Tahmini Ders İçeriği
(Tentative Couse Schedule – Syllabus)

1. Hafta: Sayı sistemleri, onluk/ikilik taban sayı gösterimleri, mantıksal kapılar, computer system overview, başarım (performance)

2. Hafta: 2'lik tabanda işaretli sayılar, mikroişlemci tarihi, benchmarking, başarım,

3. Hafta: Başarım, Amdahl yasası, RISC-V development Environment, Verilog HDL ile Birleşik (Combinational) devreler

4. Hafta:, Verilog HDL ile sıralı (sequential) mantıksal devre ve sonlu durum makinası tasarımı, timing analysis

5. Hafta: Aritmetik devre tasarımları: Toplama, çıkarma, çarpma, bölme, trigonometri, square-root, hyperbolic, exponential, logarithm

6. Hafta: Fixed ve Floating-Point sayı gösterimleri

7. Hafta: RISC-V buyruk kümesi mimarisi (ISA) ve buyrukların tanıtımı

8. Hafta: RISC-V buyruk kümesi mimarisi (ISA) ve buyrukların tanıtımı

9. Hafta: Tek-çevrim işlemci tasarımı (single-cycle CPU)

10. Hafta: Çok-çevrim işlemci tasarımı (multi-cycle CPU)

11. Hafta: Boruhatlı işlemci tasarımı (pipelined CPU)

12. Hafta: Bellek sistemi ve hiyerarşisi

13. Hafta: İleri mimari konuları: Branch prediction, superscalar cpu, out-of-order execution, multi-core systems

14. Hafta: Gömülü sistemler, mikrodenetleyiciler, SoCs

RISC-V GP Registers

31 … 0	
x0 / zero	Hardwired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
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
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary

32

```
addi a0,zero,5
addi a1,zero,7
lui s2,0x1000
add a2,a0,a1
sw a2,4(s2)
lw a3,4(s2)
nop

addi x10,x0,5
addi x11,x0,7
lui x18,0x1000
add x12,x10,x11
sw x12,4(x18)
lw x13,4(x18)
addi x0,x0,0
```

RISC-V GP Registers

Register Name(s)	Usage
x0/zero	Always holds 0
ra	Holds the return **address**
sp	Holds the address of the boundary of the stack
t0-t6	Holds temporary values that **do not** persist after function calls
s0-s11	Holds values that persist after function calls
a0-a1	Holds the first two arguments to the function or the return values
a2-a7	Holds any remaining arguments

RISC-V GP Registers

```
def foo ():
    x = 1
    bar ()
    z = 2

def bar ():
    y = 7
```

bar fonksiyonu içerisinde y = 7; satırında olduğumuzu düşünelim.

Bu satır işlendikten sonra foo fonksiyonu içerisinde z = 2; satırından devam edilmesi lazım.

x1 (ra) register, return adres olarak z = 2; ile ilgili instruction adresini tutar

x2 (sp) registerı, stack adresini tutar. Stack, her bir fonksiyon çağrıldığında değeri azalır, yani bellekte alanı azalır.

Fonksiyon tamamlanıp fonksiyondan çıkıldığında sp registerı ile fonksiyon çağrılmadan önceki duruma (state) geri dönülmüş olunur.

RV32I: RISC-V Base Integer ISA

Beyond being recent and open, RISC-V is unusual since, unlike almost all prior ISAs, it is modular. At the core is a base ISA, called RV32I, which runs a full software stack. RV32I is frozen and will never change, which gives compiler writers, operating system developers, and assembly language programmers a stable target. The modularity comes from optional standard extensions that hardware can include or not depending on the needs of the application. This modularity enables very small and low energy implementations of RISC-V, which can be critical for embedded applications. By informing the RISC-V compiler what extensions are included, it can generate the best code for that hardware. The convention is to append the extension letters to the name to indicate which are included. For example, RV32IMFD adds the multiply (RV32M), single-precision floating point (RV32F), and double-precision floating point extensions (RV32D) to the mandatory base instructions (RV32I) – (from rvbook)

RV32I: RISC-V Base Integer ISA

RV32I

Integer Computation

add { – / immediate }

subtract

{ and / or / exclusive or } { – / immediate }

{ shift left logical / shift right arithmetic / shift right logical } { – / immediate }

load upper immediate

add upper immediate to pc

set less than { – / immediate } { – / unsigned }

Control transfer

branch { equal / not equal }

branch { greater than or equal / less than } { – / unsigned }

jump and link { – / register }

Loads and Stores

{ load / store } { byte / halfword / word }

load { byte / halfword } unsigned

Miscellaneous instructions

fence loads & stores

fence.instruction & data

environment { break / call }

control status register { read & clear bit / read & set bit / read & write } { – / immediate }

31–25	24–20	19–15	14–12	11–7	6–0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi

