ELEC3608 Computer Architecture

Instruction Set Architecture

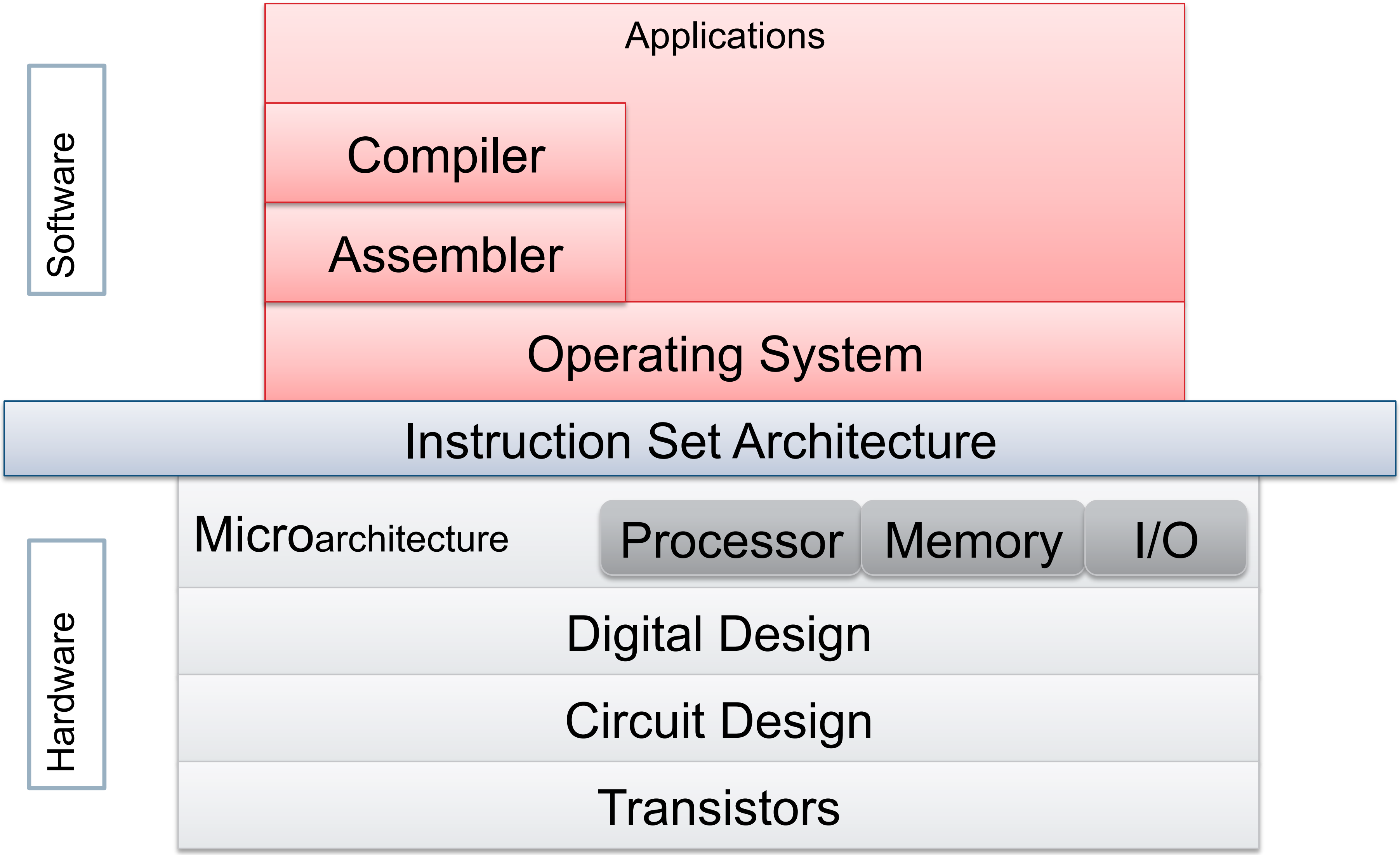
Philip H.W. Leong
School of Electrical and Information Engineering,
The University of Sydney

