control Flow Summary

- Conditional branches:
  - Use a PC-relative displacement of at least 8 bits
- Support register-indirect and PC-relative addressing for jump instructions
  - Useful for to supporting useful features such as returns, virtual procedures, function pointers, etc.

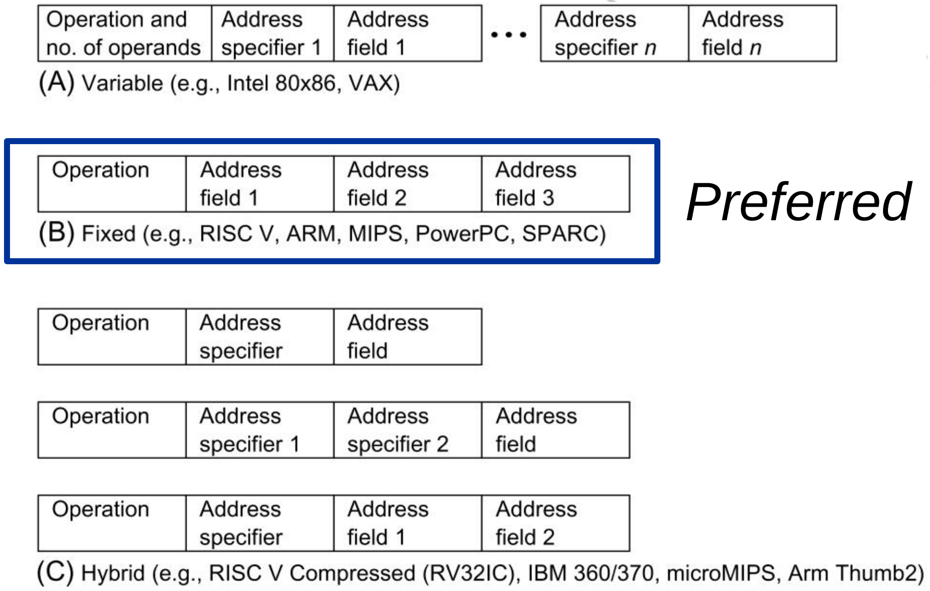
ENCODING AN ISA

ISA Observations

- We have analyzed instruction usage quite extensively
- Overall findings:
  - We are leaning towards a load-store architecture
  - Displacement, immediate and register indirect addressing modes
  - Support 8-, 16-, 32- and 64-bit integers and 32- and 64-bit floating point
  - Need instructions for:
    - Simple operations (arithmetic, load, store, etc.)
    - PC-relative conditional branches
    - Jump and link instructions for procedure calls
    - Register indirect jumps for procedure return
- Next step: Encode this into an instruction set

ISA Encoding

- Opcode specifies the operation
- Need to balance:
  - The desire to have as many registers and addressing modes as possible
  - Lower average program size favors fewer registers and fewer different addressing modes
  - Instructions should be easy to decode



Operation and no. of operands	Address specifier 1	Address field 1	...	Address specifier *n*	Address field *n*

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

Operation	Address field 1	Address field 2	Address field 3

(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)

