Universität Bremen

dfki Deutsches Forschungszentrum für Künstliche Intelligenz German Research Center for Artificial Intelligence

Systems of High Safety and Security

Lecture 11 from 14.01.2026: A Real ISA: TinyRV32I

Winter term 2025/26

Christoph Lüth

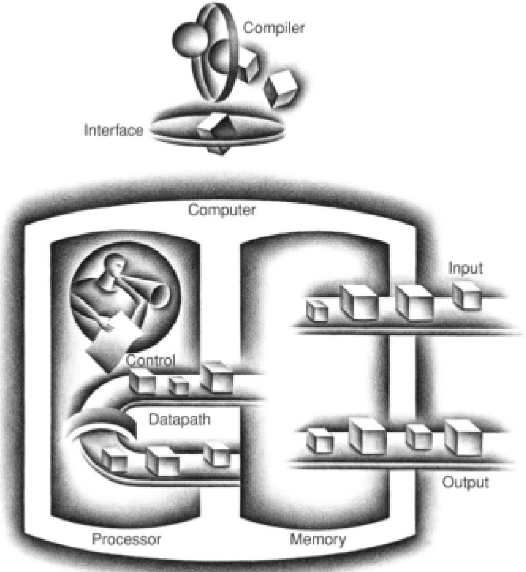
Organisatorisches

- Die Vorlesung am 28.01. **fällt aus!**
- Bitte an der stud.ip-Evaluation teilnehmen.

Roadmap

- Introduction
- Legal Requirements - Norms and Standards
- The Development Process
- Hazard Analysis
- The Big Picture: Hybrid Systems
- Temporal Logic with LTL and CTL
- Operational Semantics
- Axiomatic Semantics - Specifying Correctness
- Floyd-Hoare Logic - Proving Correctness
- A Simple Compiler and its Correctness
- TinyRV32I
- Hardware Verification & Conclusions

Basic Organisation of a Computer



How does the CPU work?

- The CPU works in a cycle:
  1. **Fetch** instruction from memory
  2. **Process** instruction (may necessitate fetching additional data from memory)
  3. **Store** result in memory
- Programmer's view: **finite state machine**
- Programmed via **assembly language**

The Programmer's View

- The **Instruction Set Architecture** (ISA) defines:
  - Instructions (encoding in memory and semantics)
  - Number and types of registers
  - Memory access, addressing modes
- The **Application Binary Interface** (ABI) defines additionally:
  - calling conventions
  - system software interface

Existing ISAs

- Intel/AMD x86:
  - Used in desktops and servers
  - Evolved from Intel 8086
  - Example of an organically grown **CISC** architecture (Complex instruction set computer)
- ARM:
  - Used in mobile devices and SoCs (smartphones, tablets)
  - Evolved from same roots as RISC-V
  - ARM sells **licensed IP cores**
  - Example of a **RISC** architecture (Reduced instruction set computer)

A Brief History of RISC-V

- Predecessors: RISC architectures originating from UC Berkeley (e.g. SPARC) and from Stanford (MIPS)
- RISC-V started in May 2010 as a summer project at UC Berkeley.
- First publication of ISA in May 2011.
- First tapeout of RISC-V chip in 2011.
- First RISC-V Workshop in January 2015.

RISC-V Today

- RISC-V chips commercially available.
- RISC-V International (Foundation) incorporated in Switzerland.
  -
  - Members include Alibaba, Google, Huawei, Intel, Qualcomm, ZTE, Seagate, Western Digital (premium); Infineon, nvidia, NXP, Synopys, Cadence, Sony, Siemens, Samsung, IBM, Hitachi, ...
- Commercial ecosystem beginning to materialize (e.g. Codasip in Munich)
- RISC-V is part of movement to **open source hardware** (open silicone)

RISC-V Design Principles

1. Regularity supports simplicity.

RISC-V Design Principles

1. Regularity supports simplicity.

2. Smaller is faster.

RISC-V Design Principles

1. Regularity supports simplicity.
2. Smaller is faster.
3. Make the common case fast.

RISC-V Design Principles

1. Regularity supports simplicity.
2. Smaller is faster.
3. Make the common case fast.
4. Good design demands good compromises.

Operations

- Basic Arithmetic: addition, subtraction, multiplication, division.
- Operate on **registers**.
- Example: Addition

```
a= b+ c;
```

```
add a, b, c
```

- Example: Subtraction

```
x= y- z;
```

```
sub x, y, z
```

Complex Expressions

- For complex expressions, we use **temporary** registers

```
a= b+ c- d;
```

```
sub x, c, d
add a, b, x
```

- Note we can **reuse** temporary registers.