Introduction

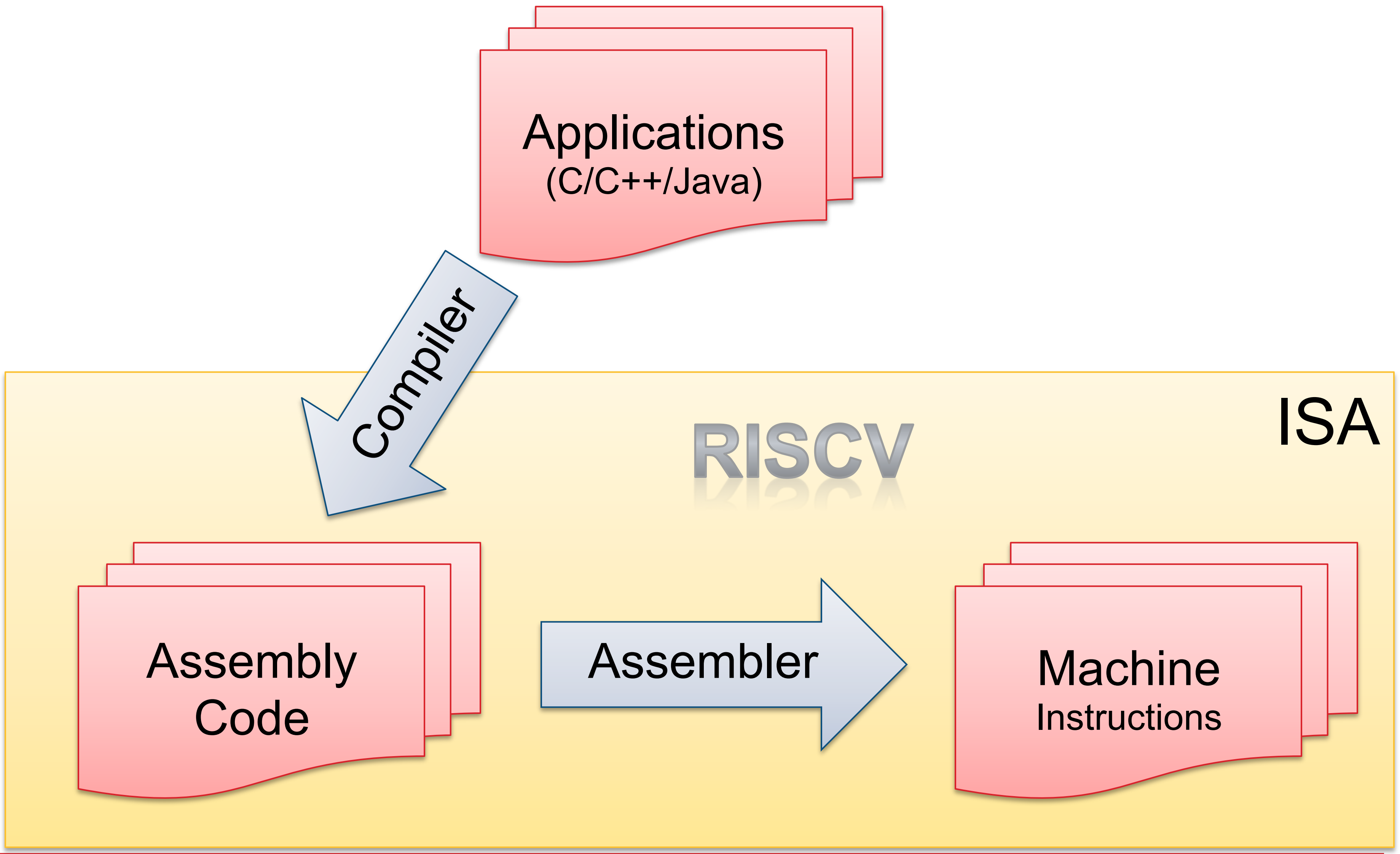
Introduction

- Instruction set architecture defines the user observable behavior of a processor
  - A contract between hardware and software
- Usually includes:
  - Observable state of a processor
  - A set of machine instructions
  - Semantics of the instruction and processor execution

Computer Architecture: HW/SW Interface



Review: Compiling Applications



Note about learning an ISA

- By learning how ONE ISA is designed, you learn how various architectural tradeoffs are performed.
- NOT to memorize details of a single ISA

- The subset of RISC-V covered in class is very similar to MIPS:
  - More examples in COD
  - Will use MIPS software for simulation in some cases

RISC-V ISA Version 2.0

RISC-V ISA

- New RISC design from UC Berkeley
- Realistic & complete ISA, but open & small
- Not over-architected for a certain implementation style
- 32-bit and 64-bit address space variants
  - RV32 and RV64
- Designed to be extensible
  - High-performance multiprocessing
  - Efficient instruction encoding for embedded system
  - etc
- Easy to subset/extend for education/research
- Complete compilation toolchain
- FPGA and ASIC designs

RISC-V Foundation

http://www.riscv.org

RISC-V ISA 2.0 (draft)

- A revised version to the original definition
- Divides into an integer base ISA + extension
  - Floating point, etc
- Revised instruction coding
  - Make hardware design easier
- Added 128-bit support
- Much more…
- We will be using RISC-V ISA 2.0 this semester:
  - Things may break
  - RISC-V base integer ISA is similar to MIPS in textbook
  - Will try cross reference between RISCV and MIPS

RISC-V ISA Overview

- The riscv processor has a **load-store architecture**
  - Most instructions operate on values directly to and from the **register file**
  - Dedicated **load/store instructions** to transfer data between register file and main memory
- RV32 has a 32-bit **word length**
  - Fixed instruction length of 32 bits
  - A data word is 32 bits wide
  - RISCV-V ISA designed to allow variable length instruction code and instruction extension
    - Not covered in this course

RV32 Processor State – Base Integer Design

- Register file with **32** 32-bit general-purpose registers
- Labeled x0 – x31
  - x0 always contains the value 0
- Application Binary Interface (ABI) names used in assembler
  - t0 – t6 for temp variables
  - s0 – s11 for saved variables, etc
- Additional **Program Counter** (PC) register
  - Contains the memory address of the current executing instruction

Register	ABI Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5-7	t0-2
x8	s0/fp
x9	s1
x10-17	a0-7
x18-27	s2-11
x28-31	t3-6

XLEN=32 in RV32

https://blog.riscv.org/2015/01/announcing-the-risc-v-gcc-4-9-port-and-new-abi/

Integer Register-Register Operations

- Basic arithmetic and bit operations:
  - Arithmetic: ADD, SUB
  - Bit Operations: AND, OR, XOR, SLL, SRL, SRA
  - Comparison: SLT, SLTU
- Take two **source** register inputs (denoted rs1, rs2) and produce one **destination** register output (denoted rd).