Preferred

Operation	Address specifier	Address field

Operation	Address specifier 1	Address specifier 2	Address field

Operation	Address specifier	Address field 1	Address field 2

(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

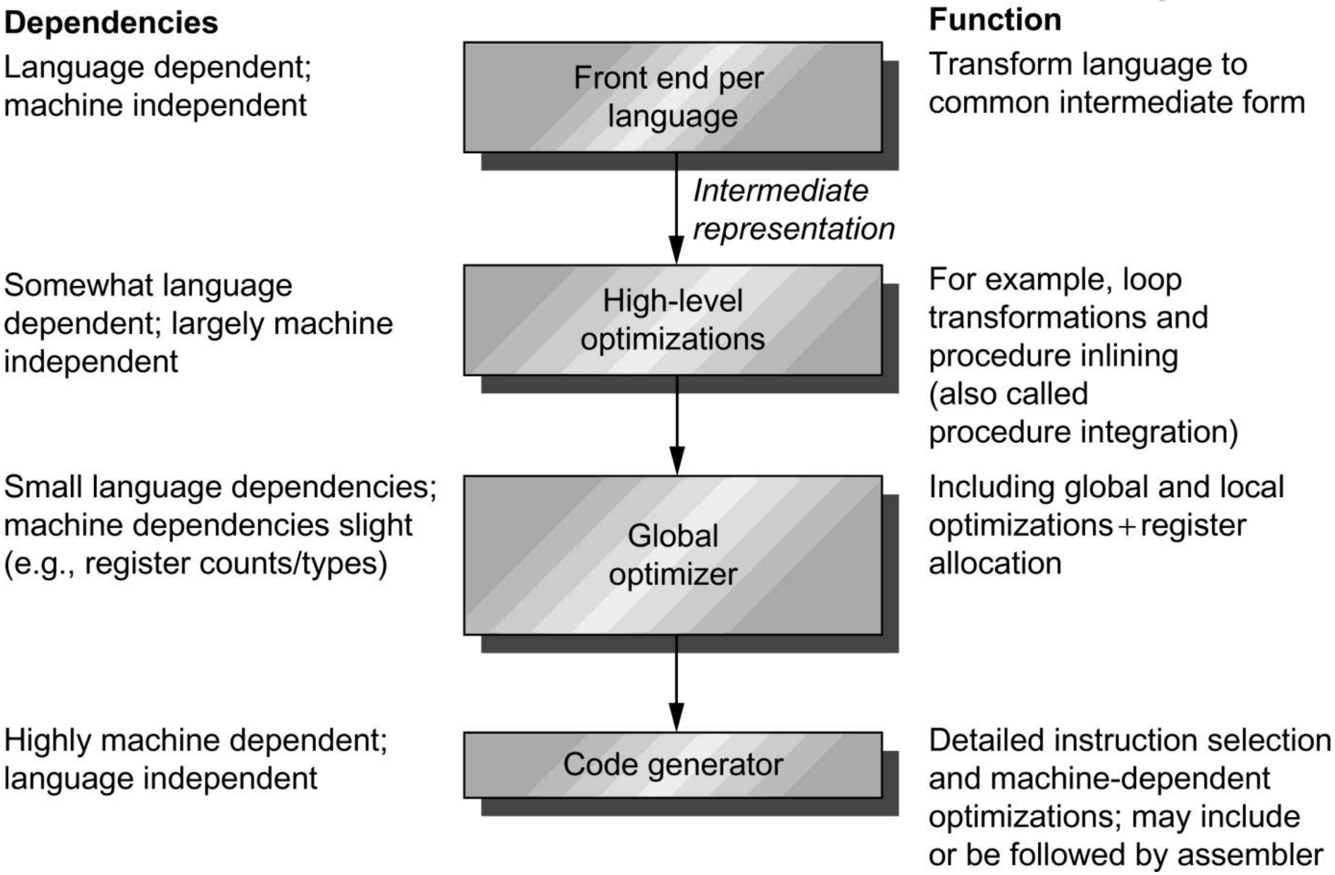
Fixed vs. variable instruction size

Reducing Code Size in RISCs

- Embedded devices have limited memory
- Requiring that all instructions are 32 bit long wastes memory
  - Common operations
  - Instructions with small immediates
- Response: Variable length instruction sets such as ARM Thumb

