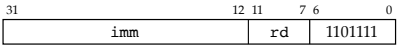
2.3. Control Instructions

JAL

`jal rd, addr`

R[rd] ← PC + 4

PC ← addr

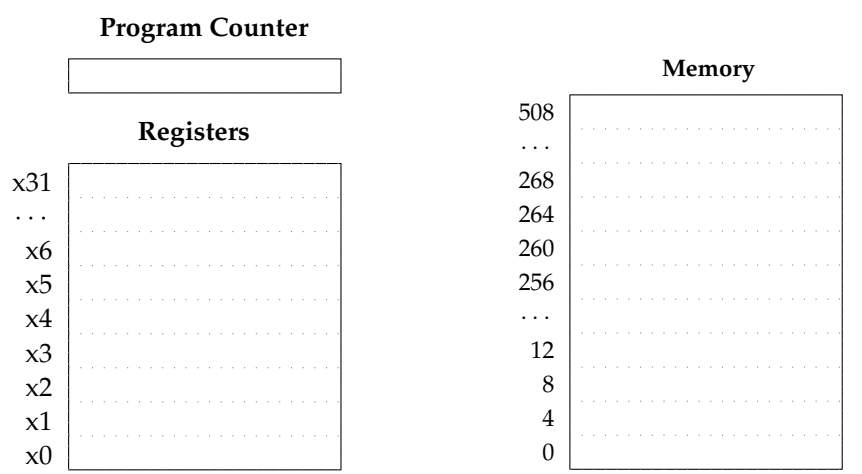


J-type Immediate

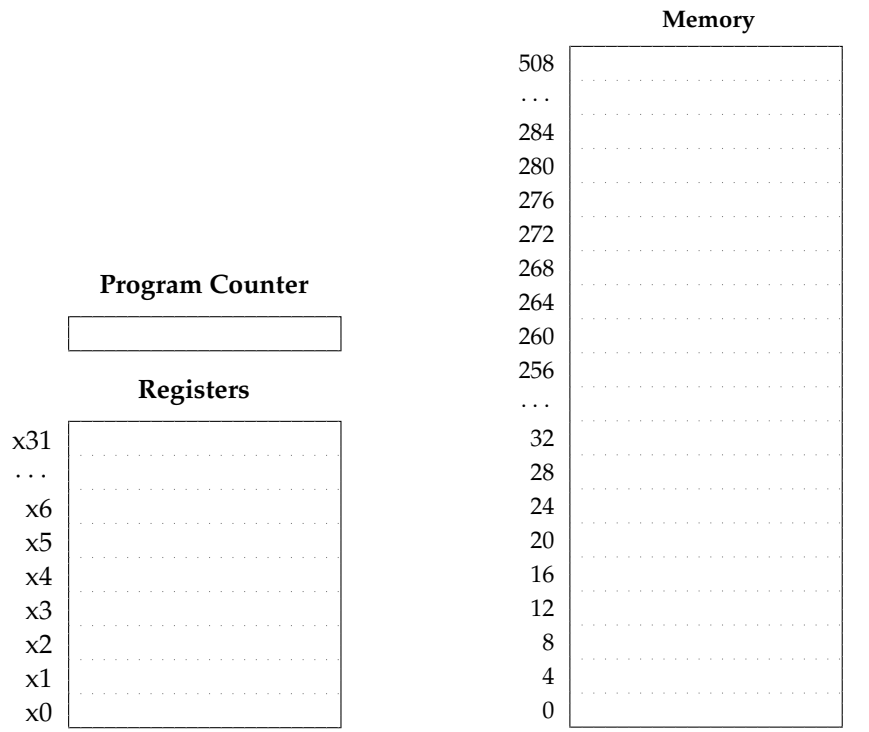
addr encoded as PC + sext(imm)

- Explanation of instruction semantics
  - addr : memory byte address
- *Jump and Link Instructions*
  - Write the current program counter (PC) to a general-purpose register
  - Unconditionally write the PC with a new target address

```
 0 addi x1, x0, 1  # R[x1] = 1
 4 jal  x2, 12     # R[x2] = PC + 4; PC = 12
 8 addi x3, x0, 3  # R[x3] = 3
12 addi x4, x0, 4  # R[x4] = 4
16 addi x5, x0, 5  # R[x5] = 5
```



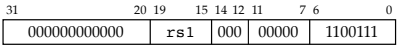
```
 0 addi x1, x0, 1
 4 jal  x0, 16
 8 addi x2, x0, 2
12 jal  x0, 24
16 addi x3, x0, 3
20 jal  x0, 8
24 addi x4, x0, 4
28 addi x5, x0, 5
```



JR

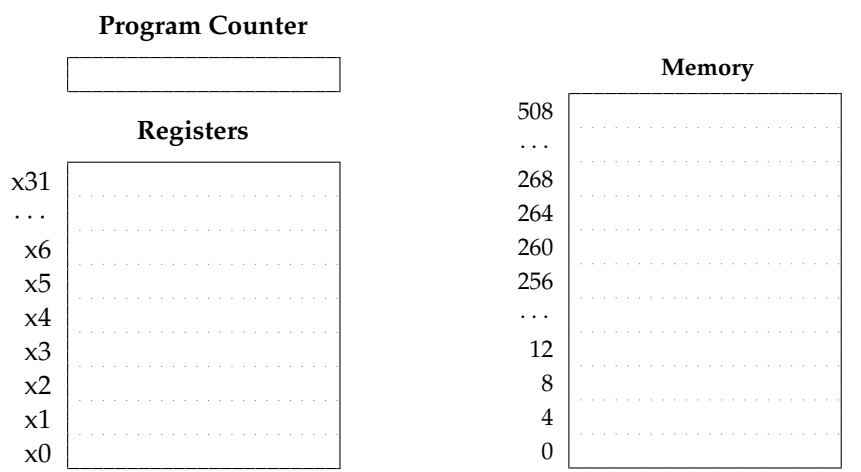
`jr rs1`

PC ← R[`rs1`]



- Explanation of instruction semantics
  - addr : memory byte address
- *Jump Register Instructions*
  - Read a new target address from a general-purpose register
  - Unconditionally write the PC with the target address

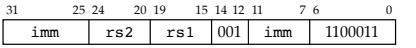
```
 0 addi x1, x0, 1   # R[x1] = 1
 4 addi x2, x0, 16  # R[x2] = 16
 8 jr   x2          # PC = R[x2]
12 addi x3, x0, 3   # R[x3] = 3
16 addi x4, x0, 4   # R[x4] = 4
20 addi x5, x0, 5   # R[x5] = 5
```



BNE

`bne rs1, rs2, addr`

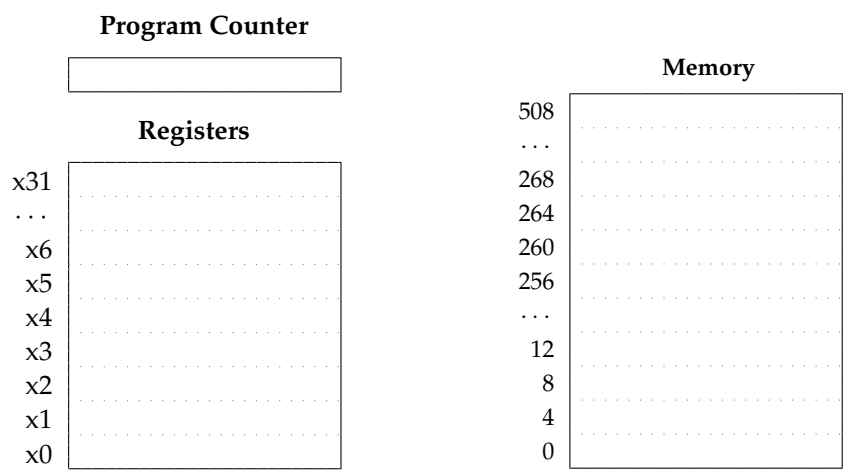
if (R[rs1] != R[rs2]) PC ← addr
else PC ← PC + 4