Operands

- In C, we can **declare** an arbitrary amount of variables of many different types (**short int**, **int**, **char**, **double**)
- Here, the number and type is **fixed**:
  - 32 **registers** of 32 bit each (one **word**)
  - Note contents are **untyped**
  - These are called x0. . . x31
  - x0 is hard-wired to constant 0.
  - Other registers have other uses (per convention).
- Why 32? **Smaller is faster.**

RISC-V Register Set

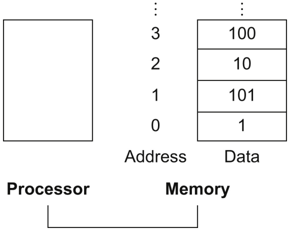
Register	Name	Use
x0	zero	Constant value 0
x1		
x2		
x3		
x4		
x5		
x6		
x7		
x8		
x9		
x10		
x11		
x12		
x13		
x14		
x15		

Register	Name	Use
x16		
x17		
x18		
x19		
x20		
x21		
x22		
x23		
x24		
x25		
x26		
x27		
x28		
x29		
x30		
x31		

Width of registers: 32 bits.

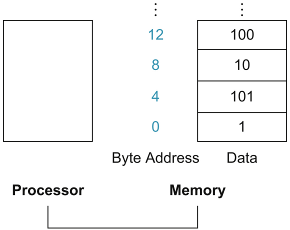
Memory Operands

- **Data transfer instructions** access the memory.
- Memory is (can be viewed as) a large, one-dimensional array.



Memory Operands

- **Data transfer instructions** access the memory.
- Memory is (can be viewed as) a large, one-dimensional array.
- Adressed in **bytes** (one word = 4 bytes = 32 bits)
- Acessing memory: `lw t, offset(s)`
  - Read from address in register `s` (plus `offset`) into target register `t`.
  - Note that in C, `a[i]` is **defined** to be `*(a+ i)`



Example:

```
x= x+ a[2]
```

Assuming `a` is `x20` and `x` is `x21`

```
lw  x9, 8(x20)      ;; x9 = *(x20+ 8) == *(x20+4*2) == a[2]
add x21, x21, x9    ;; x = x+ a[2]
```

Constants and Immediates

- Operations can also use **constants** (immediates) encoded into the instruction.
- Immediates are 12 bit long and can be signed or unsigned.
- Examples:

```
a= a+ 4
b= b- 12
```

```
addi x5, x5, 4      ;; x5 is a
addi x6, x6, -12    ;; x6 is b
```

- Initialization:

```
a= 0
b= 125
c= -78
```

```
addi x5, zero, 0      ;; x5 is a
addi x6, zero, 125    ;; x6 is b
addi x7, zero, -78    ;; x7 is c
```

Larger Constants

- To create large immediates, use **lui**
- Loads a 20-bit immediate into the most significant 20 bits, fills LSB with 0.
- Example:

```
a= 0xABCDE123;
```

```
lui x5, 0xABCDE
addi x5, x5, 0x123
```

- If 12-bit immediate is negative (Bit 11 is 1), upper immediate must be incremented by one:

```
a= 0xDEADBEEF;
```

```
lui x5, 0xDEADC
addi x5, x5, 0xEEF
```

Shifts and Logical Instructions

- Logical operations operate on bit patterns:

Logical operation	**C**	**RISC-V**	
Shift left	<<	**sll**, **slli**	Fills with 0, multiplication with 2^i
Shift right	>>	**srl**, **srli**	Fills with 0
Shift right arithmetic	>>	**sra**, **srai**	Fills with sign bit, division by 2^i
Bitwise And	&	**and**, **andi**	Not the logical conjunction (&&)
Bitwise Or	|	**or**, **or**	Not the logical disjunction (||)
Bitwise Xor	^	**xor**, **xori**	

- Immediate arguments for **andi**, **ori**, **xori** are 12 bits sign-extended.