0000000		shamt	rs1	001	rd	0010011	I slli
0000000		shamt	rs1	101	rd	0010011	I srli
0100000		shamt	rs1	101	rd	0010011	I srai
0000000		rs2	rs1	000	rd	0110011	R add
0100000		rs2	rs1	000	rd	0110011	R sub
0000000		rs2	rs1	001	rd	0110011	R sll
0000000		rs2	rs1	010	rd	0110011	R slt
0000000		rs2	rs1	011	rd	0110011	R sltu
0000000		rs2	rs1	100	rd	0110011	R xor
0000000		rs2	rs1	101	rd	0110011	R srl
0100000		rs2	rs1	101	rd	0110011	R sra
0000000		rs2	rs1	110	rd	0110011	R or
0000000		rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	000	00000	1110011	I ecall
000000000001			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I csrrsi
csr			zimm	111	rd	1110011	I csrrci

RISC-V Assembly Examples

31 … 0	
x0 / zero	Hardwired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
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
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary

32

C Code:

f = (g + h) – (i + j);

// f,g,h,i,j variables are stored in x18-x22

Compiled RISC-V Code:

add x5,x19,x20

add x6,x21,x22

sub x18,x5,x6

RISC-V Assembly Examples (Arithmetic)

C Code:

f = (g + h) – (i + j);

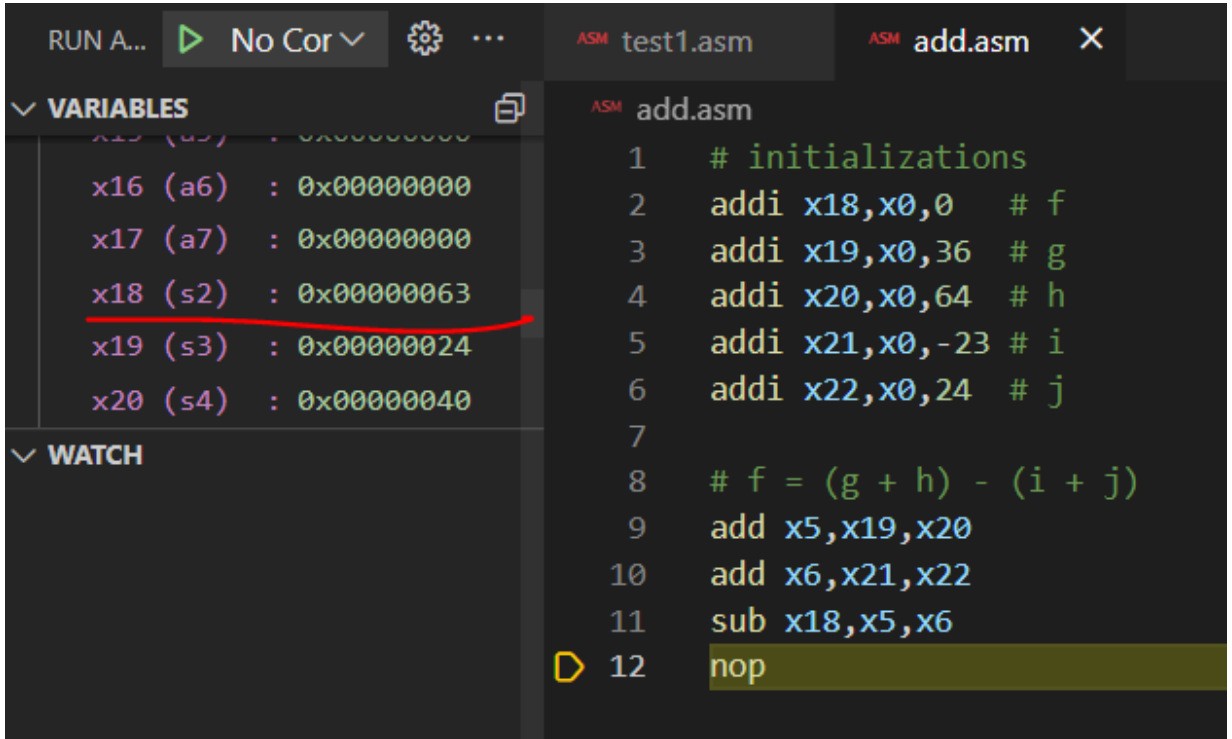
// f,g,h,i,j variables are stored in x18-x22

Compiled RISC-V Code:

add x5,x19,x20

add x6,x21,x22

sub x18,x5,x6



Arithmetic	ADD	R	ADD	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm
	Add Upper Imm to PC	U	AUIPC	rd,imm

RISC-V Assembly Examples (Memory Access)