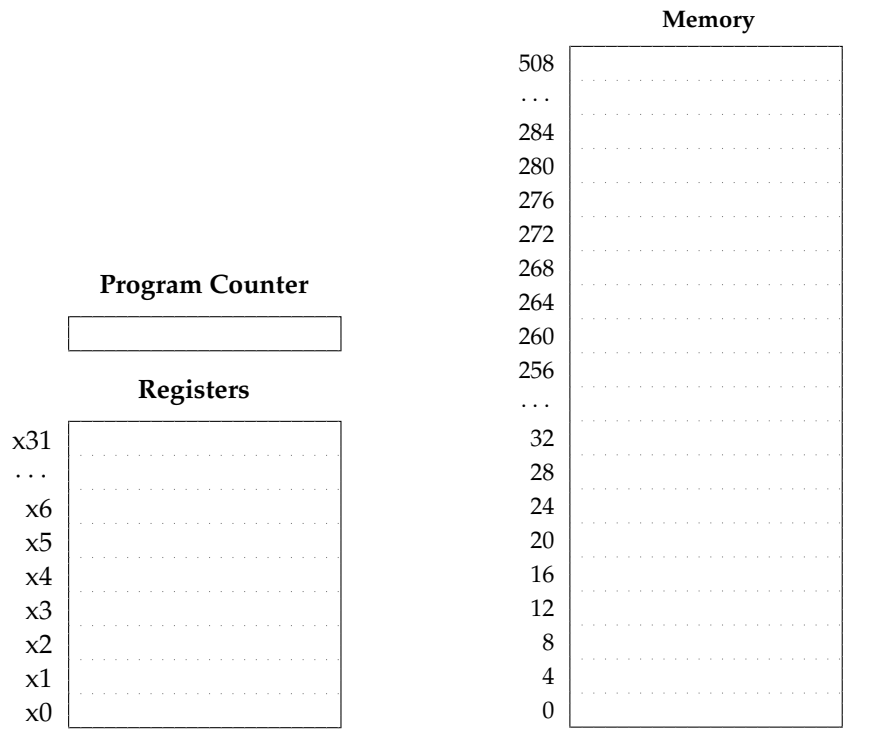
B-type Immediate
addr encoded as PC + sext(imm)

- Explanation of instruction semantics
  - addr : memory byte address
- *Branch Instructions*
  - Compare the values from two general-purpose registers
  - Conditionally write PC with a new target address based on comparison

```
 0 addi x1, x0, 1   # R[x1] = 1
 4 bne  x1, x0, 12  # if ( R[x1] != R[x0] ) PC = 12
 8 addi x2, x0, 2   # R[x2] = 2
12 addi x3, x0, 3   # R[x3] = 3
16 addi x4, x0, 4   # R[x4] = 4
```



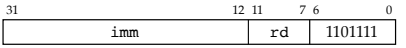
```
 0 addi x1, x0, 0
 4 addi x2, x0, 2
 8 addi x1, x1, 1
12 addi x4, x0, 4
16 addi x5, x0, 5
20 bne  x1, x2, 8
24 addi x6, x0, 6
```



JAL

```
jal rd, addr
```

R[rd] ← PC + 4
PC ← addr

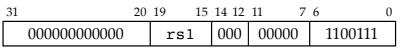


J-type Immediate
addr encoded as PC + sext(imm)

JR

```
jr rs1
```

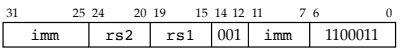
PC ← R[rs1]



BNE

```
bne rs1, rs2, addr
```

if (R[rs1] != R[rs2]) PC ← addr
else PC ← PC + 4



B-type Immediate
addr encoded as PC + sext(imm)

- Explanation of instruction semantics
  - addr : memory byte address
- *Jump and Link Instructions*
  - Write the current program counter (PC) to a general-purpose register
  - Unconditionally write the PC with a new target address
- *Jump Register Instructions*
  - Read a new target address from a general-purpose register
  - Unconditionally write the PC with the target address
- *Branch Instructions*
  - Compare the values from two general-purpose registers
  - Conditionally write PC with a new target address based on comparison
- Instruction memory addresses
  - Memory addresses will always be *byte* addresses (i.e., 4-byte word aligned), thus bottom two bits of the address must always be zero!
  - If target address is encoded as an immediate relative to the PC, then control instruction only supports a limited range
  - If target address is in a register supports entire virtual address space

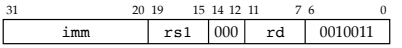
2.4. TinyRV1 ISA Specification (preliminary)

ADDI

`addi rd, rs1, imm`

R[rd] ← R[rs1] + sext(imm)

PC ← PC + 4



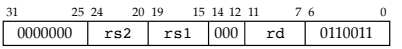
I-type Immediate

ADD

`add rd, rs1, rs2`

R[rd] ← R[rs1] + R[rs2]

PC ← PC + 4

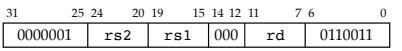


MUL

`mul rd, rs1, rs2`

R[rd] ← R[rs1] × R[rs2]

PC ← PC + 4

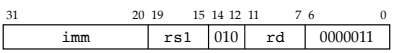


LW

`lw rd, imm(rs1)`

R[rd] ← M[R[rs1] + sext(imm)]

PC ← PC + 4



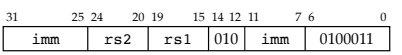
I-type Immediate

SW

`sw rs2, imm(rs1)`

M[R[rs1] + sext(imm)] ← R[rs2]

PC ← PC + 4



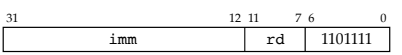
S-type Immediate

JAL

`jal rd, addr`

R[rd] ← PC + 4

PC ← addr



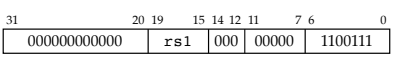
J-type Immediate

addr encoded as PC + sext(imm)

JR

`jr rs1`

PC ← R[rs1]

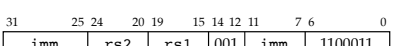


BNE

`bne rs1, rs2, addr`

if (R[rs1] != R[rs2]) PC ← addr

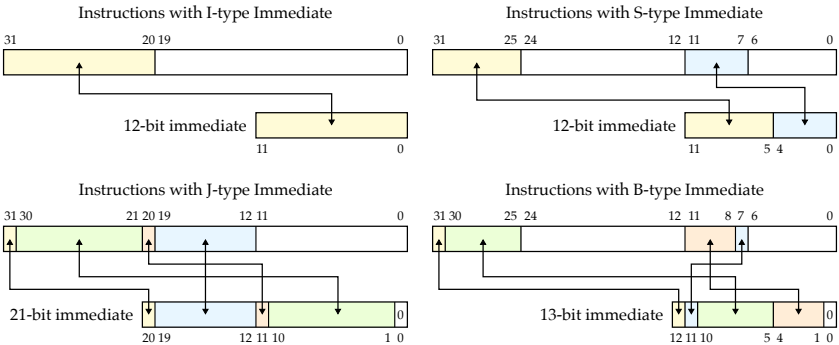
else PC ← PC + 4



B-type Immediate

addr encoded as PC + sext(imm)

TinyRV1 Immediate Types



Notes

- R[`x0`] is always zero
- PC reset value is `0x00000000`
- All addresses must be word aligned (i.e., bottom two address bits are zero)
- 512B of available physical memory, valid address range `0x00000000`–`0x000001ff`
- Unaligned or invalid memory accesses result in undefined behavior
- `nop` is pseudo-instruction for `addi x0, x0, 0`

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	J&L	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR RETired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

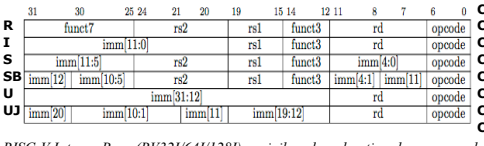
RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