Shifts and Logical Instructions

- Logical operations operate on bit patterns:

Logical operation	C	RISC-V	
Shift left	<<	**sll**, **slli**	Fills with 0, multiplication with 2^i
Shift right	>>	**srl**, **srli**	Fills with 0
Shift right arithmetic	>>	**sra**, **srai**	Fills with sign bit, division by 2^i
Bitwise And	&	**and**, **andi**	Not the logical conjunction (&&)
Bitwise Or	|	**or**, **or**	Not the logical disjunction (||)
Bitwise Xor	^	**xor**, **xori**	

- Immediate arguments for **andi**, **ori**, **xori** are 12 bits sign-extended.
- Logical **negation** is exclusive-or with 0xFFFFFFFF:

```
xori s3, s4, -1   ;; Note -1 is sign-extended to 0xFFFFFFFF
```

- Immediate arguments for the shift operations are 5 bits (why?)
- In C, right-shift of negative values is **implementation-defined**.

The Full Peano: Multiplication

- Multiplication: what is the problem?

The Full Peano: Multiplication

- Multiplication: what is the problem?
  Multiplying two N-bit numbers produces $2N$-bit number
- We can either ignore that, or get the result in parts:

Multiply low	**mul** x0, x1, x3	Lower 32 bit of result
Multiply high	**mulh** x0, x1, x2	Higher 32 bits of result, both signed
Multiply high	**mulhu** x0, x1, x2	Higher 32 bits of result, both unsigned
Multiply high	**mulsh** x0, x1, x2	Higher 32 bits of result, signed/unsigned

- Example:

```
mulh  x1, x5, x6
mul   x2, x5, x5       ;; Now x1,x2 = x5* x6
```

- No immediates
- Peano arithmetic is not part of **basic** RISC-V (RV32I), but RVM **extension**.

The Full Peano: Division

- Division has different problem: remainders

div x5, x6, x7	Signed division
divu x5, x6, x7	Unsigned division
rem x5, x6, x7	Signed remainder
remu x5, x6, x7	Unsigned remainder

Program and Control Flow

- Programs are stored in memory.
- The **program counter** (PC) determines next instruction to be fetched from memory.
- **All** instructions are four bytes long.
- PC is incremented by four while executing the instruction.
- Example:

Memory address	Instruction
0x538	**lw** x9, 20(x20)
0x53C	**add** x9, x21, x9
0x540	**sw** x9, 44(x20)

Control Flow: Branches

- To achieve Turing completeness, need some kind of **conditional** aka. **branches**.
- These can **conditionally** change the **control flow**.
- Conditional branches compare two registers and if the condition is true, set PC to target given by an **offset** from current PC.
- Comparisons include equality, less, and their negation:

beq x5, x6, n	If x5 equals x6 branch to PC+n
bne x5, x6, n	If x5 does not equal x6 then go to PC+n
blt x5, x6, n	If x5 is less than x6 (signed) then go to PC+n
bge x5, x6, n	If x5 is greater or equal to x6 (signed) then go to PC+n
bltu x5, x6, n	If x5 is less than x6 (unsigned) then go to PC+n
bgeu x5, x6, n	If x5 is greater or equal to x6 (unsigned) then go to PC+n

- **bgt**, **ble**, **bgtu**, **bleu** are syntactic sugar, e.g. **bgt** x5, x6, n is **blt** x6, x5, n

Conditional Branches

- Branch offset is given as 13-bit signed immediate (only 12 bits given, must be even).
  - Can only branch to $\pm 2^{12} = 4096$ bytes.
- In assembly, branches are given as **labels:**

```
  addi x4, x4, -1          ; decrement x4 by one
  beq  x5, x0, afterdiv    ; if x5 is zero skip division
  div  x6, x5, x4          ; x6 := x4 div x5
afterdiv:
  addi x6, x6, 1           ; increment x6 by one
```

- Control structures such as case conditions, switches, and loops built from these elementary branches.
- Other ISAs use **condition flags** upon which to branch, or transfer result of comparison to register and branch on register content.

Control Flow: Jumps

- An unconditional branch is called a **jump**, written `j target`.

```
switch (button) {
  case 1:
    amt= 5;
    break;
  case 2:
    amt= 10;
    break;
  case 3:
    amt= 100;
    break;
  default:
    amt= 0;
  }
```

(Note **switch** is rather weird.)

```
; x5 = button, x6= amt
case1:
  addi x8, zero, 1
  bne x5, x8, case2
  addi x6, zero, 5
  j done
case2:
  addi x8, zero, 2
  bne x5, x8, case3
  addi x6, zero, 10
  j done
case3:
  addi x8, zero, 3
  bne x5, x8, default
  addi x6, zero, 100
  j done
default:
  addi x6, zero, 0
done:
```

Example: Loops (1)