ADD/SUB

Assembler	Semantics
ADD rd, rs1, rs2	rd ← rs1 + rs2
SUB rd, rs1, rs2	rd ← rs1 - rs2

- 3 operands:
  - 2 sources and 1 destination
- Add/sub the values of the 2 source registers (rs1, rs2) and store the result in destination register (rd)
- Discard overflow
- Lower 32 bits are written

ADD/SUB Example

› C code:

```
y = (e + f) – (g + h);
```

› Assume e, f, g, h, y stored in a0, a1, a2, a3, a4 respectively.

› RISCV code:

```
ADD t0, a0, a1
ADD t1, a2, a3
SUB a4, t0, t1
```

Bit Operations

Assembler	Semantics
AND rd, rs1, rs2	rd ← rs1 AND rs2
OR rd, rs1, rs2	rd ← rs1 OR rs2
XOR rd, rs1, rs2	rd ← rs1 XOR rs2

› Similar semantics to ADD/SUB instructions

› Bit operations useful for masking values

› E.g.:

```
#  t0 ← 0xABCD0123
#  t1 ← 0x000F0000
AND t2, t0, t1
#  t2 contains 0x000D0000
```

Quick Quiz

```
#  t0 ← 0x00C00005
#  t1 ← 0xFFFFFFFF
XOR t2, t0, t1
```

What is the value stored in t2?

Instruction Encoding

RISC-V Instruction Encoding

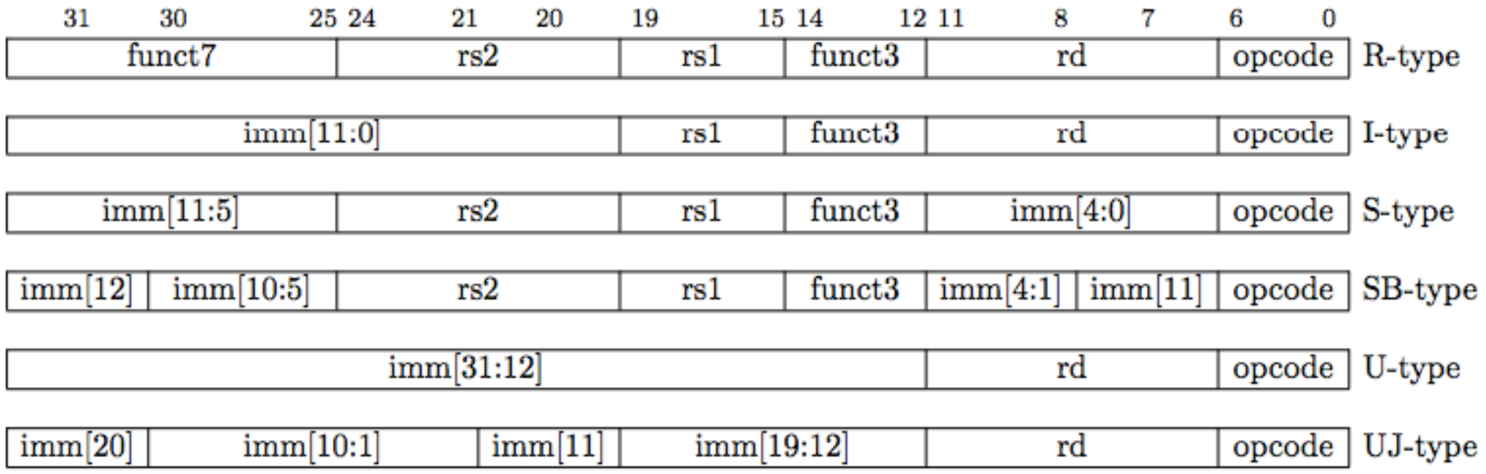
› Each supported instruction in the ISA must be uniquely encoded to describe its function

› Any unique encoding works, but good encoding may optimize:

- Regularity; Simple hardware decode; Instruction length; Extensibility

› In RV32 6 general types of instructions: R, I, S, SB, U, UJ

- Favors hardware regularity over human readability
- Each instruction is 32-bit wide



31	30 25	24 21	20	19 15	14 12	11 8	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[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	SB-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	UJ-type

R-type Instruction Encoding

- All the simple integer register-register operations mentioned so far are encoded as R-type instructions

R-Type format

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

- Instructions within the same type shares similar encoding
- Differentiate only by the funct3 and funct7 fields

Quick Quiz

› If the size of register file is increased to include 64 registers, how many bits are needed to encode rs1?

› How would the above change affect the number of instructions representable?