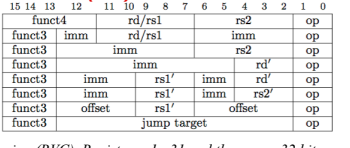
Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Load Double	CL	C.LD rd',rs1',imm	LD rd',rs1',imm*8
	Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
	Load Quad	CL	C.LQ rd',rs1',imm	LQ rd',rs1',imm*16
	Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Store Double	CS	C.SD rs1',rs2',imm	SD rs1',rs2',imm*8
	Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
	Store Quad	CS	C.SQ rs1',rs2',imm	SQ rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,imm
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

32-bit Instruction Formats



16-bit (RVC) Instruction Formats



RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MUL{W|D}	rd,rs1,rs2
	MULtiply upper Half	R	MULH	rd,rs1,rs2		
	MULtiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply upper Half Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIV{W|D}	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REM{W|D}	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMU{W|D}	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV{64,128}	
Load	Load Reserved	R	LR.W	rd,rs1	LR.{D|Q}	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.{D|Q}	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.{D|Q}	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.{D|Q}	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.{D|Q}	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.{D|Q}	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.{D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.{D|Q}	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.{D|Q}	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.{D|Q}	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.{D|Q}	rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP FI Pt)		+RV{64,128}	
Move	Move from Integer	R	FMV.{H|S}.X	rd,rs1	FMV.{D|Q}.X	rd,rs1
	Move to Integer	R	FMV.X.{H|S}	rd,rs1	FMV.X.{D|Q}	rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W	rd,rs1	FCVT.{H|S|D|Q}.{L|T}	rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU	rd,rs1	FCVT.{H|S|D|Q}.{L|T}U	rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q}	rd,rs1	FCVT.{L|T}.{H|S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q}	rd,rs1	FCVT.{L|T}U.{H|S|D|Q}	rd,rs1
Load	Load	I	FL{W,D,Q}	rd,rs1,imm		
Store	Store	S	FS{W,D,Q}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D|Q}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D|Q}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D|Q}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D|Q}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D|Q}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D|Q}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D|Q}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D|Q}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D|Q}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D|Q}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D|Q}	rd,rs1,rs2		
Compare	Compare Float =	R	FEQ.{S|D|Q}	rd,rs1,rs2		
	Compare Float <	R	FLT.{S|D|Q}	rd,rs1,rs2		
	Compare Float ≤	R	FLE.{S|D|Q}	rd,rs1,rs2		
Categorization	Classify Type	R	FCLASS.{S|D|Q}	rd,rs1		
Configuration	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

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

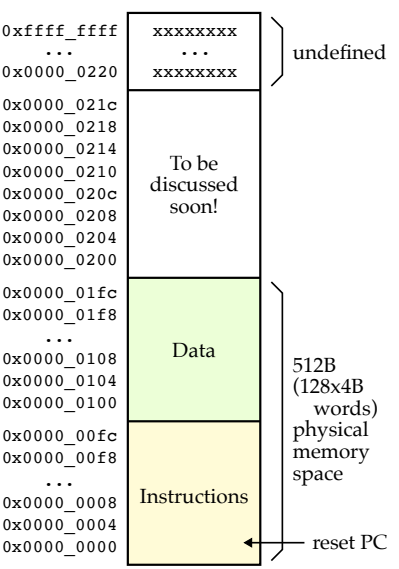
RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, {} means set, so `L{D|Q}` *is both* `LD` *and* `LQ`*. See risc.org. (8/21/15 revision)*

3. TinyRV1 Assembly Programming

- Assembler directives and labels enable more productive assembly programming
- Memory-mapped input/output (I/O) enable assembly programs to interact with external devices
- Conditionals, loops, arrays, and functions enable more complex assembly programs
- Compilers, assemblers, linkers, and loaders enable automatically transforming programs written in high-level languages into machine programs running on a computer

3.1. Assembler Directives and Labels

- Assembler directives tell the assembler how to assemble the program
- Assembly program is divided into many sections
  - `.text` directive specifies section for instructions (default section)
  - `.data` directive specifies section for global data
- `.word` directive specifies 4B of global data should be loaded at the corresponding memory location instead of an instruction



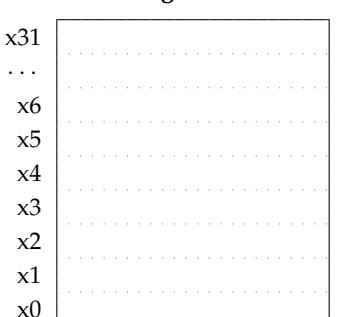
```
.text
  addi x1, x0, 256
  addi x2, x0, 260
  addi x3, x0, 264

  lw   x4, 0(x1)
  lw   x5, 0(x2)
  add  x6, x4, x5
  sw   x6, 0(x3)

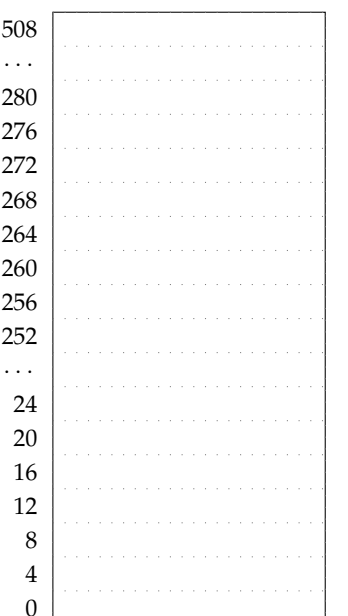
.data
  .word 7
  .word 9
  .word 0
```

- Before executing assembly program
  - Reset PC to zero (if PC is shown)
  - Make sure `x0` is zero
  - Load the program (`.text` section) into memory starting at address 0
  - Load the data (`.data` section) into memory starting at address 256

Registers



Memory

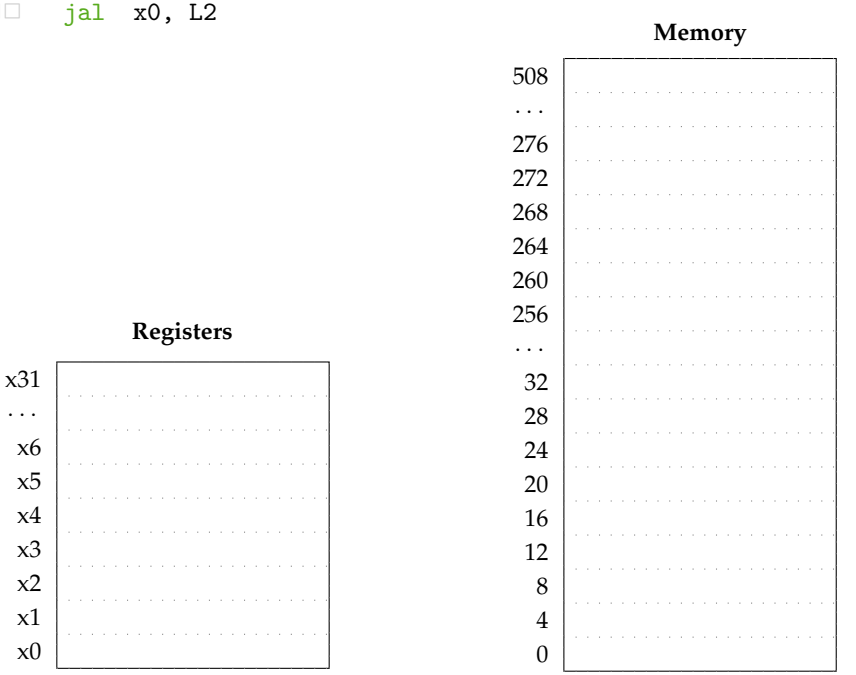


```
    addi x1, x0, 0
    addi x2, x0, 0
    addi x3, x0, 0
    jal  x0, L1
    addi x4, x0, 0

L2:
    addi x1, x0, 1
    addi x2, x0, 1

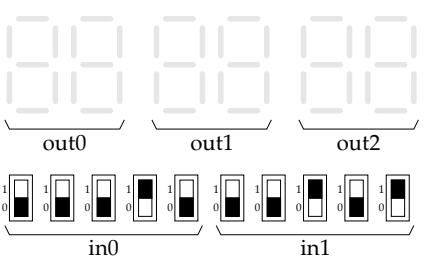
L1:
    addi x3, x0, 1
    jal  x0, L2
```