- A simple loop: find the integer square root.

```
int a, i;
i= 0;
while ((i+1)*(i+1) <= a) i++;
```

```
;; a is in x5, i in x6
   addi  x6, zero, 0      ; set i to 0
loop:
   addi  x7, x6, 1        ; x7= i+1
   mul   x8, x7, x7       ; x8= (i+1)*(i+1)
   bgt   x8, x5, end      ; if x8 > a then exit
   add   x6, x7, zero     ; set i to i+1
   j     loop
end:                      ; result is in x6
```

Example: Arrays

- This example finds an element in an array and returns the index,

```c
int a[100];
int x;

int i= 100-1:
while (i>= 0 && a[i] != x) i--;
```

```
;; array is in x8, x in x9, i in x5
  add x5, zero, 99
loop:
  blt x5, zero, exit
  slli x6, x5, 2      ; x6= x5*4
  add  x6, x6, x8     ; x6= &a[x5]
  lw   x7, 0(x6)      ; x7= a[x5]
  beq  x7, x9, exit   ; if a[x5] == x exit
  addi x5, x5, -1     ; x5–
  j loop
exit:
```

Arrays of Characters: Strings

- Strings are arrays of **characters**. The end of the string is denoted by the **zero character**.
- Characters can be **bytes** (8 bit, traditional ASCII) or **half-words** (16 bit, Unicode[1]).

<table>
<tr><th>ASCII value</th><th>Char-acter</th><th>ASCII value</th><th>Char-acter</th><th>ASCII value</th><th>Char-acter</th><th>ASCII value</th><th>Char-acter</th><th>ASCII value</th><th>Char-acter</th><th>ASCII value</th><th>Char-acter</th></tr>
<tr><td>32</td><td>space</td><td>48</td><td>0</td><td>64</td><td>@</td><td>80</td><td>P</td><td>96</td><td>`</td><td>112</td><td>p</td></tr>
<tr><td>33</td><td>!</td><td>49</td><td>1</td><td>65</td><td>A</td><td>81</td><td>Q</td><td>97</td><td>a</td><td>113</td><td>q</td></tr>
<tr><td>34</td><td>"</td><td>50</td><td>2</td><td>66</td><td>B</td><td>82</td><td>R</td><td>98</td><td>b</td><td>114</td><td>r</td></tr>
<tr><td>35</td><td>#</td><td>51</td><td>3</td><td>67</td><td>C</td><td>83</td><td>S</td><td>99</td><td>c</td><td>115</td><td>s</td></tr>
<tr><td>36</td><td>$</td><td>52</td><td>4</td><td>68</td><td>D</td><td>84</td><td>T</td><td>100</td><td>d</td><td>116</td><td>t</td></tr>
<tr><td>37</td><td>%</td><td>53</td><td>5</td><td>69</td><td>E</td><td>85</td><td>U</td><td>101</td><td>e</td><td>117</td><td>u</td></tr>
<tr><td>38</td><td>&</td><td>54</td><td>6</td><td>70</td><td>F</td><td>86</td><td>V</td><td>102</td><td>f</td><td>118</td><td>v</td></tr>
<tr><td>39</td><td>'</td><td>55</td><td>7</td><td>71</td><td>G</td><td>87</td><td>W</td><td>103</td><td>g</td><td>119</td><td>w</td></tr>
<tr><td>40</td><td>(</td><td>56</td><td>8</td><td>72</td><td>H</td><td>88</td><td>X</td><td>104</td><td>h</td><td>120</td><td>x</td></tr>
<tr><td>41</td><td>)</td><td>57</td><td>9</td><td>73</td><td>I</td><td>89</td><td>Y</td><td>105</td><td>i</td><td>121</td><td>y</td></tr>
<tr><td>42</td><td>*</td><td>58</td><td>:</td><td>74</td><td>J</td><td>90</td><td>Z</td><td>106</td><td>j</td><td>122</td><td>z</td></tr>
<tr><td>43</td><td>+</td><td>59</td><td>;</td><td>75</td><td>K</td><td>91</td><td>[</td><td>107</td><td>k</td><td>123</td><td>{</td></tr>
<tr><td>44</td><td>,</td><td>60</td><td><</td><td>76</td><td>L</td><td>92</td><td>\</td><td>108</td><td>l</td><td>124</td><td>|</td></tr>
<tr><td>45</td><td>-</td><td>61</td><td>=</td><td>77</td><td>M</td><td>93</td><td>]</td><td>109</td><td>m</td><td>125</td><td>}</td></tr>
<tr><td>46</td><td>.</td><td>62</td><td>></td><td>78</td><td>N</td><td>94</td><td>^</td><td>110</td><td>n</td><td>126</td><td>~</td></tr>
<tr><td>47</td><td>/</td><td>63</td><td>?</td><td>79</td><td>O</td><td>95</td><td>_</td><td>111</td><td>o</td><td>127</td><td>DEL</td></tr>
</table>

Traditional ASCII representation

[1] Unicode can also be encoded in other lengths, it's complicated.