THE ROLE OF COMPILERS

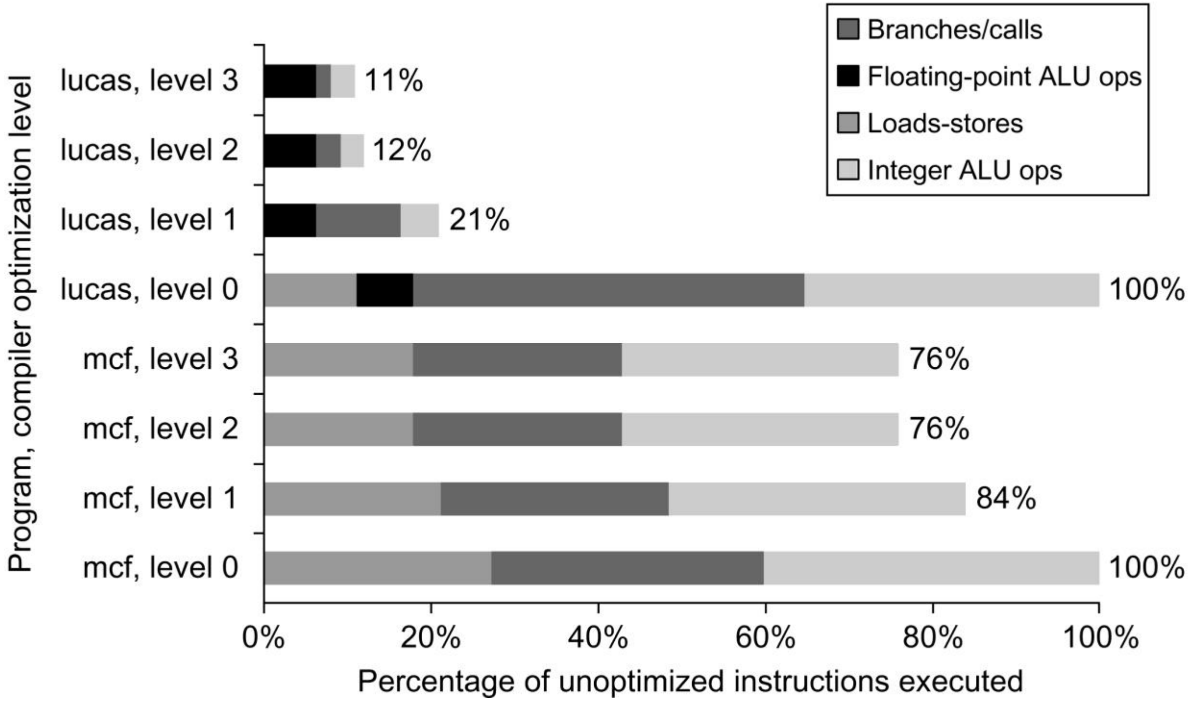
Compiler Structure



Dependencies		Function
Language dependent; machine independent	Front end per language	Transform language to common intermediate form
	Intermediate representation	
Somewhat language dependent; largely machine independent	High-level optimizations	For example, loop transformations and procedure inlining (also called procedure integration)
Small language dependencies; machine dependencies slight (e.g., register counts/types)	Global optimizer	Including global and local optimizations + register allocation
Highly machine dependent; language independent	Code generator	Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

Compilers are very complex and correctness is the key requirement

Performance Impact of Compiler Optimizations



Key takeaway: Compiler optimizations can have huge impact

How Can Architects Help Compiler Writers?