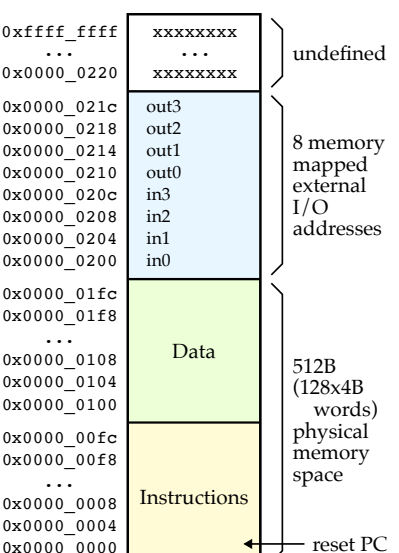
- Assembler labels ...
  - instead of absolute target addresses
  - are placed at beginning of line with colon
  - indicate instruction location in program
  - are not "executed" (no X)
  - do not take up any space in memory
  - assembler turns into PC-relative offsets



3.2. Memory-Mapped I/O

- Processor needs some way interact with the external enviroment
  - Receiving data from switches, buttons, keyboard, touch screen, network
  - Sending data to a seven-segment display, screen, speaker, network
- Memory-mapped input/output (I/O) involves choosing special memory addresses which are "mapped" to external devices
  - Loading or storing to these special memory-mapped I/O addresses does *not* read or write physical memory
  - Loading from a memory-mapped I/O address reads data from an external device (e.g., reads current value from switches)
  - Storing to a memory-mapped I/O address writes data to an external device (e.g., writes value to seven-segment display)
- TinyRV1 embedded system
  - in0 & in1 connected to switches
  - in2 connected to push button
  - in3 connected to breadboard pin
  - out0, out1, out2 connected to seven-segment displays
  - out3 connected to breadboard pin



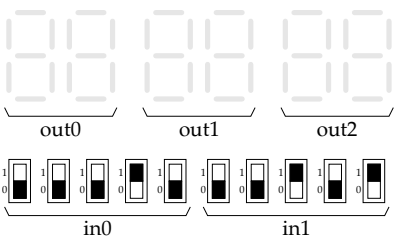


```
lw  x1, 0x200(x0)  # x1 = in0
lw  x2, 0x204(x0)  # x2 = in1

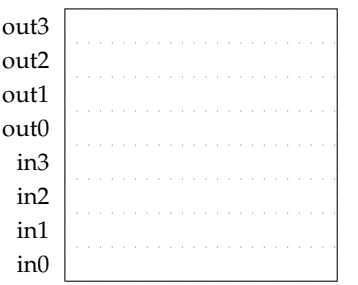
sw  x1, 0x210(x0)  # out0 = x1
sw  x2, 0x214(x0)  # out1 = x2

add x3, x1, x2     # x3 = x1 + x2

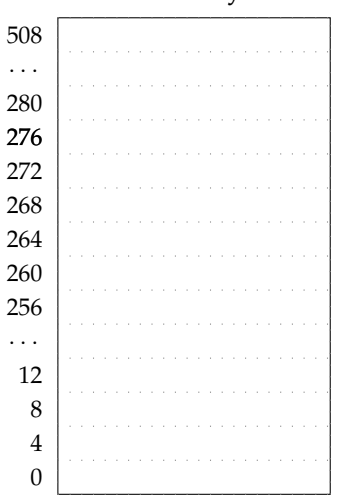
sw  x3, 0x218(x0)  # out2 = x3
```



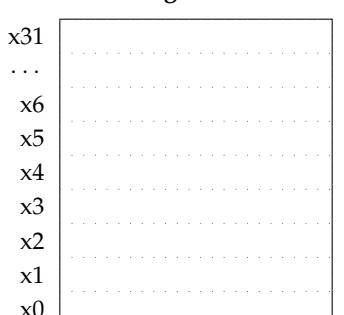
External I/O



Memory



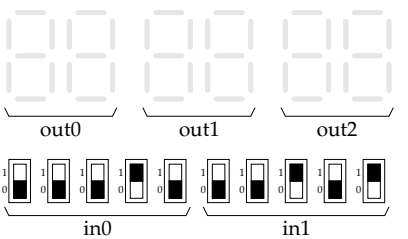
Registers



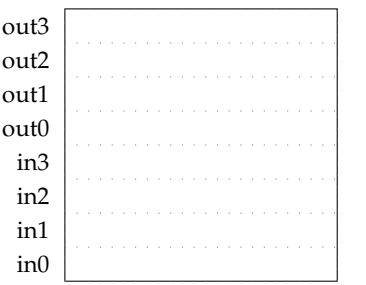
```
lw  x1, 0x200(x0)
sw  x1, 0x210(x0)

add x2, x1, 1
add x3, x1, -1

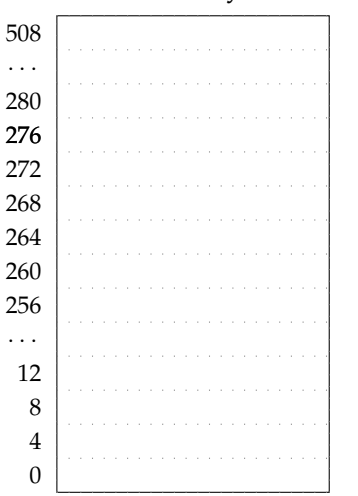
sw  x2, 0x214(x0)
sw  x3, 0x218(x0)
```



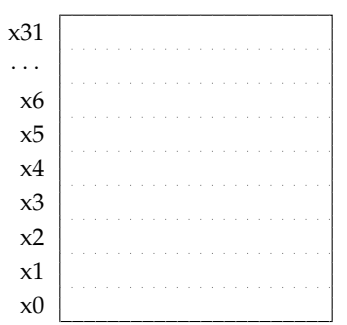
External I/O



Memory



Registers



3.3. Conditionals

```
1 out0 = in0
2 is_zero = 0
3 if ( in0 == 0 ):
4   is_zero = 1
5 out1 = is_zero
```

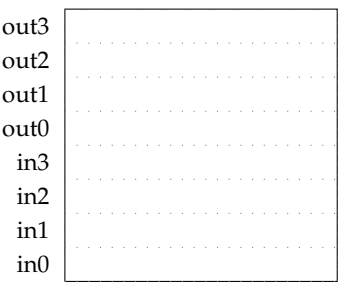
```
  lw   x1, 0x200(x0)  # read in0
  sw   x1, 0x210(x0)  # out0 = in0

  addi x2, x0, 0      # is_zero = 0

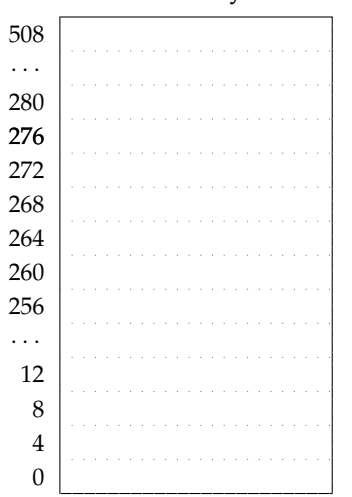
  bne  x1, x0, L1     # if statement
  addi x2, x0, 1      # is_zero = 1
L1:

  sw   x2, 0x214(x0)  # out1 = is_zero
```