Immediate Operations

- Very often programs need to operate on (small) numerical constants in the code
  - e.g.: “a = b + 3”
- RV32 includes register-immediate instructions for these cases
- Small constant embedded within the instruction.
  - Arithmetic: ADDI
  - Bit Operations: ANDI, ORI, XORI
  - Comparison: SLTI, SLTIU
- Take one **source** register inputs (rs1) and produce one **destination** register output (rd).

I-type Instruction Encoding

I-Type format

› Most register-immediate operations are encoded using the I-type format

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

› NOTE: since the constant is encoded *within* the instruction, size of immediate must be < 32 bits

› In RV32, I-type instruction, immediate constants are encoded using 12 bits

› Note that bit locations of rs1, funct3, rd and opcode are the same as in R-type

- Simplicity ➔ high performance

ADDI Example

› C code:

```
y = y + e + 1
```

› Assume e, y stored in a0, a1 respectively.

› RISCV code:

```
ADD  t0, a1, a0
ADDI a1, t0, 1
```

Why is there no SUBI instruction… ?

Number Systems

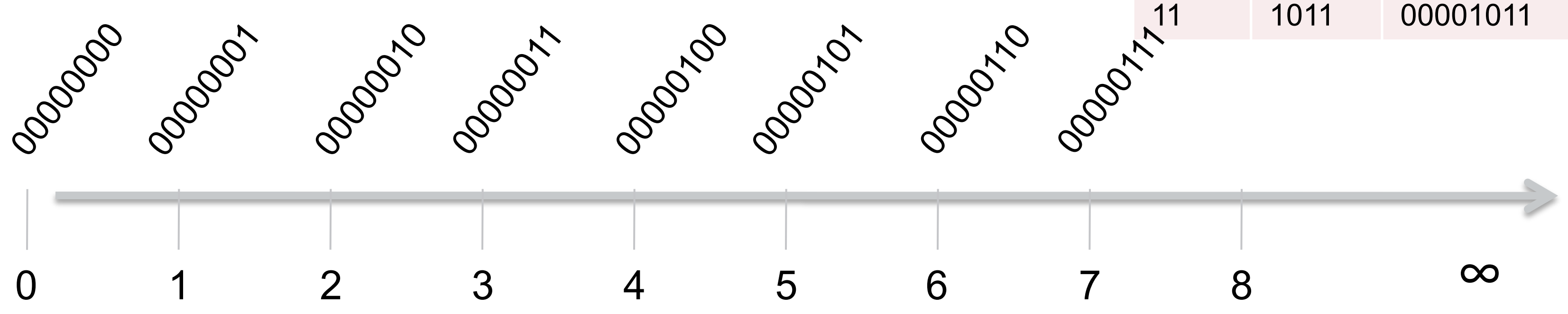
Unsigned Numbers

- Represent non-negative binary numbers (0, 1, 2, 3, …) using their natural binary representations
- An n-bit bitstring can represent numbers in

$$[0, 2^n - 1]$$

- Represents **equally spaced** integers on the number line

Value	Binary	Bitstring (8-bit)
0	0	00000000
1	1	00000001
2	10	00000010
3	11	00000011
4	100	00000100
5	101	00000101
6	110	00000110
7	111	00000111
8	1000	00001000
9	1001	00001001
10	1010	00001010
11	1011	00001011



Two’s Complement (1)

- Negative numbers are represented by the 2’s complement of the absolute value of that number
  - 2’s complement of an *n*-bit number *v* is the value $2^n - v$
- For example, to represent value -3 using a 4-bit bitstring:
  - 3 $\rightarrow$ “0011”
  - $2^4 - 3 = 16-3 = 13$
  - $\therefore$ -3 $\rightarrow$ “1101”

Two's complement (2)

- 2's complement of a number can be obtained by "adding 1 to the 1's complement of the number"

Original	23	00010111
1's complement		11101000
2's complement	-23	11101001

- The value of a bitstring $\{b_{n-1}b_{n-2}\cdots b_0\}$ in 2's complement can be calculated as:

$$-2^{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

- Range: $[-2^{n-1}, 2^{n-1} - 1]$
  - Note that is asymmetric

Quick Quiz

› What is the binary representation of 13?

$\boxed{1101_2}$

› What is the binary representation of -13?

1	0010
2	0011
3	Cannot be represented
4	Cannot be determined based on given information

Quick Quiz

› What are the binary representations of 13 and -13 as 8-bit integers?

1 13: 00001101 -13: 00000011

2 13: 00001101 -13: 11110011

› What are the binary representations of 13 and -13 as 12-bit integers?

1 13: 000000001101 -13: 000011110011

2 13: 000000001101 -13: 111111110011

Sign Extension

› The sign of an integer can only be known with the width of integer specified

› In 2's complement representations, the most significant bit (MSB) of an integer denotes its sign:

- MSB=1 ➔ negative
- MSB=0 ➔ non-negative

› To cast an integer value to a representation with more bits, it must be **sign-extended** to preserve its sign:

- If it is negative number, fill the additional bits with 1
- If it is non-negative, fill the additional bits with 0

ADDI with negative numbers

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

- Subtraction with immediate values can be achieved by using negative immediate with ADDI
- Recall that immediate constants are 12 bits long in all I-type instructions
  - Range: [-2^{11}, $2^{11}-1$]
- In general, instructions must clearly specify in its semantics if it is treating the value as signed or unsigned numbers
  - E.g. SLTI vs SLTIU