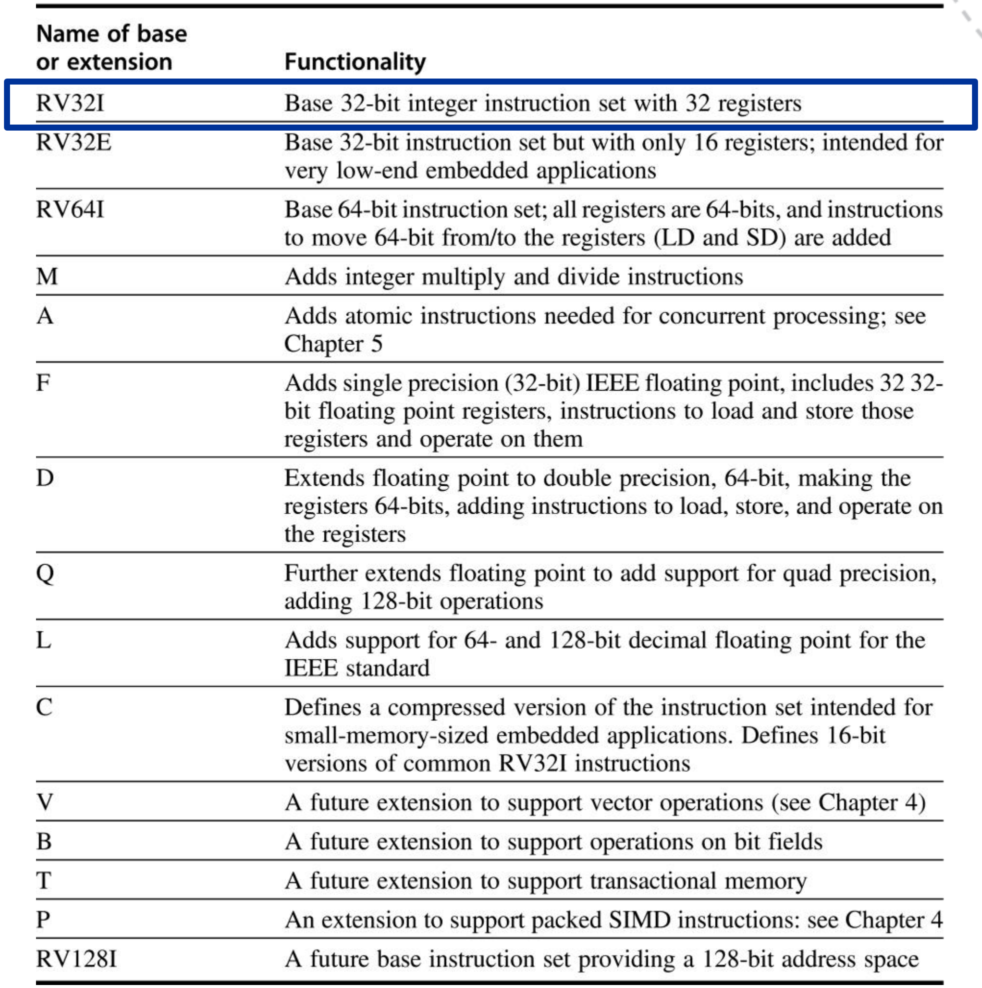
- Provide regularity
  - Ideally, ISAs should be orthogonal (aka independent). E.g., all instructions support all addressing modes.
- Provide primitives, not solutions
  - Instructions that match language constructs tend to not be used
- Simplify trade-offs between alternatives
  - The compiler writer needs to find the best instruction sequence to use in a given case
- Provide instructions that bind quantities known at compile time as constants
  - No need to recompute stuff that we already know

THE RISC-V ISA

RISC-V Choices

- Ties together the insights from each aspect of ISA design we have covered
- Overall:
  - General-purpose register model and load-store architecture
  - Supports displacement, immediate and register-indirect addressing with appropriate displacements
  - Supports 8-, 16-, 32- and 64-bit integers and 32- and 64-bit floating point data types
  - Focus on the simple, dominating instructions load, store, add, subtract, move register-register and shift
  - Branching and compare according to the analysis
  - At least 16 (preferably 32) registers. Orthogonal, minimalist ISA.

RISC-V Dialects

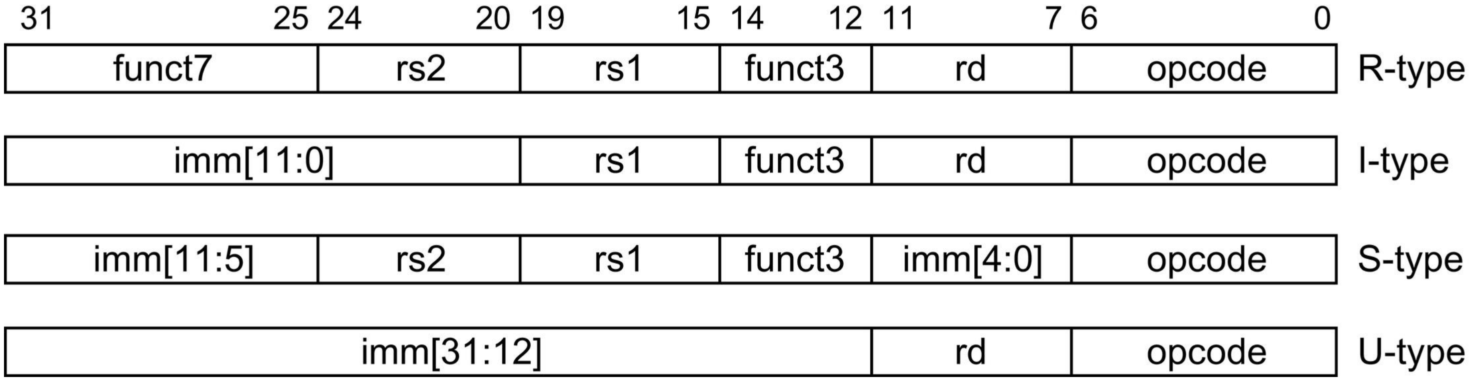


Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

Our focus in TDT4255

ISA Zoo?