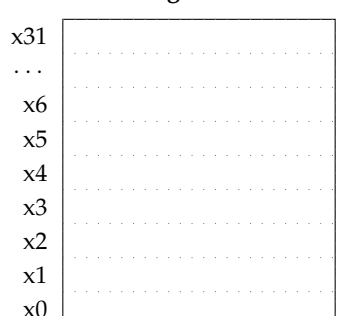
External I/O



Memory



Registers



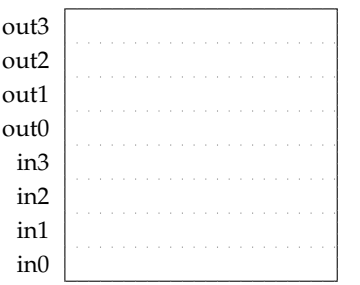
```
out0 = in0
if ( in0 == 0 ):
  is_zero = 1
else:
  is_zero = 0
out1 = is_zero
```

```
  lw   x1, 0x200(x0)
  sw   x1, 0x210(x0)

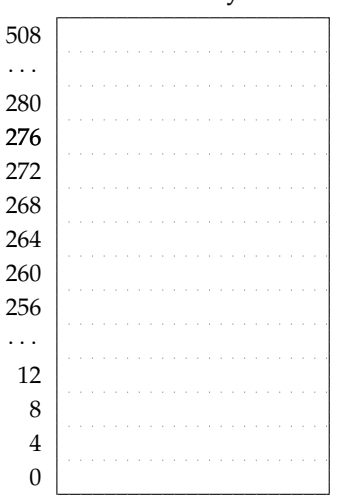
  bne  x1, x0, L1
  addi x2, x0, 1
  jal  x0, L2
L1:
  addi x2, x0, 0

L2:
  sw   x2, 0x214(x0)
```

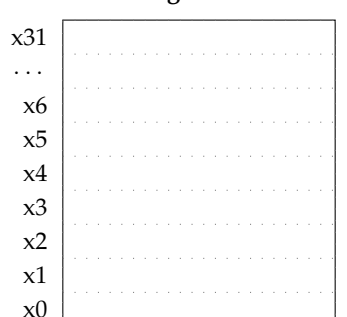
External I/O



Memory



Registers



3.4. Loops

```
1 out0 = in0
2 stop = in0 + 1
3 fact = 1
4 for i in range(1,stop):
5   fact = fact * i
6 out1 = fact
```

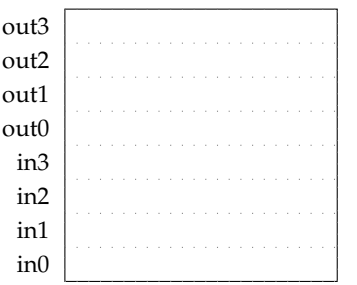
```
  lw   x1, 0x200(x0)  # read in0
  sw   x1, 0x210(x0)  # out0 = in0

  addi x2, x1, 1      # stop = in0+1
  addi x3, x0, 1      # fact = 1
  addi x4, x0, 1      # i = 1

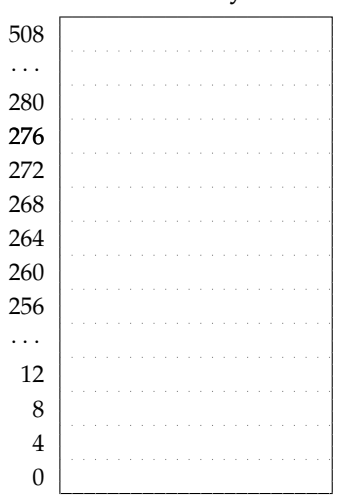
loop:
  mul  x3, x3, x4     # fact = fact*i
  addi x4, x4, 1
  bne  x4, x2, loop

  sw   x3, 0x214(x0)  # out1 = fact
```

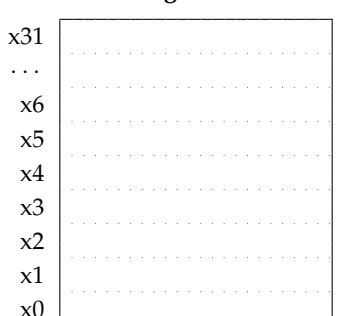
External I/O



Memory



Registers



```
out0 = in0
out1 = in1
pow  = 1
for i in range(in1):
  pow = pow * in0
out1 = pow
```

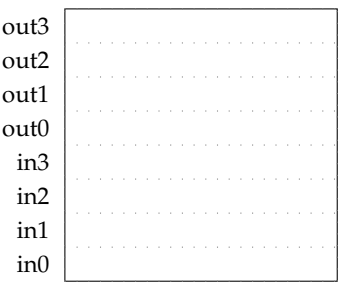
```
  lw   x1, 0x200(x0)
  lw   x2, 0x204(x0)
  sw   x1, 0x210(x0)
  sw   x2, 0x214(x0)

  addi x3, x0, 1

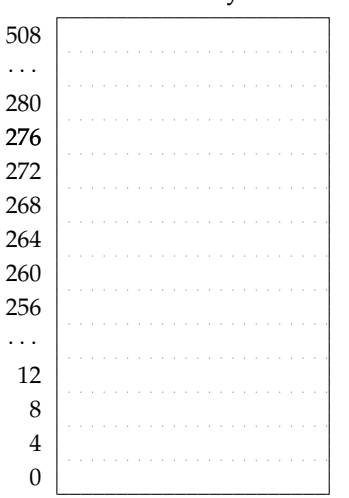
loop:
  mul  x3, x3, x1
  addi x2, x2, -1
  bne  x2, x0, loop

  sw   x3, 0x218(x0)
```

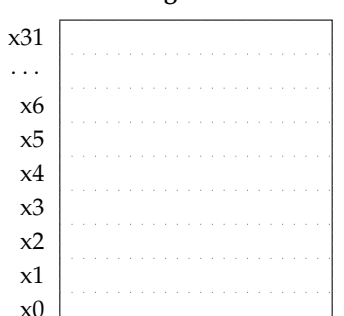
External I/O



Memory



Registers



3.5. Arrays

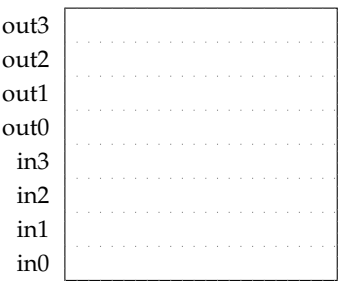
```
1 src = [ 1, 2, 3 ]
2 len = 3
3 for i in range(len):
4   out[i] = src[i] + in0
```

```
  lw   x1, 0x200(x0)  # read in0
  addi x2, x0, 256    # src array
  addi x3, x0, 0x210  # out array
  addi x4, x0, 0      # i = 0
  addi x5, x0, 12     # len = 3

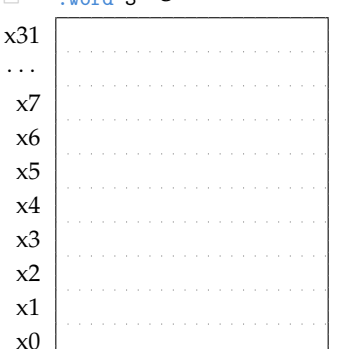
loop:
  addi x6, x2, x4     #
  lw   x7, 0(x6)      # x7 = src[i]
  add  x7, x7, x1     # x7 = x7 + in0
  addi x6, x3, x4     #
  sw   x7, 0(x6)      # out[i] = x7
  addi x4, x4, 4      # i = i + 1
  bne  x4, x5, loop

.data
  .word 1
  .word 2
  .word 3
```