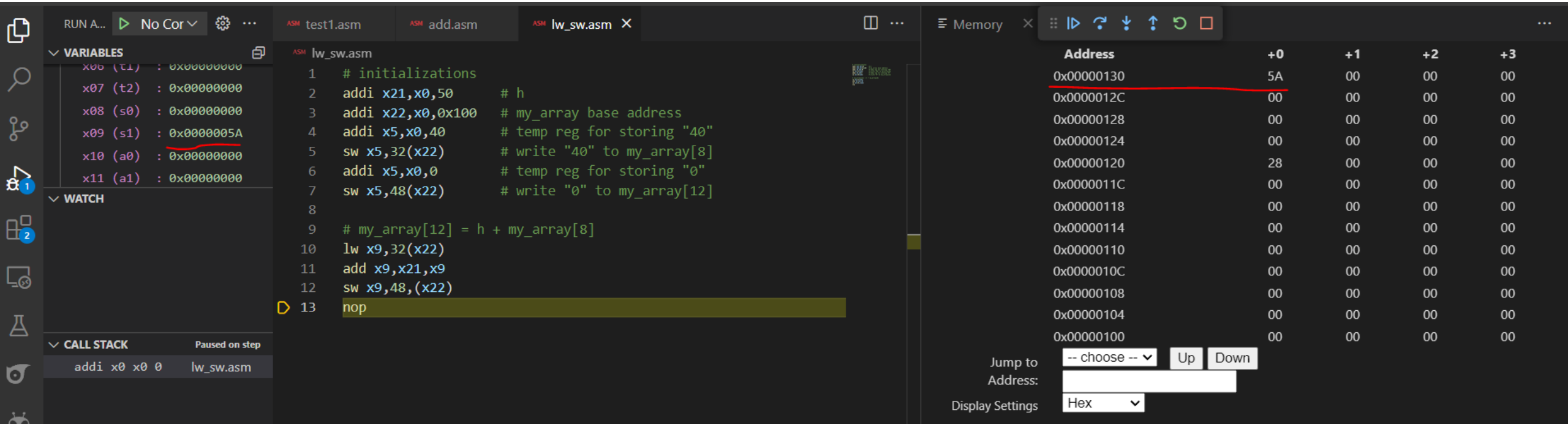
C Code:

```
int my_array[20];
int h;
my_array[12] = h  + my_array[8];
// my_array base address is in x22
// h is in x21
```

Compiled RISC-V Code:

```
lw x9,32(x22)
add x9,x21,x9
sw x9,48,(x22)
```

Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
	Load Word	I	LW	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



RISC-V Assembly Examples (Shift)

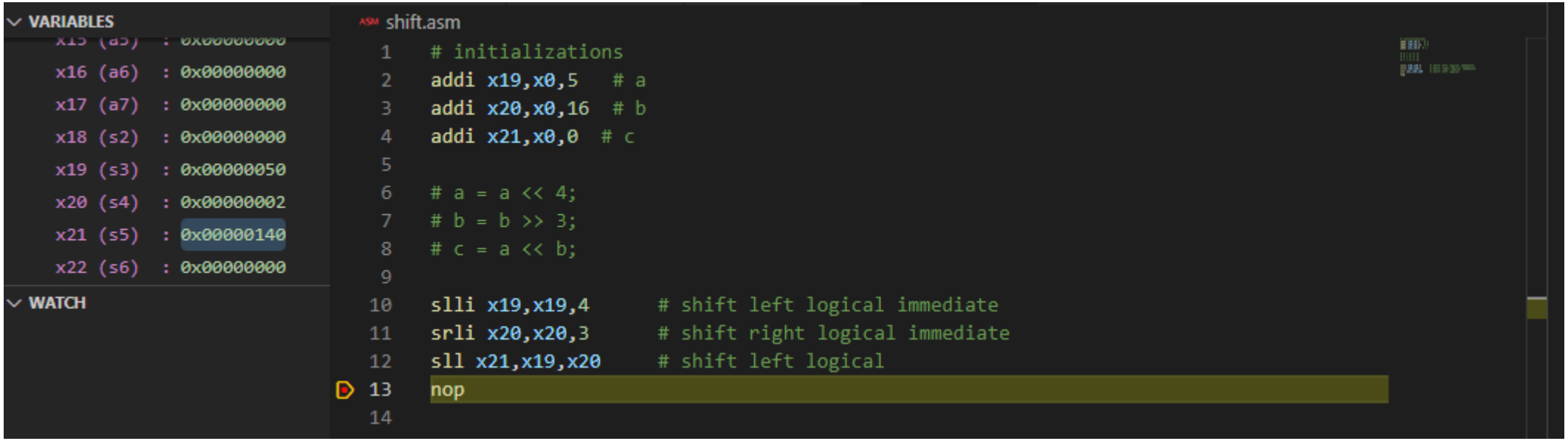
C Code:

```
int a,b,c;
a = a << 4;
b = b >> 3;
c = a << b;
// a is in x19, b in x20, c in x21
```