String Handling

- Strings have **variable length**.
- To access bytes, use **lb**, **lbu** or **sb**:
  - **lb** x5, imm(x6) reads sign-extended byte from adress in x6 plus offset imm into x5
  - **lbu** x5, imm(x6) reads zero-extended byte from adress in x6 plus offset imm into x5
  - **sb** x5, imm(x6) store byte from x5
- To access half-words, use **lh**, **lhu**, **sh** which are similar.

More Complex Expressions

- Given this piece of C:

```
t= (a+b)− (c+d);
```

- Is the following translation correct:

```
add x5 x1 x2     ; x5 := a+b
sub x5 x5 x3     ; x5 := x5- c == a+b- c
sub x5 x5 x4     ; x5 := x5- d == a+b- c- d == a+c - (c+d)
```

More Complex Expressions

- Given this piece of C:

```
t= (a+b)- (c+d);
```

- Is the following translation correct:

```
add x5 x1 x2     ; x5 := a+b
sub x5 x5 x3     ; x5 := x5- c == a+b- c
sub x5 x5 x4     ; x5 := x5- d == a+b- c- d == a+c - (c+d)
```

- Only if no **overflow** occurs!

Detecting Overflow

- Other ISAs have **overflow detection** (e.g. a status bit, "carry", which gets set if result overflows)
- RISC-V does not: "*We did not include special instruction-set support for overflow checks [. . .] as many checks can be cheaply implemented.*" [?]
- For unsigned addition: overflow if result less than either of the operands:

```
add x5, x6, x7
bltu x5, x6, overflow
```

- For signed addition:

If sign of one operand is known:

```
addi x5, x6, +imm
blt  x6, x6, overflow
```

General case: sum should be less than one of operands iff other operand is negative:

```
add t5, t6, t7
slti t8, t7, 0
slt t9, t5, t6
bne t8, t9, overflow
```

Syntactic Sugar: Pseudo-Instructions

- RISC-V lacks some instructions found in other ISAs which are used quite often.
- These can be derived as **syntactic sugar** (abbreviations).
- Examples include:

Pseudoinstruction	Base instruction	Meaning
nop	**addi** x0, x0, 0	No operation
li rd, imm	**lui** followed by **addi**	Load immediate
mv rd, rs	**addi** rd, rs, 0	Copy register
not rd, rs	x**ori** rd, rs, −1	Invert register
j offset	**jal** x0, offset	Unconditional jump
jal offset	**jal** x1, offset	Jump and link
ret	**jalr** x0, x1, 0	Return

Advanced Aspects

Function Calls

To support function calls, we need:

1. A way to **save** (and **restore**) the current PC.
2. A way to pass **parameter** and **return values**.
3. A way to **encapsulate** the function body.

Calling and Returning

- RISC-V provides two instructions to call and return from a function.
- **jal** ra, adress saves the PC to ra and jumps to adress
- **jr** ra jumps to address in ra

```c
int main() {
 foo();
 foo();
 }
...
void foo() {
  return;
  }
```

```
main:
  jal ra, foo
  jal ra, foo

...

foo:
  jr ra
```

Passing Parameters

- Parameters are passed (and values returned) in **registers**.
- Registers x1 to x8 are used to pass arguments.
- This is **binding convention**, and hence they are called a0 to a7.
- Registers x1 (and if needed for large values, x2) also hold the **return value**.
  - Functions only return value, much like C.
- Register x1 holds the **return adress** and is called ra

RISC-V Register Set

Register	Name	Use
x0	zero	Constant value 0
x1	ra	Return address
x2		
x3		
x4		
x5		
x6		
x7		
x8		
x9		
x10	a0	Function arg/return val
x11	a1	Function arg/return val
x12	a2	Function argument
x13	a3	Function argument
x14	a4	Function argument
x15	a5	Function argument