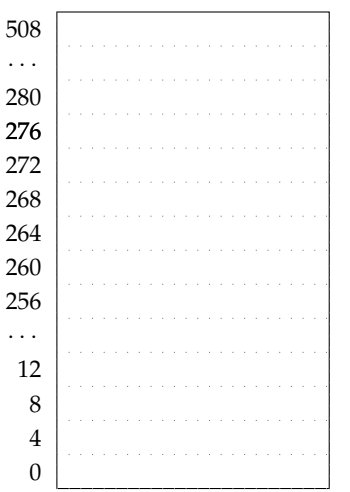
External I/O



Registers



Memory



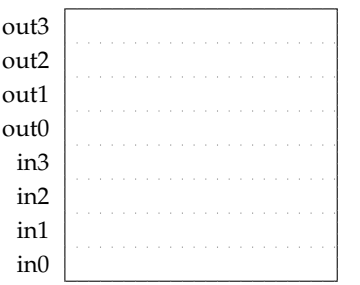
```
src = [ 1, 2, 3 ]
len = 3
for i in range(len):
  out[i] = src[i] + in0
```

```
  lw   x1, 0x200(x0)  # read in0
  addi x2, x0, 256    # src ptr
  addi x3, x0, 0x210  # out ptr
  addi x4, x0, 3      # len = 3

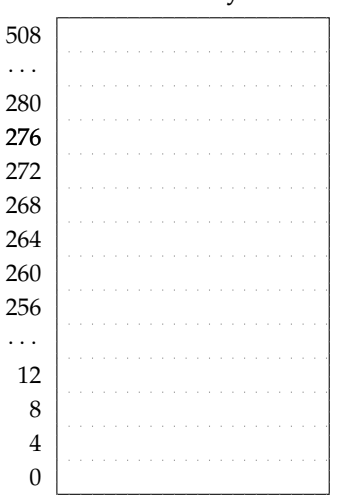
loop:
  lw   x5, 0(x2)      # x7 = src[i]
  add  x5, x5, x1     # x7 = x7 + in0
  sw   x5, 0(x3)      # out[i] = x7
  addi x2, x2, 4      # src ptr bump
  addi x3, x3, 4      # out ptr bump
  bne  x4, x4, -1
  bne  x4, x0, loop

.data
  .word 1
  .word 2
  .word 3
```

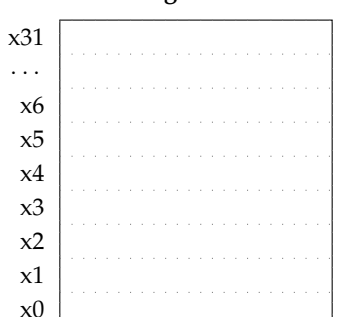
External I/O



Memory



Registers



3.6. Functions

```
1 def sum( a, b ):
2   result = a + b
3   return result
4
5 out0 = sum( in0, 2 )
6 out1 = sum( in1, 3 )
```

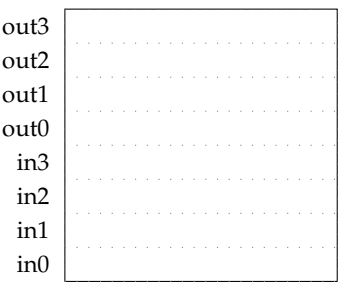
```
  jal  x0, start

sum:
  add  x3, x1, x2    # result = a + b
  jr   x4            # return result

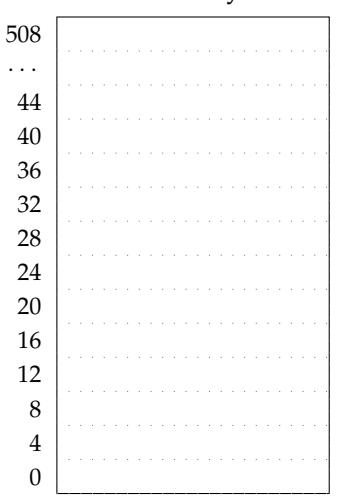
start:
  lw   x1, 0x200(x0) # x1 = in0
  addi x2, x0, 2     # x2 = 2
  jal  x4, sum       # tmp = sum(x1,x2)
  sw   x3, 0x210(x0) # out0 = tmp

  lw   x1, 0x204(x0) # x1 = in0
  addi x2, x0, 3     # x2 = 2
  jal  x4, sum       # tmp = sum(x1,x2)
  sw   x3, 0x214(x0) # out1 = tmp
```

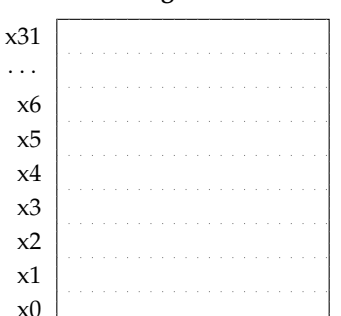
External I/O



Memory



Registers



- How do we know ...
  - what registers are used for arguments? return values? return address?
  - what registers the function can use as temporaries?
  - what registers the function must save to memory before using?
  - what memory locations can be used for temporary data?
- Calling convention is standard interface for calling functions

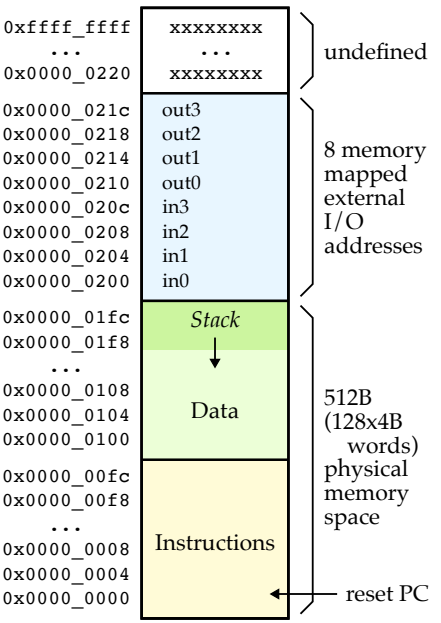
Register	Saver	Description
x28–x31	caller	Temporary values
x18–x27	callee	Saved registers
x11–x17	caller	Function arguments
x10	caller	Function arg & return value
...		
x5–x7	caller	Temporary values
...		
x2	callee	Stack pointer
x1	caller	Function return address
x0	—	Always zero

```
sum:
  add  x10, x11, x12   # result = a + b
  jr   x1              # return result

start:
  lw   x11, 0x200(x0)  # x11 = in0
  addi x12, x0, 2      # x12 = 2
  jal  x1,  sum        # tmp = sum(x11,x12)
  sw   x10, 0x210(x0)  # out0 = tmp

  lw   x11, 0x204(x0)  # x11 = in0
  addi x12, x0, 3      # x12 = 2
  jal  x1,  sum        # tmp = sum(x11,x12)
  sw   x10, 0x214(x0)  # out1 = tmp
```

- Stack is memory region dedicated for functions to use as scratch space (i.e., for temporary data)
  - Stack expands (uses more memory) as program needs more scratch space
  - Stack contracts (uses less memory) as program needs less scratch space
  - In TinyRV1, stack is located at address 0x1fc and expands "downwards"
- Stack pointer is a register that points to the top of the stack
  - Functions can allocate temporary data on the stack and then decrement the stack pointer so the next function knows where it can allocate its own temporary data
  - Before a function returns it needs to deallocate all of its stack data by incrementing the stack pointer back to its original value



We will use a "vector-vector-add" function throughout the course as a simple microbenchmark for performance analysis

Pseudo-Code

```
def vvadd( dest, src0, src1, size ):
  for i in range(size):
    dest[i] = src0[i] + src1[i]
```