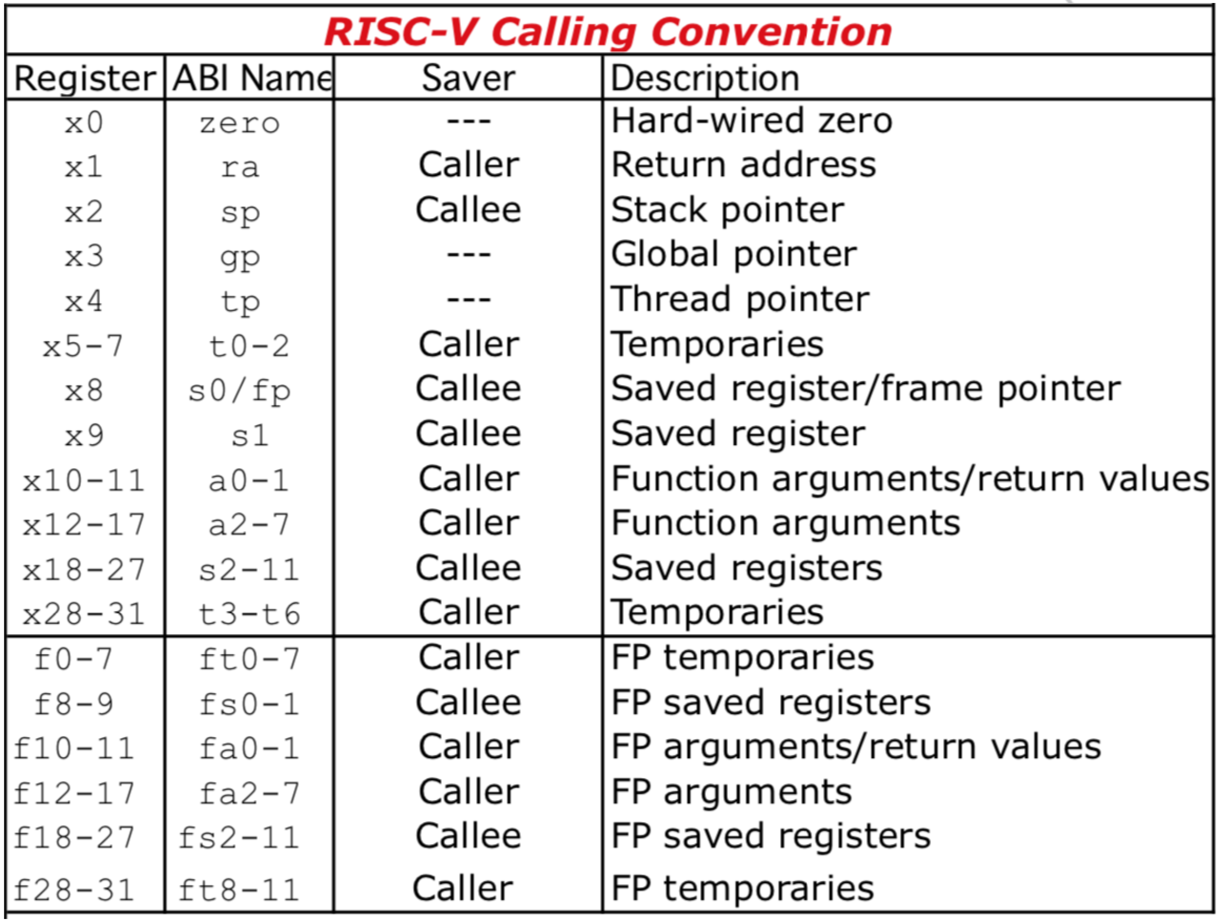
RISC-V Operand Types



31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

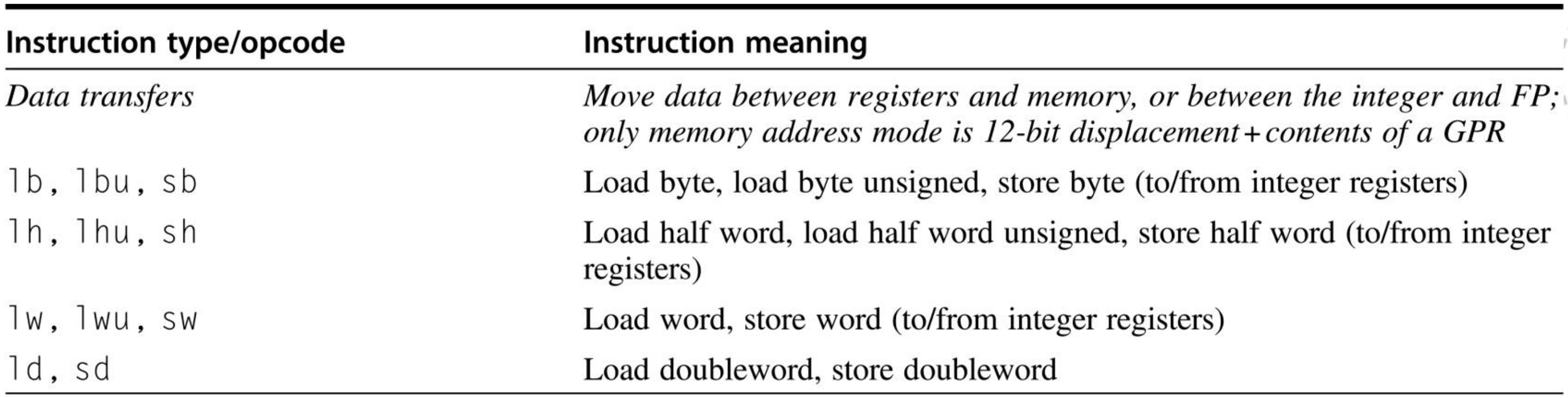