Compiled RISC-V Code:

```
slli x19,x19,4
slri x20,x20,3
sll x21,x19,x20
```

Shifts Shift Left Logical	R	SLL	rd,rs1,rs2
Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt
Shift Right Logical	R	SRL	rd,rs1,rs2
Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt
Shift Right Arithmetic	R	SRA	rd,rs1,rs2
Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt



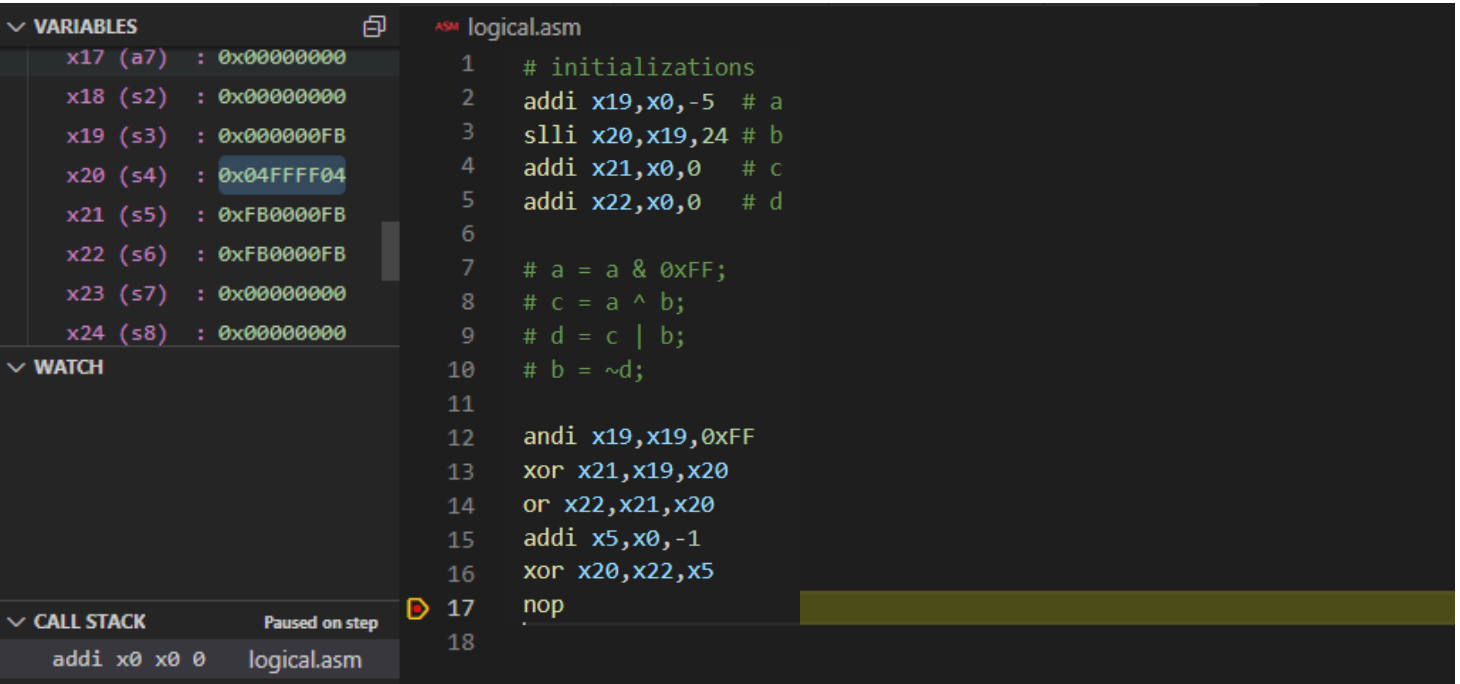
RISC-V Assembly Examples (Logical)

C Code:

```
int a,b,c,d;
a = a & 0xFF;
c = a ^ b;
d = c | b;
b = ~d;
// a is in x19, b in x20, c in x21, d in x22
```

Compiled RISC-V Code:

```
andi x19,x19,0xFF
xor x21,x19,x20
or x22,x21,x20
addi x5,x0,-1
xor x20,x22,x5
```