Assembly Program

```
  jal x0, start

# wait_button()

wait_button:
  addi x5, x0, 1
wait_posedge:
  lw   x6, 0x208(x0)
  bne  x6, x5, wait_posedge
wait_negedge:
  lw   x6, 0x208(x0)
  bne  x6, x0, wait_negedge
  jr   x1

# vvadd( dest, src0, src1, size )

vvadd:
loop:
  lw    x5,  0(x11)   # x5 = src0[i]
  lw    x6,  0(x12)   # x6 = src1[i]
  add   x7,  x5, x6   # x7 = x5 + x6
  sw    x7,  0(x10)   # dest[i] = x7
  addi  x11, x11, 4   # src0 ptr bump
  addi  x12, x12, 4   # src1 ptr bump
  addi  x10, x10, 4   # src2 ptr bump
  addi  x13, x13, -1
  bne   x13, x0, loop
  jr    x1            # return

start:
  # setup vvadd arguments
  addi  x11, x0, 256
  addi  x12, x0, 370
  addi  x10, x0, 454
  addi  x13, x0, 21

  # call vvadd and trigger
  # breadboard pin for timing
  addi  x18, x0, 1
  sw    x18, 0x21c(x0)
  jal   x1,  vvadd
  sw    x0,  0x21c(x0)

  # display results
  addi  x18, x0, 454
  addi  x19, x0, 21
loop2:
  lw    x20, 0(x18)
  sw    x20, 0x210(x0)
  jal   x1,  wait_button
  addi  x18, x18, 4
  addi  x19, x19, -1
  bne   x19, x0, loop2

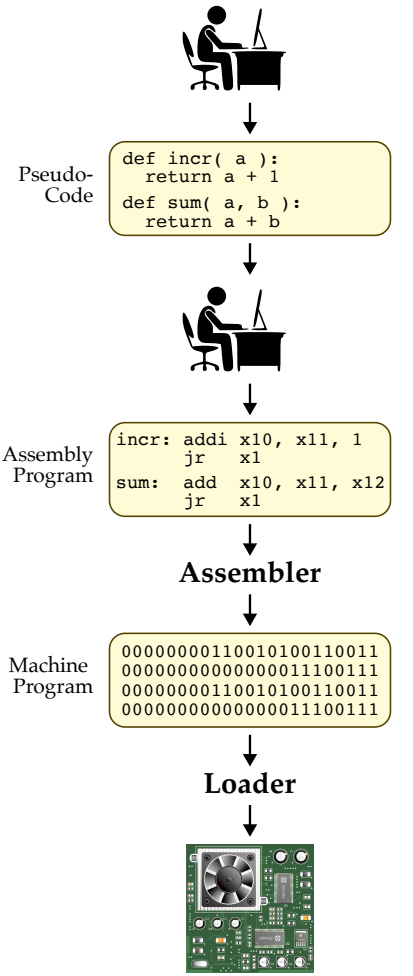
done:
  jal   x0,  done

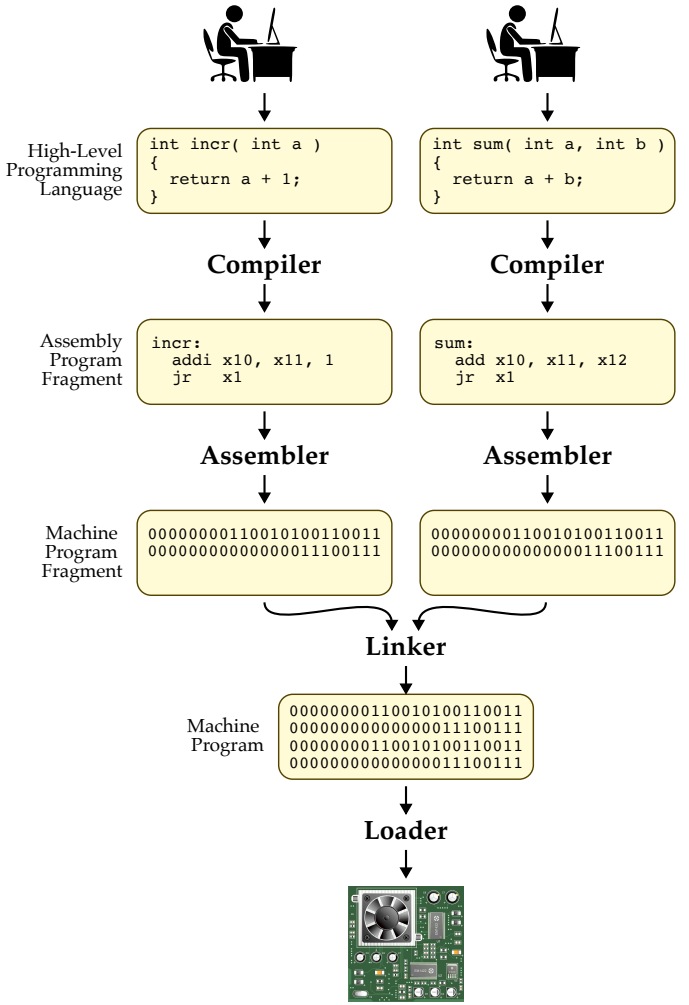
.data
  # src0 @256, 21 elm
  .word 3
  .word 2
  ...

  # src1 @370, 21 elm
  .word 7
  .word 4
  ...

  # dest @454, 21 elm
  .word 0
  .word 0
  ...
```

3.7. Compilers, Assemblers, Linkers, and Loaders




High-Level Programming Language
int incr(int a)
{
return a + 1;
}
int sum(int a, int b)
{
return a + b;
}
Compiler
Compiler
Assembly Program Fragment
incr:
addi x10, x11, 1
jr x1
sum:
add x10, x11, x12
jr x1
Assembler
Assembler
Machine Program Fragment
00000000110010100110011
00000000000000011100111
00000000110010100110011
00000000000000011100111
Linker
Machine Program
00000000110010100110011
00000000000000011100111
00000000110010100110011
00000000000000011100111
Loader

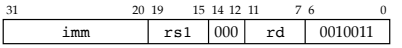
3.8. TinyRV1 ISA Specification

ADDI

`addi rd, rs1, imm`

R[rd] ← R[rs1] + sext(imm)

PC ← PC + 4



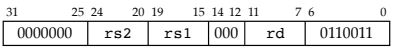
I-type Immediate

ADD

`add rd, rs1, rs2`

R[rd] ← R[rs1] + R[rs2]

PC ← PC + 4

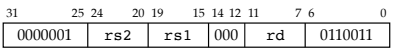


MUL

`mul rd, rs1, rs2`

R[rd] ← R[rs1] × R[rs2]

PC ← PC + 4

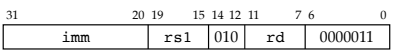


LW

`lw rd, imm(rs1)`

R[rd] ← M[R[rs1] + sext(imm)]

PC ← PC + 4



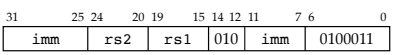
I-type Immediate

SW

`sw rs2, imm(rs1)`

M[R[rs1] + sext(imm)] ← R[rs2]

PC ← PC + 4



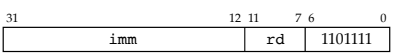
S-type Immediate

JAL

`jal rd, addr`

R[rd] ← PC + 4

PC ← addr



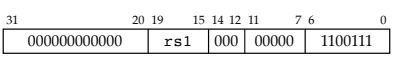
J-type Immediate

addr encoded as PC + sext(imm)

JR

`jr rs1`

PC ← R[rs1]

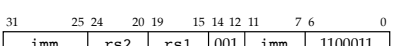


BNE

`bne rs1, rs2, addr`

if (R[rs1] != R[rs2]) PC ← addr

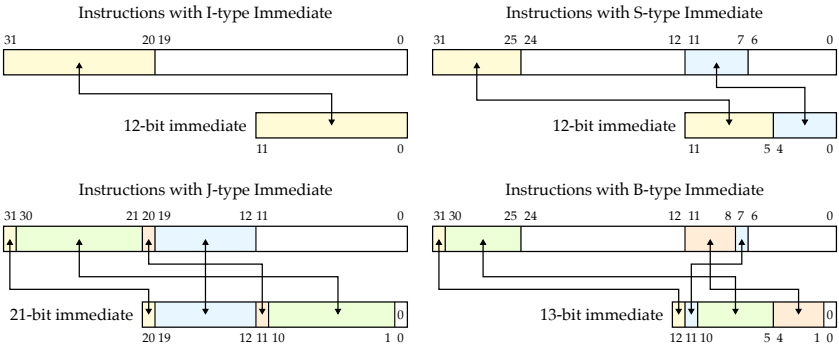
else PC ← PC + 4



B-type Immediate

addr encoded as PC + sext(imm)