RISC-V Registers



RISC-V Calling Convention

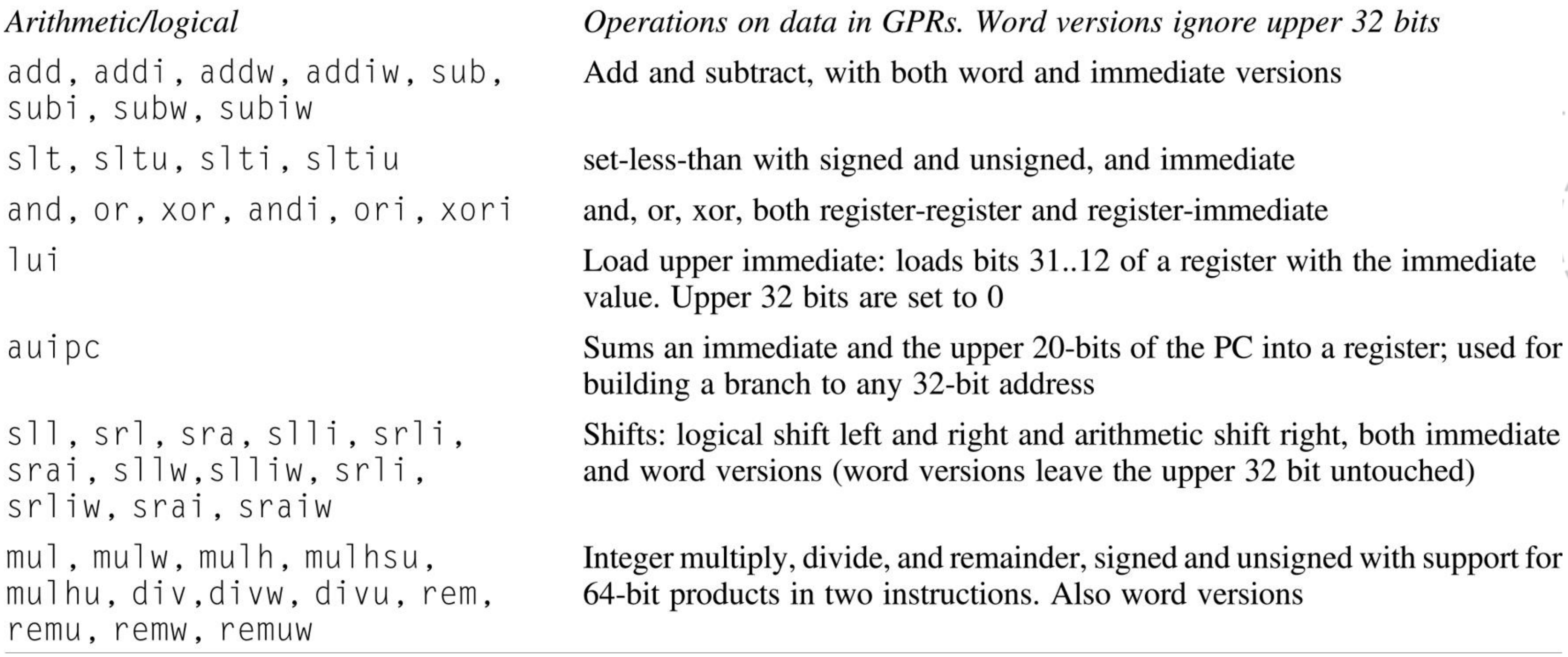
Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