Load Upper Immediate

U-Type format

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI

- Load upper bits of immediate constant
- Form constants of any values in combination with ADDI, etc
- No need for sign extension
- NOTE: opcode and rd in the same location as most other instructions

Memory Operations

Memory Operations

- All program data + instruction originally stored in main memory
  - Array, structure, user data, string, etc
- Dedicated instruction to load/store data between register file and main memory
- NOTE: registers are much faster than main memory
  - Limited registers availability (32 registers vs. 4GB of memory)
  - Compiler must use registers wisely for best performance

RV32 Memory Model

- Memory is byte-addressable
  - Each address corresponds to 1 byte of data
- Native data types:
  - Word: 32 bit (`int`)
  - Half-word: 16 bit (`short`)
  - Byte: 8 bit (`char`)
- RV32 has a **Little-Endian** byte addressing scheme
  - The least significant byte is at the lowest address

Endianness

```
int A[20]
A[0]=0xABCD0123
A[1]=0xFACECAFE
```

	Big Endian	Little Endian
0x000A0007	FE	FA
0x000A0006	CA	CE
0x000A0005	CE	CA
0x000A0004	FA	FE
0x000A0003	23	AB
0x000A0002	01	CD
0x000A0001	CD	01
0x000A0000	AB	23

Word Alignment

- Words are **naturally aligned** at 4-byte boundary
- Half-words naturally aligned at 2-byte boundary
- RV32I allows non-aligned access
  - **Not common** in most processors
  - See ISA spec

	aligned	aligned	Not aligned
0x000A0007		AB	
0x000A0006		CD	
0x000A0005		01	AB
0x000A0004		23	CD
0x000A0003	AB		01
0x000A0002	CD		23
0x000A0001	01		
0x000A0000	23		

Memory Operations

I	imm[11:0]	rs1	f3	rd	opcode
	offset[11:0]	base	width	dest	LOAD

S	imm[11:5]	rs2	rs1	f3	imm[4:0]	opcode
	offset[11:5]	src	base	width	offset[4:0]	STORE

Load:

(dest) $\leftarrow$ M[(base) + offset]

Store:

M[(base) + offset] $\leftarrow$ (src)

- Immediate is sign extended
- LW/SW – Word
- LH/SH – Half Word
- LB/SB – Byte

Memory Operations – Load

```
int y = e + f[3];
```

- Assume e, f, y stored in a0, a1, a2 respectively.
- RISCV code:

```
lw  t0, 12(a1)   # 12 = 3 x 4
ADD a2, a0, t0
```

- LW, LH, LB instructions load word, half-word and byte from memory

Memory Operations – Store

```
int f[7] = f[6] + f[5];
```

- Assume f is stored in a0.
- RISCV code:

```
lw  t0, 20(a0)   # 20 = 5 x 4
lw  t1, 24(a0)
ADD t0, t0, t1
sw  t0, 28(a0)
```