Register	Name	Use
x16	a6	Function argument
x17	a7	Function argument
x18		
x19		
x20		
x21		
x22		
x23		
x24		
x25		
x26		
x27		
x28		
x29		
x30		
x31		

Example: Function Call with Arguments and Return Value

▶

```c
int main()
{ int y;
  y= diffofsums(2, 3, 4);
}

int diffofsums(int x, int y, int z)
{
  int r;
  r= (x+y)-(y+z);
  return r;
}
```

```
main:
  addi a0, 2
  addi a1, 3
  addi a2, 4
  jal ra, diffofsums
  add  x23, a0, zero  ;; x23 is y

diffofsums:
  add x5, a0, a1    ;; x+ y
  add x6, a1, a2    ;; y+ z
  sub x7, x5, x6    ;; r= (x+y)-(y+z)
  add a0, x7, zero  ;; set return value
  jr  ra            ;; return
```

The Stack

- What happens if we need more than eight parameters?
- We place them on the **stack**.
- Each function may allocate a **stack frame** to store parameters, local variables etc.
- Just like trees, stacks grow **downwards**.
- Register x2 is used as stack pointer, called sp.

Address	Data	
BEFFFAE8	AB000001	← sp
BEFFFAE4		
BEFFFAE0		
BEFFFADC		
⋮	⋮	

(a)

Address	Data	
BEFFFAE8	AB000001	
BEFFFAE4	12345678	
BEFFFAE0	FFEEDDCC	← sp
BEFFFADC		
⋮	⋮	

(b)

Example:

- (a) sp points to last used value.
- (b) After values 0x12345678 and 0xFFEEDDCC have been stored on the stack.

RISC-V Register Set

Register	Name	Use
x0	zero	Constant value 0
x1	ra	Return address
x2	sp	Stack pointer
x3		
x4		
x5		
x6		
x7		
x8		
x9		
x10	a0	Function arg/return val
x11	a1	Function arg/return val
x12	a2	Function argument
x13	a3	Function argument
x14	a4	Function argument
x15	a5	Function argument

Register	Name	Use
x16	a6	Function argument
x17	a7	Function argument
x18		
x19		
x20		
x21		
x22		
x23		
x24		
x25		
x26		
x27		
x28		
x29		
x30		
x31		

Encapsulation

- Functions should be encapsulated, i.e. **leave no trace**.
- In C, we have local variables which cannot be changed by called function.
- Here, variables are registers — who **preserves** which registers?
  - **Callee-saved**: Called function must make sure not to change these.
  - **Called-saved**: Calling function must save these if needed after call.

Preserved (callee-saved)	Nonpreserved (caller-saved)
Return adress ra	Temporary registers x5-x7,x28-x31
Stack pointer sp	Argument registers a0-a7
Saved registers (x19-x27)	
Stack above sp	Stack below sp

RISC-V Register Set

Register	Name	Use
x0	zero	Constant value 0
x1	ra	Return address
x2	sp	Stack pointer
x3		
x4		
x5	t0	Temporary register
x6	t1	Temporary register
x7	t2	Temporary register
x8	s0/fp	Saved reg/frame pointer
x9	s1	Saved register
x10	a0	Function arg/return val
x11	a1	Function arg/return val
x12	a2	Function argument
x13	a3	Function argument
x14	a4	Function argument
x15	a5	Function argument

Register	Name	Use
x16	a6	Function argument
x17	a7	Function argument
x18	s2	Saved register
x19	s3	Saved register
x20	s4	Saved register
x21	s5	Saved register
x22	s6	Saved register
x23	s7	Saved register
x24	s8	Saved register
x25	s9	Saved register
x26	s10	Saved register
x27	s11	Saved register
x28	t3	Temporary registers
x29	t4	Temporary registers
x30	t5	Temporary registers
x31	t6	Temporary registers

How To Call A Function (1)

1. Allocate space on stack to save preserved registers
2. Store values of registers on stack.
3. Execute function.
4. Restore original values from stack.
5. Deallocate space on stack.