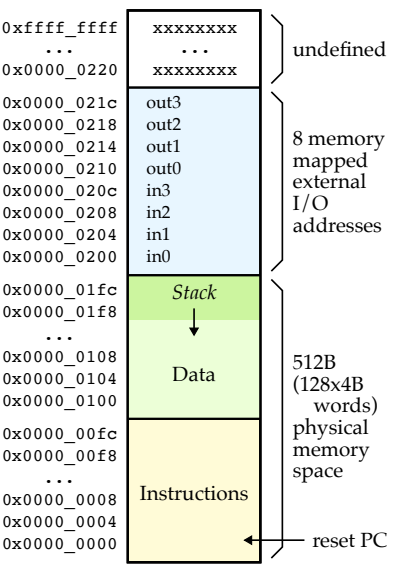
TinyRV1 Immediate Types



TinyRV1 Calling Convention

Register	Saver	Description
`x28–x31`	caller	Temporary values
`x18–x27`	callee	Saved registers
`x11–x17`	caller	Function arguments
`x10`	caller	Function arg & return value
...		
`x5–x7`	caller	Temporary values
...		
`x2`	callee	Stack pointer
`x1`	caller	Function return address
`x0`	—	Always zero

TinyRV1 Virtual Address Map

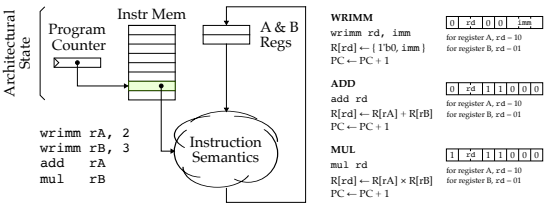


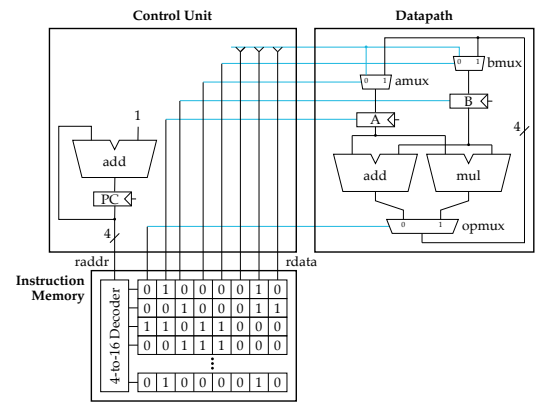
Notes

- R[`x0`] is always zero
- PC reset value is `0x00000000`
- All addresses must be word aligned (i.e., bottom two address bits are zero)
- Unaligned or invalid memory accesses result in undefined behavior
- `nop` is pseudo-instruction for `addi x0, x0, 0`

4. From Architecture to Microarchitecture

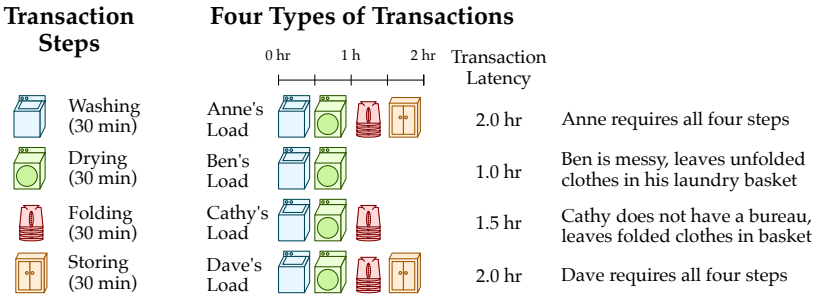
- Processor
  - Instructions are "transactions" that execute on a processor
  - Architecture: defines the hardware/software interface
  - Microarchitecture: how hardware executes sequence of instructions



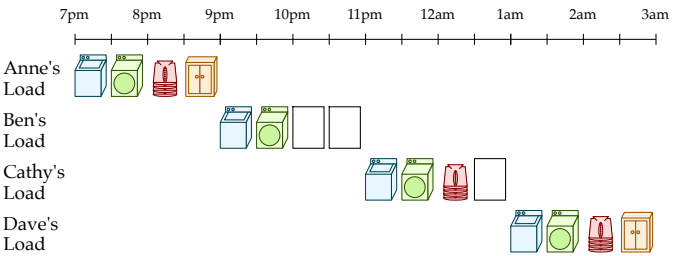


- Laundry
  - Cleaning a load of laundry is a "transaction"
  - Architecture: high-level specification, dirty clothes in, clean clothes out
  - Microarchitecture: how laundry room actually processes multiple loads

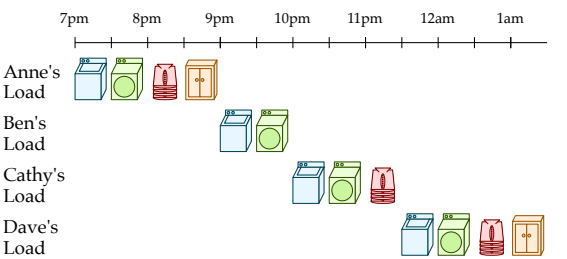
4.1. Processor Microarchitectural Design Patterns



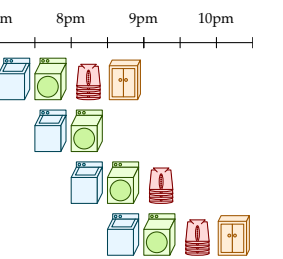
Fixed Time Slot Laundry (Single-Cycle Processors)



Variable Time Slot Laundry (Multi-Cycle Processors)

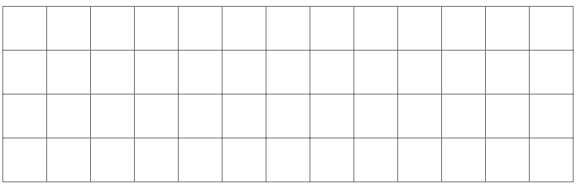


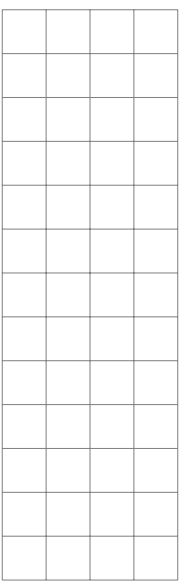
Pipelined Laundry



4.2. Transaction Diagrams

 **W**: Washing  **D**: Drying  **F**: Folding  **S**: Storing





Key Concepts

- Transaction latency is the time to complete a single transaction
- Execution time or total latency is the time to complete a sequence of transactions
- Throughput is the number of transactions executed per unit time

4.3. Analyzing Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

	Cycles	Time	Time	
Microarchitecture	**Instruction**	**Cycle**	**Program**	**Area**
Single-Cycle Processor	1	long	medium	medium
Multi-Cycle Processor	>1	short	high	low
Pipelined Processor	≈1	short	low	high

- Estimate the dynamic instruction count for the following vector-vector-add function assuming `n` is 21?

```
1  loop:
2    lw    x5,  0(x11)
3    lw    x6,  0(x12)
4    add   x7,  x5, x6
5    sw    x7,  0(x10)
6    addi  x11, x11, 4
7    addi  x12, x12, 4
8    addi  x10, x10, 4
9    addi  x13, x13, -1
10   bne   x13, x0, loop
11   jr    x1
```