- SW, SH, SB instructions store word, half-word and byte to memory

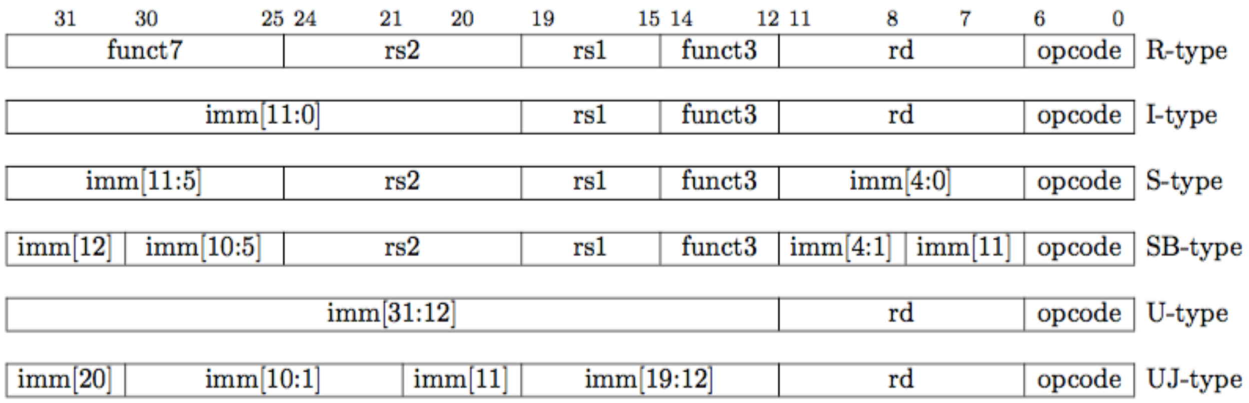
Load/Store Instruction Encoding

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

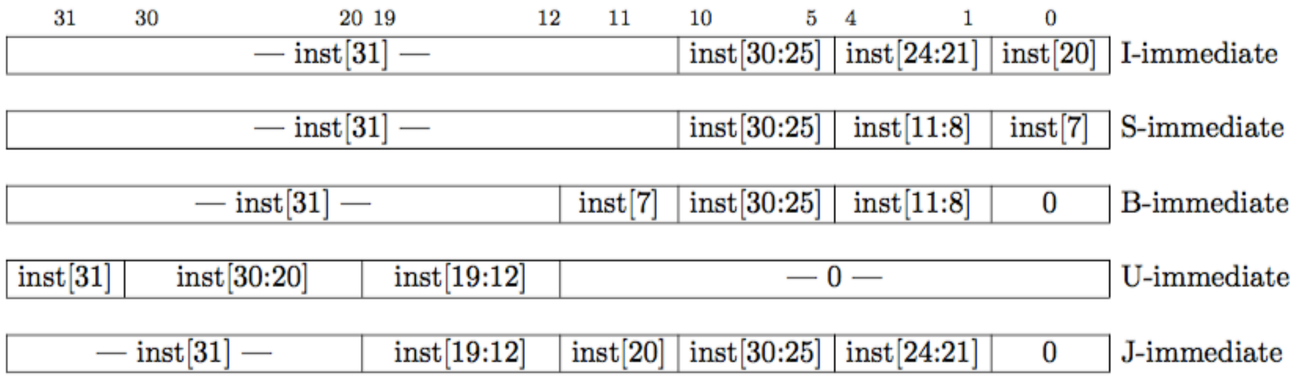
- LOAD instructions are encoded as I-type
- STORE instructions are encoded as S-type
- funct3 encodes the length (W, H, B)
- 12-bit offset is sign extended and added to base address in rs1
- Note the bit location of imm[11:5] and imm[4:0] in S-type

Immediate Formats



31	30 25	24 21	20	19 15	14 12	11 8	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[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	SB-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	UJ-type

Figure 2.3: RISC-V base instruction formats showing immediate variants.



31	30 20	19 12	11	10 5	4 1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

Branch Operations

Branch Operations

- Normally programs execute sequentially
  - The next instruction to fetch is normally at location PC + 4
- Decision making constructs requires fetching instruction from run-time determined location
  - if-the-else, do…while, goto, switch, etc
- RISCV has two types of control transfer instructions:
  - **Unconditional jump**
  - **Conditional branch**

THE UNIVERSITY OF SYDNEY

Conditional Branches

SB-Type format

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

› 12-bit branch immediate encoded as *signed offset in multiples of 2*

- E.g. offset[12:1] = 0x00C ➔ jumps to PC + 12*2 = PC + 24

Assembler	**Semantics**
BEQ rs1, rs2, dest	if rs1=rs2 then branch dest
BNE rs1, rs2, dest	If rs1≠rs2 then branch dest
BLT rs1, rs2, dest	If rs1<rs2 then branch dest
BGE rs1, rs2, dest	If rs1≥rs2 then branch dest
BLTU, BGEU	Unsigned comparison of BLT, BGE

Unconditional Jump-and-Link

31	30 — 21	20	19 — 12	11 — 7	6 — 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	offset[20:1]			dest	JAL

UJ-Type format

› Semantics:
  - Unconditional jump to PC+offset*2
  - Store return address (PC+4) in rd

Jump

› J-immediate encodes a signed offset in multiples of 2 bytes
  - 20 bits ➔ ±1MiB range

› Jump relative to PC

› Plain jump (J assembler) use rd=x0

… and-Link …

```
v = 9
r = sqrt(v)
r = r + 27
```

- To facilitate function calls
  - Call to function and return to the next instruciton
- Convention is to use x1 (ra) as return register

```
0xA000 addi a0, zero, 9
0xA004 jal  ra, sqrt
0xA008 addi a0, a0, 27
.
.
0xBC04 <. . .>
0xBC08 <. . .>
0xBC0C <. . .>
0xBC10 return
```

+ 0x1C00

- return to 0xA008
- 0xA008 stored in register ra from the jal instruction

Unconditional JALR

I-Type format

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

› Similar to JAL, except destination is relative to the value of rs1

› Semantics:

- Unconditional jump to offset + (rs1)
- Store return address (PC+4) in rd

› Note

- the offset is not offset by multiple of 2
- Entirely same as an I-type format instruction

JAL + JALR for function calls

```
0xA000 addi a0, zero, 9
0xA004 jal  ra, 0x1C00
0xA008 addi a0, a0, 27
.
.
0xBC04 <. . .>
0xBC08 <. . .>
0xBC0C <. . .>
0xBC10 jalr zero, ra
```

If-Then-Else Conversion

```
if (a >= b) {
        c = a – b
} else {
        c = a + b
}
```

```
          blt a1, a0, truepart
          add    a2, a1, a0
          j   exit
truepart: sub a2, a1, a0
exit    :  <…>
```