Example Revisited

- Here is a version of diffofsums which saves the preserved registers s3-s5.
- Calling function does not need to be changed (obvsly).

```c
int diffofsums(int x,
               int y,
               int z)
{
  int r;


  r= (x+y)-(y+z);




  return r;
}
```

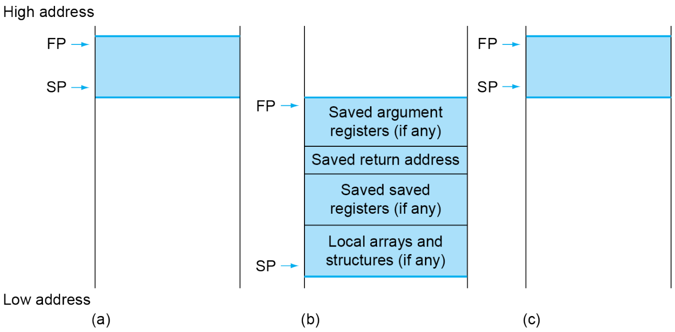
```
diffofsums:
  addi sp, sp, -12    ;; make space for three registers
  sw   s3, 8(sp)
  sw   s4, 4(sp)      ;; save s3- s5 on stack
  sw   s5, 0(sp)
  add  s3, a0, a1     ;; x+ y
  add  s4, a1, a2     ;; y+ z
  sub  s5, s3, s4     ;; r= (x+y)-(y+z)
  add  a0, s3, zero   ;; set return value
  lw   s3, 8(sp)
  lw   s4, 4(sp)      ;; restore s3- s5 from stack
  lw   s5, 12(sp)
  addi sp, sp, 12     ;; deallocate stack space
  jr   ra             ;; return
```

How To Call A Function (2)

1. Allocate space on stack (**stack frame**).
2. Store preserved registers on stack.*
3. Store return address.*
4. Execute function.
5. Restore return address.*
6. Restore registers.*
7. Deallocate space on stack frame.

The start of the stack frame is called fp (frame pointer).

* if required.



(a) before, (b) during, (c) after function all

Example: A Recursive Function

- The factorial function.

```c
int factorial(int n) {
  if (n <= 1)
    return 1;



  else
    return
      n* factorial(n-1);
```

```
fact: addi sp, sp, -8
      sw a0, 4(sp)         ; save a0, ra
      sw ra, 0(sp)
      addi t0, zero, 1     ; temp = 1
      bgt a0, t0, else     ; if a0 > 0
      addi a0, zero, 1     ; set return value
      addi sp, sp, 8       ; remove stack frame
      jr ra                ; return 1
else: addi a0, a0, -1      ; a0= a0- 1
      jal factorial        ; recursive call
      lw t1, 4(sp)         ; old value of a0
      lw ra, 0(sp)         ; old return value
      addi sp, sp, 8       ; remove stack frame
      mul a0, t1, a0
      jr ra                ; return n*fact(n-1)
```

Full RISC-V Register Set

Register	Name	Use
x0	zero	Constant value 0
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary register
x6	t1	Temporary register
x7	t2	Temporary register
x8	s0/fp	Saved reg/frame pointer
x9	s1	Saved register
x10	a0	Function arg/return val
x11	a1	Function arg/return val
x12	a2	Function argument
x13	a3	Function argument
x14	a4	Function argument
x15	a5	Function argument

Register	Name	Use
x16	a6	Function argument
x17	a7	Function argument
x18	s2	Saved register
x19	s3	Saved register
x20	s4	Saved register
x21	s5	Saved register
x22	s6	Saved register
x23	s7	Saved register
x24	s8	Saved register
x25	s9	Saved register
x26	s10	Saved register
x27	s11	Saved register
x28	t3	Temporary registers
x29	t4	Temporary registers
x30	t5	Temporary registers
x31	t6	Temporary registers

TinyRISCV

- TinyRISCV is a small but representative excerpt of the RISC-V ISA suitable for teaching.
- Many versions exist, this one invented by Christopher Batten (Cornell U)
- Two versions:

TinyRV1:	TinyRV2:
- ADD, ADDI, MUL	- ADD, ADDI, SUB, MUL
- LW, SW	- AND, ANDI, OR, ORI, XOR, XORI
- JAL, JR	- SLT, SLTI, SLTU, SLTIU
- BNE	- SRA, SRAI, SRL, SRLI, SLL, SLLI
	- LUI, AUIPC
	- LW, SW
	- BEQ, BNE, BLT, BGE, BLTU, BGEU
	- CSRR, CSRW

Implementing the Simple Stack Machine

Register Allocation and Context

- The stack pointer is x2 (sp)
  - x2 points to the **top** of the stack
- Two registers for temporary arguments: x5 and x6 (t0 and t1)
- We assume there is injective mapping Γ which maps variable names to addresses

LOAD, ADD, STORE

LOADI n is

```
lui  x5, n >> 12
add  x5, x5, n & 0xFFF  ;; load n into x5
addi x2, x2, 4          ;; incr SP
sw   x5, 0(x2)          ;; push
```