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

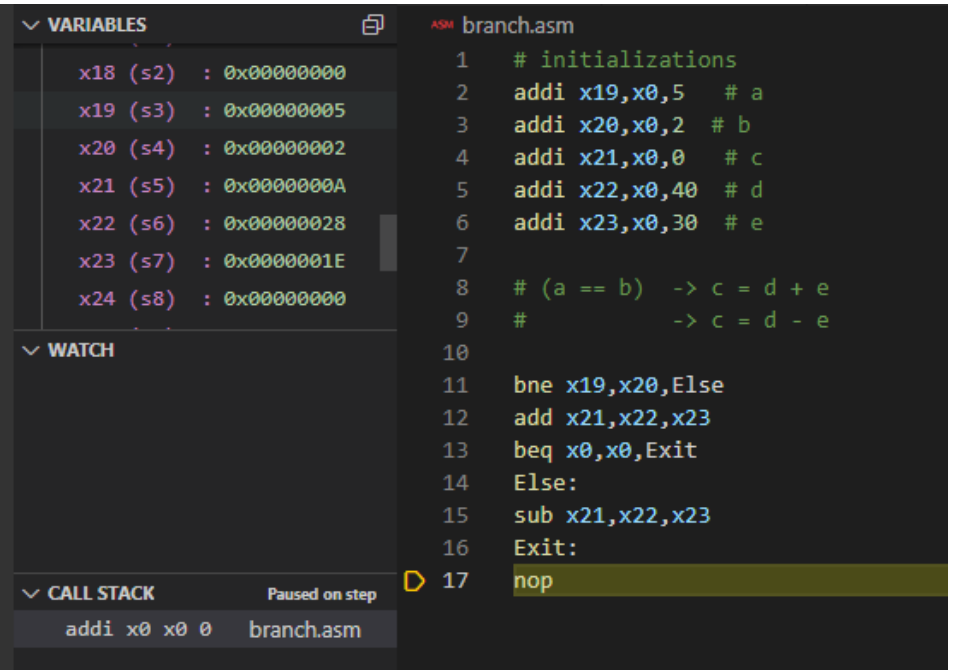
RISC-V Assembly Examples (Conditional)

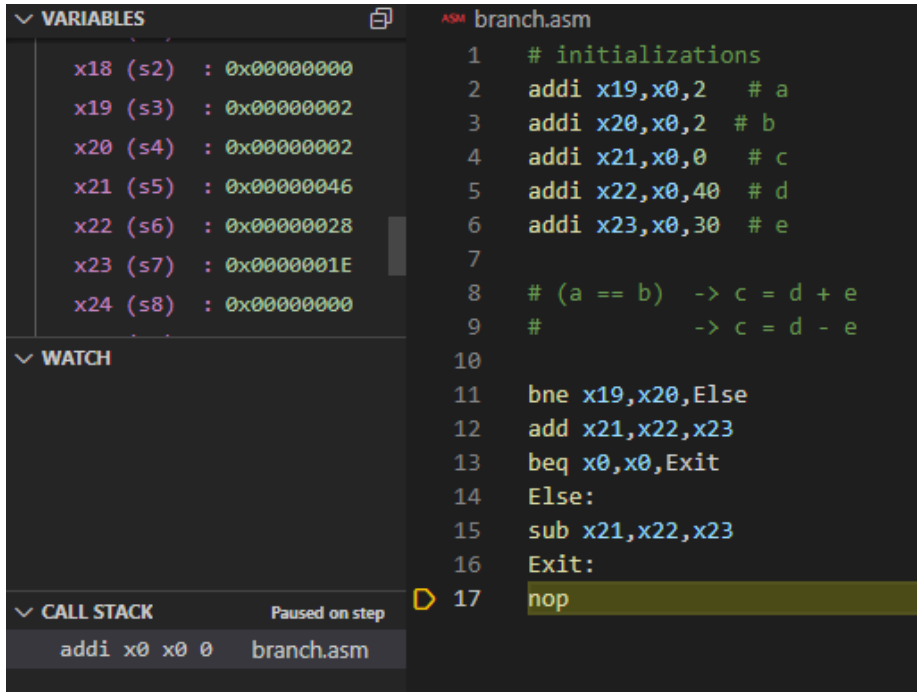
```
int a,b,c,d,e;
if (a == b)
   c = d + e;
else
   c = d - e;
// a x19, b x20, c x21, d x22, e x23
```

Compiled RISC-V Code:

```
bne x19,x20,Else
add x21,x22,x23
beq x0,x0,Exit
Else:
sub x21,x22,x23
Exit:
```

Branches	Branch =	B	BEQ	rs1,rs2,imm
	Branch ≠	B	BNE	rs1,rs2,imm
	Branch <	B	BLT	rs1,rs2,imm
	Branch ≥	B	BGE	rs1,rs2,imm
	Branch < Unsigned	B	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm





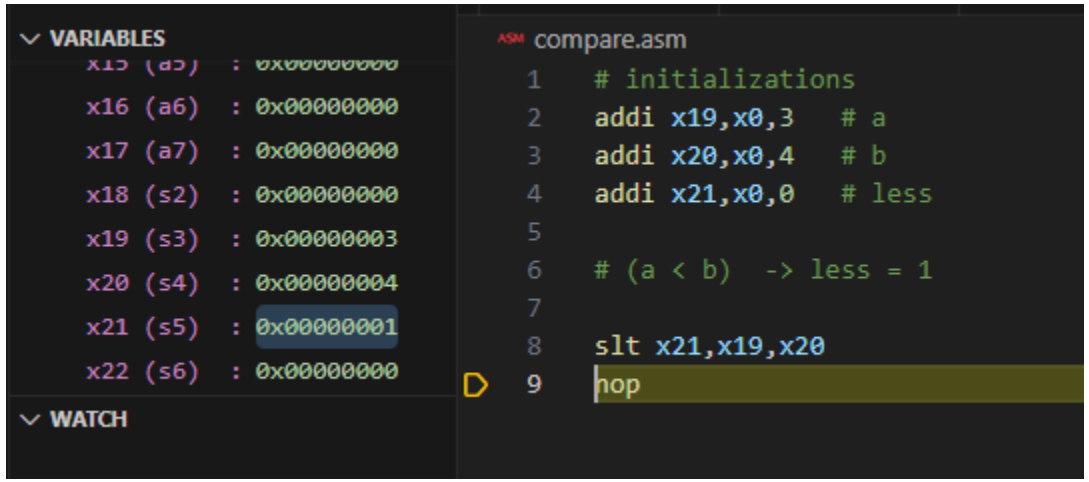
RISC-V Assembly Examples (Compare)

C Code:

```
int a,b,less;
if (a < b)
  less = 1;
// a x19, b x20, less x21
```

Compiled RISC-V Code:

```
slt x21,x19,x20
```



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

RISC-V Assembly Examples (While Loop)

C Code:

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x = 0;
while (pow != 128) {
pow = pow * 2;
x = x + 1;
}
```

Compiled RISC-V Code:

```
# s0 = pow, s1 = x
addi s0, zero, 1 # pow = 1
add s1, zero, zero # x = 0
addi t0, zero, 128 # t0 = 128
while: beq s0, t0, done # pow = 128?
slli s0, s0, 1 # pow = pow * 2
addi s1, s1, 1 # x = x + 1
j while # repeat loop
done:
```

RISC-V Assembly Examples (For Loop)

C Code:

```
// add the numbers from 0 to 9
int sum = 0;
int i;
for (i = 0; i < 10; i = i + 1) {
sum = sum + i;
}
```

Compiled RISC-V Code:

```
# s0 = i, s1 = sum
addi s1, zero, 0 # sum = 0
addi s0, zero, 0 # i = 0
addi t0, zero, 10 # t0 = 10
for: bge s0, t0, done # i >= 10?
add s1, s1, s0 # sum = sum + i
addi s0, s0, 1 # i = i + 1
j for # repeat loop
done:
```

RISC-V Procedures

Hemen her progamlama dilinde prosedürler (fonksiyonlar) mevcuttur.

Bir prosedür, kendisine girdi olarak aktarılan verileri alarak üzerinde işlemler gerçekleştirerek ortaya çıkan sonucu, prosedürü çağıran üst modülün erişebileceği belirli bir alana kaydeder.

RISC-V ISA'de 32 adet registerdan 8 adedi (x10-x17) prosedür parametreleri için ayrılmıştır

x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
x16 / a6	Function argument
x17 / a7	Function argument

1 adet özel register (x1/ra – return address) prosedürün çağrıldığı andaki instruction adresini tutar

Prosedür çağrıları için RV32I içerisinde 2 adet instruction mevcuttur: jump-and-link (jal) ve jal return (jalr)

jal instruction, PC değerine prosedürün olduğu adresi atar ve prosedürün çağrıldığı (caller) adresi kaydeder

jalr instruction, çağrılan prosedür (callee) bittiğinde PC değerine dönüş adresini atar

RISC-V Assembly Examples (Procedure Calls)

```
# Main program
_start:
    # Call the add_numbers procedure with inputs 5 and 10
    addi a0, x0, 5
    addi a1, x0, 10
    jal ra, add_numbers

    # Exit the program
    li a7, 10
    ecall
# Declare the procedure called "add_numbers"
add_numbers:
    addi s1, s0, 0  # Initialize a counter variable
    addi s2, s0, 0  # Initialize a sum variable

loop:
    add s2, s2, a0  # Add the input number to the sum
    addi s1, s1, 1  # Increment the counter