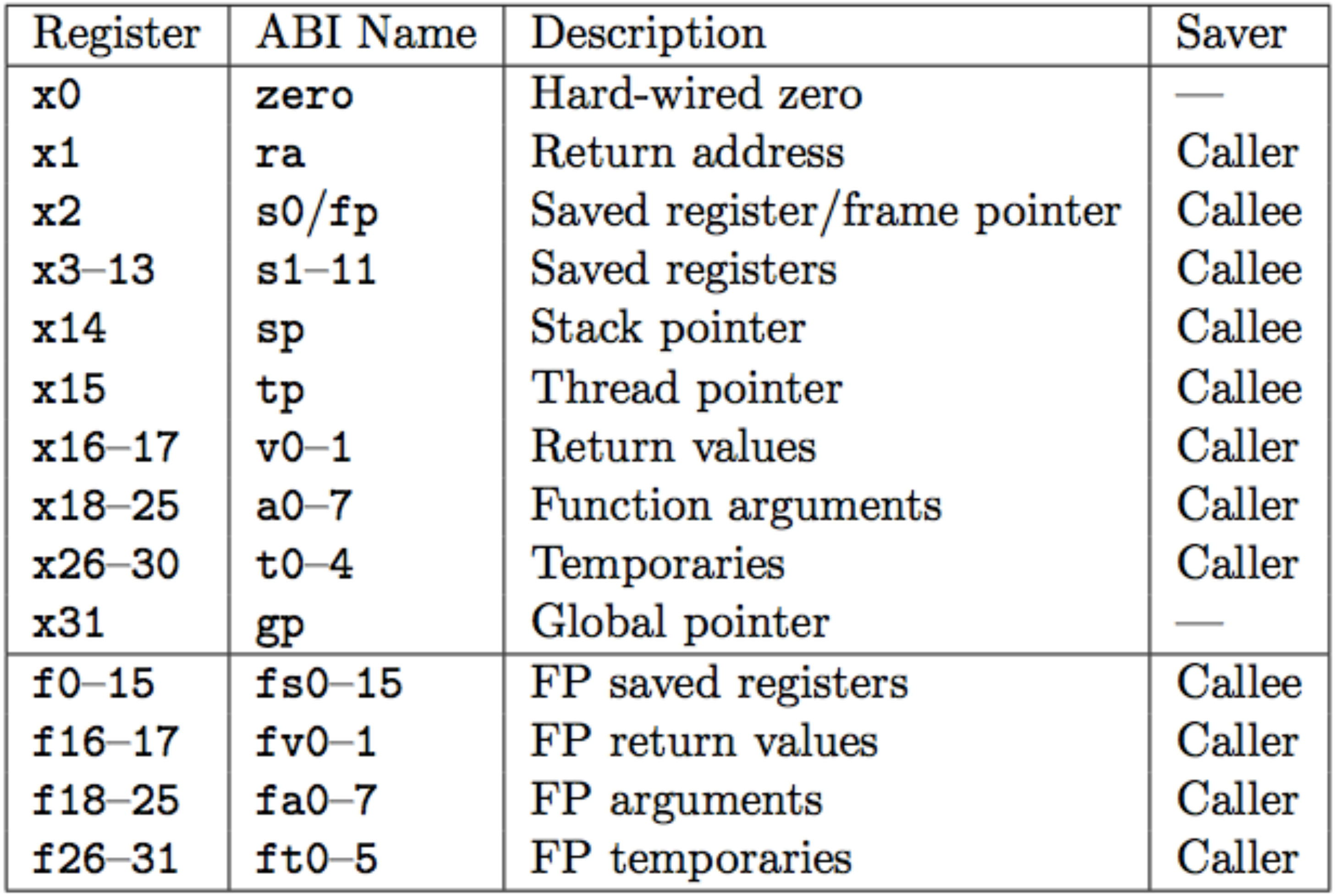
Procedure Calls

Procedure Calling

› Steps required

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure’s operations
5. Place result in register for caller
6. Return to place of call

Register Usage



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	s0/fp	Saved register/frame pointer	Callee
x3–13	s1–11	Saved registers	Callee
x14	sp	Stack pointer	Callee
x15	tp	Thread pointer	Callee
x16–17	v0–1	Return values	Caller
x18–25	a0–7	Function arguments	Caller
x26–30	t0–4	Temporaries	Caller
x31	gp	Global pointer	—
f0–15	fs0–15	FP saved registers	Callee
f16–17	fv0–1	FP return values	Caller
f18–25	fa0–7	FP arguments	Caller
f26–31	ft0–5	FP temporaries	Caller

Leaf Procedure Example

› C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, …, j in a0, …, a3
- f in s0 (hence, need to save s0 on stack)
- Result in a0