RISC-V Data Transfer Instructions



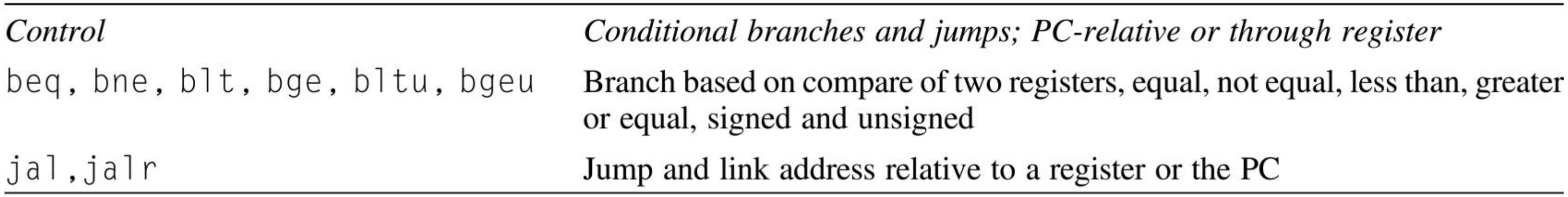
Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement + contents of a GPR*
`lb, lbu, sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh, lhu, sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw, lwu, sw`	Load word, store word (to/from integer registers)
`ld, sd`	Load doubleword, store doubleword

RISC-V ALU Instructions



Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
add, addi, addw, addiw, sub, subi, subw, subiw	Add and subtract, with both word and immediate versions
slt, sltu, slti, sltiu	set-less-than with signed and unsigned, and immediate
and, or, xor, andi, ori, xori	and, or, xor, both register-register and register-immediate
lui	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
auipc	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
sll, srl, sra, slli, srli, srai, sllw,slliw, srli, srliw, srai, sraiw	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
mul, mulw, mulh, mulhsu, mulhu, div,divw, divu, rem, remu, remw, remuw	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions

RISC-V Control Instructions



Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
`jal,jalr`	Jump and link address relative to a register or the PC

What about jump (j)?

j TARGET

Assembler rewrites

jal x0, TARGET

Explanation:
Link value is written to x0, but x0 is hardwired to 0

SUMMARY

ISA Lecture Summary

- ISA design is a good example of the quantitative approach to computer design
  - Identify a bunch of representative programs
  - Measure key trade-offs to identify what matters (the common case)
  - Understand how decisions impact the other systems in the stack (in this case mostly compilers)
  - Design an architecture that satisfies all constraints adequately

- The RISC-V ISA is the conclusion of a 30+ year trial-and-error approach
  - Possibly disruptive due to the open-source definition
  - Technological superiority does not seem to matter (and it is an open question if RISC-V turns out to be so)