    blt s1, a1, loop  # If the counter is less than the second input number, jump to

    add a0, s2, x0  # Set the output number to the sum
    jr ra  # Return from the procedure
```

Jump & Link	J&L	J	JAL	rd,imm
	Jump & Link Register	I	JALR	rd,rs1,imm

RISC-V Procedures – Need for extra registers

RISC-V ISA'da 8 adet register (x10-x17) procedure çağrıları ve sonuçları için ayrılmıştır.

Peki bazı prosedürlerde bu 8 adet register yeterli gelmezse ne olacak? Prosedürün (callee) işi tamamlanıp return adrese geri dönüldüğünde, prosedürü çağıranın (caller) durumunun (state) eski haline gelmesi lazım.

Prosedür çağrıldığında 8 registerdan daha fazla ihtiyaç varsa compiler ana bellekte stack olarak ayrılan yeri kullanır.

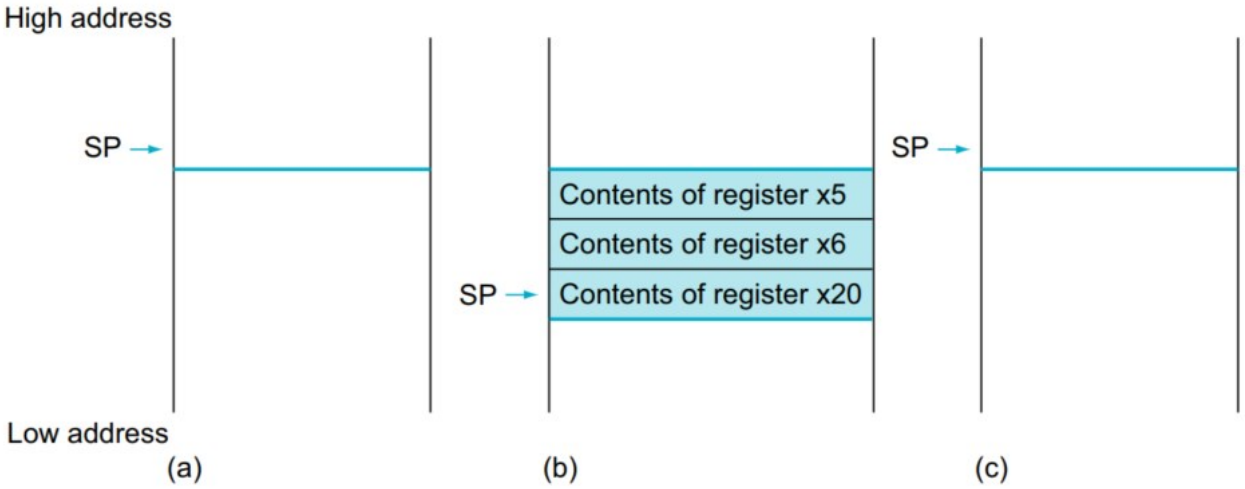
RISC-V ISA'da x2(sp – stack pointer) registeri stack adresini kaydeder. Stack'e kaydedilecek her bir ekstra register için push operasyonu, okuma için de pop operasyonu gerçekleşir.

Stack adresi yüksek bir değerden başlar (0x7FFFFFF0) ve her bir push işleminde -4 azaltılır.

RISC-V Assembly Examples (Procedure Calls)

C Code:

```
int leaf_example (int g, int h, int i, int j)
{
int f;
f = (g + h) – (i + j);
return f;
}
// g,h,i,j → x10-x13          f → x20
```



Compiled RISC-V Code:

```
leaf_example:
addi sp, sp, -12 // adjust stack to make room for 3 items
sw x5, 8(sp) // save register x5 for use afterwards
sw x6, 4(sp) // save register x6 for use afterwards
sw x20, 0(sp) // save register x20 for use afterwards
add x5, x10, x11 // register x5 contains g + h
add x6, x12, x13 // register x6 contains i + j
sub x20, x5, x6 // f = x5 – x6, which is (g + h) – (i + j)
addi x10, x20, 0 // returns f (x10 = x20 + 0)
lw x20, 0(sp) // restore register x20 for caller
lw x6, 4(sp) // restore register x6 for caller
lw x5, 8(sp) // restore register x5 for caller
addi sp, sp, 12 // adjust stack to delete 3 items
jalr x0, 0(x1) // branch back to calling routine
```

RISC-V Assembly Examples (Procedure Calls)

31 … 0	
x0 / zero	Hardwired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
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
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary
32	

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, RISC-V software separates 19 of the registers into two groups:

- x5–x7 and x28–x31: temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- x8–x9 and x18–x27: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller does not expect registers x5 and x6 to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore x20, since the callee must assume that the caller needs its value.

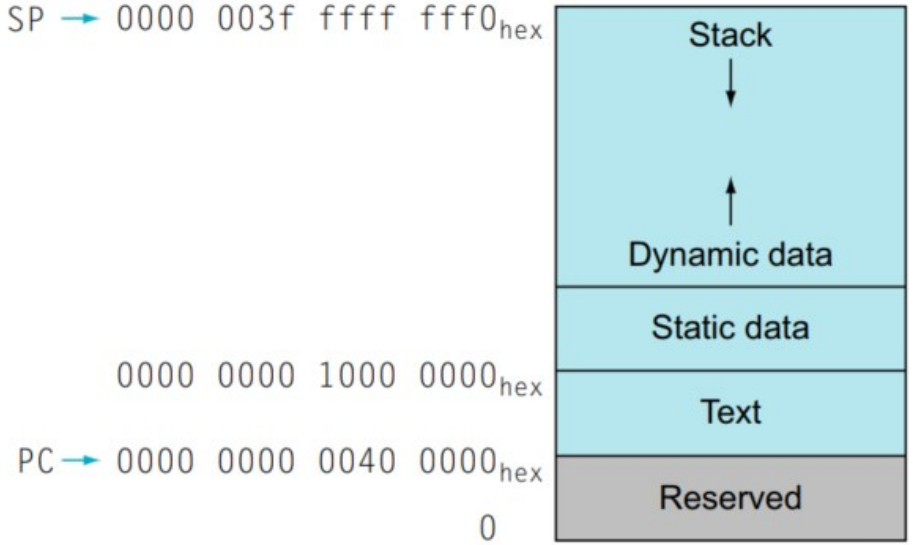
RISC-V Memory Allocation

Text: Çalıştırılabilir ve read-only binary veri, instructionları içerir. Sadece okunacağı için gömülü sistemlerde genelde ROM'a yüklenir.

Static Data: Sabitler (constants) ve diğer boyutu değişmeyen parametrelerin (initialized static variables) saklandığı yer. Yine sabit uzunluktaki array değişlenleri de burada saklanır.

Dynamic Data (Heap): Dynamic memory allocation, linked-list gibi yapılar. malloc, calloc, realloc, free etc.

Program çalıştırıldıkça Stack ve Heap birbirine doğru yaklaşır. "Stack overflow" gibi dinamik akış içerisinde hatalar almamak için program kodlanırken veya mimarisi kurulurken stack ve heap kapasiteleri dikkate alınmalıdır.



RISC-V – Instructionların Kullanım Sıklıkları

Instruction class	RISC-V examples	HLL correspondence	Frequency	
			Integer	Fl. Pt.
Arithmetic	add, sub, addi	Operations in assignment statements	16%	48%
Data transfer	lw, sw, lh, sh, lb, sb, lui	References to data structures in memory	35%	36%
Logical	and, or, xor, sll, srl, sra	Operations in assignment statements	12%	4%
Branch	beq, bne, blt, bge, bltu, bgeu	*If* statements; loops	34%	8%
Jump	jal, jalr	Procedure calls & returns; *switch* statements	2%	0%

FIGURE 2.48 RISC-V instruction classes, examples, correspondence to high-level program language constructs, and percentage of RISC-V instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks. Figure 3.24 in Chapter 3 shows average percentage of the individual RISC-V instructions executed.

Further Readings on RISC-V ISA

Nested procedures: Sayfa 108 (Hennessy-Patterson RISC-V Edition 2nd Edition)

Global pointer: Sayfa 110

Frame pointer: Sayfa 110

RISC-V Addressing for Wide Immediates and Addresses: Sayfa 120

RISC-V Addressing Mode Summary: Sayfa 125

Real Stuff: The Rest of the RISC-V System and Special Instructions: Sayfa 486

Architecture: Chapter-6 Harris - Harris

Synch	Synch thread	I	FENCE
	Synch Instr & Data	I	FENCE.I
Environment	CALL	I	ECALL
	BREAK	I	EBREAK

Control Status Register (CSR)			
	Read/Write	I	CSRRW rd,csr,rs1
	Read & Set Bit	I	CSRRS rd,csr,rs1
	Read & Clear Bit	I	CSRRC rd,csr,rs1
	Read/Write Imm	I	CSRRWI rd,csr,imm
	Read & Set Bit Imm	I	CSRRSI rd,csr,imm
	Read & Clear Bit Imm	I	CSRRCI rd,csr,imm

RISC-V Pseudo Instructions

nop	addi x0, x0, 0	No operation
li rd, immediate	*Myriad sequences*	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if ≠ zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero
fmv.s rd, rs	fsgnj.s rd, rs, rs	Copy single-precision register
fabs.s rd, rs	fsgnjx.s rd, rs, rs	Single-precision absolute value
fneg.s rd, rs	fsgnjn.s rd, rs, rs	Single-precision negate
fmv.d rd, rs	fsgnj.d rd, rs, rs	Copy double-precision register
fabs.d rd, rs	fsgnjx.d rd, rs, rs	Double-precision absolute value
fneg.d rd, rs	fsgnjn.d rd, rs, rs	Double-precision negate
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if ≠ zero
blez rs, offset	bge x0, rs, offset	Branch if ≤ zero
bgez rs, offset	bge rs, x0, offset	Branch if ≥ zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if ≤
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if ≤, unsigned

RISC-V Pseudo Instructions

pseudoinstruction	Base Instruction	Meaning
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`ret`	`jalr x0, 0(x1)`	Return from subroutine
`call offset`	`auipc x1, offset`$[31:12] + $`offset`$[11]$ `jalr x1, offset[11:0](x1)`	Call far-away subroutine
`tail offset`	`auipc x6, offset`$[31:12] + $`offset`$[11]$ `jalr x0, offset[11:0](x6)`	Tail call far-away subroutine

Assembly vs High-Level Languages

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level. (Hennessy-Patterson)