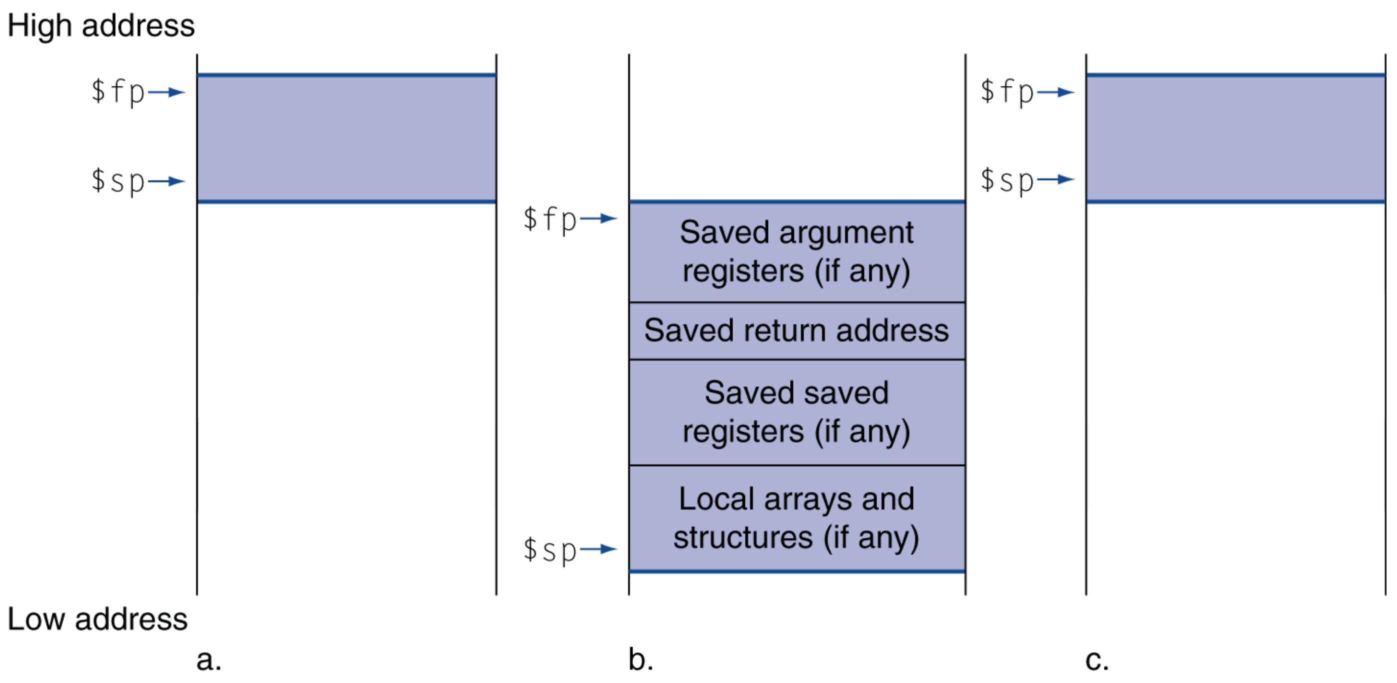
THE UNIVERSITY OF SYDNEY

Leaf Procedure Example

› RISCV code:

```
leaf_example:
    addi sp, sp, -4       Save s0 on stack
    sw   s0, 0(sp)
    add  t0, a0, a1       Procedure body
    add  t1, a2, a3
    sub  s0, t0, t1
    add  a0, s0, zero     Result
    lw   s0, 0(sp)
    addi sp, sp, 4        Restore s0
    jalr   zero, ra       Return
```

Local Data on the Stack

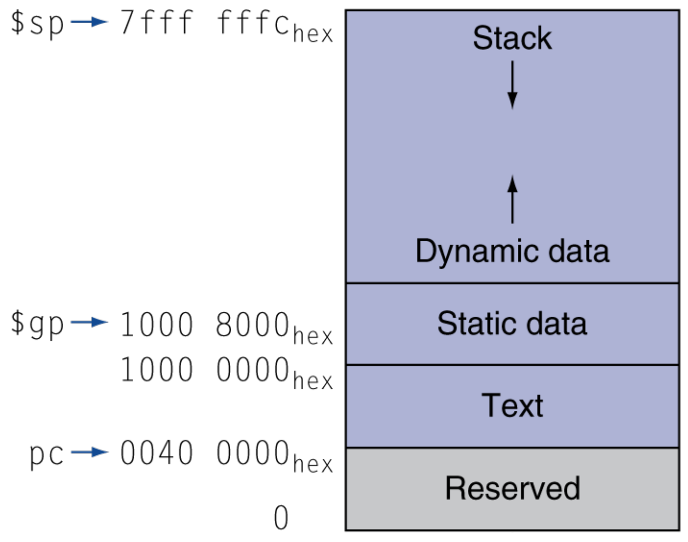


High address
$fp
$sp
Low address
a.
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
b.
$fp
$sp
c.

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

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



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument n in a0
  - Result in a0

Non-Leaf Procedure Example

› RISCV code:

```
        fact:
          addi sp, sp, -8       # adjust stack for 2 items
          sw   ra, 4(sp)        # save return address
          sw   a0, 0(sp)        # save argument - local variable
          li   t0, 2            # put a 2 into t0
          bgeu a0, t0, L1       # if n >= 2 take jump
          li   a0, 1            # otherwise, result is 1
          addi sp, sp, 8        #   pop 2 items from stack
          jr   ra               #   and return
L1:       addi a0, a0, -1       # else decrement n
          jal  fact             # recursive call to get fact(n-1)
          lw   t0, 0(sp)        # restore original n
          mul  a0, t0, a0       # multiply to get result n * fact(n-1)
          lw   ra, 4(sp)        # restore return address
          addi sp, sp, 8        # pop 2 items from stack
          jr   ra               # and return
```

Summary

- RISC-V ISA is a new academic RISC ISA used as examples
  - RV32 forms the base set of architectural feature
  - 32-bit architecture
  - 32 General Purpose Register File
- Each instruction in an ISA must be uniquely encoded.
  - Encoding and semantics of ISA must tradeoff between hardware efficiency and usability.

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Hayden So (HKU)
- MIT material derived from course 6.823
- UCB material derived from course CS152, CS252
- HKU material derived from course ELEC3441