LOAD x: Let $a = \Gamma(x)$, adress denoted by x

```
lui  x5, a >> 12
lw   x5, a & 0xFFF(x5)  ;; read from a
addi x2, x2, 4  ;; inc SP
sw   x5, 0(x2)    ;; push
```

STORE x: Let $a = \Gamma(x)$

```
lw   x5, 0(x2)   ;; top
lui  x6, a >> 12   ;; load a to x6
sw   x5, (a & 0xFFF)(x6)
addi x2, x2, -4   ;; pop (dec sp)
```

ADD und JMP

ADD:

```
lw x5, 0(x2)    ;; top
lw x6, -4(x2)   ;; top(pop)
add x5, x5, x6
addi x2, x2, -4   ;; pop
sw x5, 0(x2)     ;; push
```

JMP *n*:

```
    beq x0, x0, n    ;; condition is always
true
```

JMPLE *n*:

```
    lw x5, 0(x2)     ;; top
    lw x6, -4(x2)    ;; top(pop)
    ble x6, x5, n    ;; jump if x6< x5
```

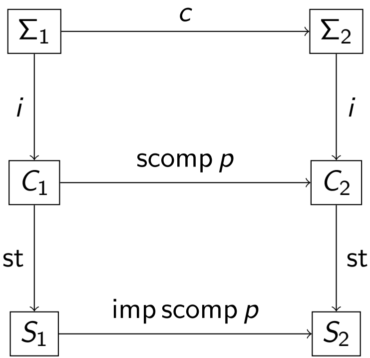
A Formal Semantics for TinyRV32

- Model Tiny32I as data type **TinyRV32**, just like **Inst** for the simple stack machine
- Define a **system state** $S = \langle Reg, Mem \rangle$ which consists of
  - Register file $Reg : \text{Seq}(x1, \ldots, x32) \to W_{32}$ maps each register to a word $w \in W_{32}$, with $W_{32} = \{0..2^{32} - 1\}$
  - Memory state $Mem : W_{32} \to W_8$ (with $W_8 = 0..2^8 - 1$, i.e. unsigned bytes)
- Define an **evaluation relation** such that $\langle c, s \rangle \to_{\text{RV}} s'$ iff program c in state s evaluates to state s'

Translating from the Simple Stack Machine to TinyRV32I

- Need to map instructions to instructions, and configurations to system states.
- Function $\mathrm{imp} : \mathbf{Instr} \to \mathrm{Seq}(\mathbf{TinyRV32})$ maps stack machine instructions to a **sequence** of **TinyRV32** instructions.
- Function $\mathrm{st} : \mathit{Config} \to S$ maps a simple stack machine configuration to a system state.
  - The stack is actually located in memory.
- Correctness given by $\langle p, c \rangle \to c' \iff \langle \mathrm{imp}\, p, \mathrm{st}(c) \rangle \to^{*}_{\mathrm{RV}} \mathrm{st}(c')$

The Whole Stack: from C to TinyRV32



c is C program, Σ_1, Σ_2 C system states

C_1, C_2 are stack machine configurations

S_1, S_2 are **TinyRV32** system states

Problem: Implementing a Stack

- Stack is an example of an **abstract data type**
- It is given by **operations** and **equations**
- Here:

$$\begin{aligned} \text{top} :& S \to \mathbb{N} \\ \text{pop} :& S \to S \\ \text{push} :& S \times \mathbb{N} \to S \end{aligned}$$

$$\begin{aligned} \text{top}(\text{push}(s, x)) &= x \\ \text{pop}(\text{push}(s, x)) &= s \end{aligned}$$

- Allows to deduce e.g. $\text{top}(\text{pop}(\text{push}(\text{push}(s, x), y))) = x$.

Implementing Stacks

- Needs an implementation of the data type, and operations.
- Need to show **equations** hold.

```c
#define MAX_SP 255

typedef struct {
    int sp;
    int st [MAX_SP];
} stack;

int top(stack s)
{ int d;

  if (s.sp> 0)
    d= s.st[s.sp- 1];
  else
    d= -1;
  return d;
}
```

```c
stack pop(stack s)
{
  if (s.sp> 0)
    s.sp= s.sp- 1;
  return s;
}

stack push(stack s, int x)
{
  s.st[s.sp]= x;
  s.sp= s.sp+ 1;
  return s;
}
```