Princeton University
COS 217: Introduction to Programming Systems
A Subset of SPARC Assembly Language

Abbreviations Used in Instruction Formats	
rs1	Source register 1.
rs2	Source register 2.
rd	Destination register.
exprX	Expression that the assembler (and linker) evaluates to a constX.
labelX	Label that the assembler (and linker) evaluates to a constX instruction displacement.

Abbreviations Used in Instruction Descriptions	
r[X]	The contents of register X. The instruction descriptions view r as an array of ints.
mem[X]	The contents of memory at location X. The instruction descriptions view mem as an array of chars.
constX	Constant that fits into X bits.
Z	Zero condition code. The instruction descriptions view Z as an int whose value is either 0 (FALSE) or 1 (TRUE).
N	Negative condition code. The instruction descriptions view N as an int whose value is either 0 (FALSE) or 1 (TRUE).
V	oVerflow condition code. The instruction descriptions view V as an int whose value is either 0 (FALSE) or 1 (TRUE).
C	Carry condition code. The instruction descriptions view C as an int whose value is either 0 (FALSE) or 1 (TRUE).

Load and Store Mnemonics (Format 3)	
	Load unsigned byte
ldub [rs1],rd	r[rd] = (unsigned int)mem[r[rs1]];
ldub [rs1+rs2],rd	r[rd] = (unsigned int)mem[r[rs1] + r[rs2]];
ldub [rs1+expr13],rd	r[rd] = (unsigned int)mem[r[rs1] + const13];
ldub [rs1-expr13],rd	r[rd] = (unsigned int)mem[r[rs1] - const13];
ldub [expr13+rs1],rd	r[rd] = (unsigned int)mem[r[rs1] + const13];
ldub [expr13],rd	r[rd] = (unsigned int)mem[const13];
	Load signed byte
ldsb [rs1],rd	r[rd] = (int)mem[r[rs1]];
ldsb [rs1+rs2],rd	r[rd] = (int)mem[r[rs1] + r[rs2]];
ldsb [rs1+expr13],rd	r[rd] = (int)mem[r[rs1] + const13];
ldsb [rs1-expr13],rd	r[rd] = (int)mem[r[rs1] - const13];
ldsb [expr13+rs1],rd	r[rd] = (int)mem[r[rs1] + const13];
ldsb [expr13],rd	r[rd] = (int)mem[const13];
	Load unsigned halfword
lduh [rs1],rd	r[rd] = *(unsigned short*)(mem + r[rs1]);
lduh [rs1+rs2],rd	r[rd] = *(unsigned short*)(mem + r[rs1] + r[rs2]);
lduh [rs1+expr13],rd	r[rd] = *(unsigned short*)(mem + r[rs1] + const13);
lduh [rs1-expr13],rd	r[rd] = *(unsigned short*)(mem + r[rs1] - const13);
lduh [expr13+rs1],rd	r[rd] = *(unsigned short*)(mem + r[rs1] + const13);
lduh [expr13],rd	r[rd] = *(unsigned short*)(mem + const13);
	Load signed halfword
ldsh [rs1],rd	r[rd] = *(short*)(mem + r[rs1]);
ldsh [rs1+rs2],rd	r[rd] = *(short*)(mem + r[rs1] + r[rs2]);
ldsh [rs1+expr13],rd	r[rd] = *(short*)(mem + r[rs1] + const13);
ldsh [rs1-expr13],rd	r[rd] = *(short*)(mem + r[rs1] - const13);
ldsh [expr13+rs1],rd	r[rd] = *(short*)(mem + r[rs1] + const13);
ldsh [expr13],rd	r[rd] = *(short*)(mem + const13);
	Load word
ld [rs1],rd	r[rd] = *(int*)(mem + r[rs1]);
ld [rs1+rs2],rd	r[rd] = *(int*)(mem + r[rs1 + r[rs2]]);
ld [rs1+expr13],rd	r[rd] = *(int*)(mem + r[rs1] + const13);
ld [rs1-expr13],rd	r[rd] = *(int*)(mem + r[rs1] - const13);
ld [expr13+rs1],rd	r[rd] = *(int*)(mem + r[rs1] + const13);
ld [expr13],rd	r[rd] = *(int*)(mem + const13);

	Load doubleword
ldd [rs1],rd	r[rd] = *(int*)(mem + r[rs1]); r[rd+1] = *(int*)(mem + r[rs1] + 4);
ldd [rs1+rs2],rd	r[rd] = *(int*)(mem + r[rs1] + r[rs2]); r[rd+1] = *(int*)(mem + r[rs1] + r[rs2] + 4);
ldd [rs1+expr13],rd	r[rd] = *(int*)(mem + r[rs1] + const13); r[rd+1] = *(int*)(mem + r[rs1] + const13 + 4);
ldd [rs1-expr13],rd	r[rd] = *(int*)(mem + r[rs1] - const13); r[rd+1] = *(int*)(mem + r[rs1] - const13 + 4);
ldd [expr13+rs1],rd	r[rd] = *(int*)(mem + r[rs1] + const13); r[rd+1] = *(int*)(mem + r[rs1] + const13 + 4);
ldd [expr13],rd	r[rd] = *(int*)(mem + const13); r[rd+1] = *(int*)(mem + const13 + 4);
	Swap
swap [rs1],rd	temp=mem[r[rs1]]; mem[r[rs1]]=r[rd]; r[rd]=temp;
swap [rs1+rs2],rd	temp=mem[r[rs1]+r[rs2]]; mem[r[rs1]+r[rs2]]=r[rd]; r[rd]=temp;
swap [rs1+expr13],rd	temp=mem[r[rs1]+const13]; mem[r[rs1]+const13]=r[rd]; r[rd]=temp;
swap [rs1-expr13],rd	temp=mem[r[rs1]-const13]; mem[r[rs1]-const13]=r[rd]; r[rd]=temp;
swap [expr13+rs1],rd	temp=mem[r[rs1]+const13]; mem[r[rs1]+const13]=r[rd]; r[rd]=temp;
swap [expr13],rd	temp=mem[expr13]; mem[expr13]=r[rd]; r[rd]=temp;
	Store byte
stb rd,[rs1]	mem[r[rs1]] = r[rd];
stb rd,[rs1+rs2]	mem[r[rs1] + r[rs2]] = r[rd];
stb rd,[rs1+expr13]	mem[r[rs1] + const13] = r[rd];
stb rd,[rs1-expr13]	mem[r[rs1] - const13] = r[rd];
stb rd,[expr13+rs1]	mem[r[rs1] + const13] = r[rd];
stb rd,[expr13]	mem[const13] = r[rd];
	Store halfword
sth rd,[rs1]	*(short*)(mem + r[rs1]) = r[rd];
sth rd,[rs1+rs2]	*(short*)(mem + r[rs1] + r[rs2]) = r[rd];
sth rd,[rs1+expr13]	*(short*)(mem + r[rs1] + const13) = r[rd];
sth rd,[rs1-expr13]	*(short*)(mem + r[rs1] - const13) = r[rd];
sth rd,[expr13+rs1]	*(short*)(mem + r[rs1] + const13) = r[rd];
sth rd,[expr13]	*(short*)(mem + const13) = r[rd];
	Store word
st rd,[rs1]	*(int*)(mem + r[rs1]) = r[rd];
st rd,[rs1+rs2]	*(int*)(mem + r[rs1] + r[rs2]) = r[rd];
st rd,[rs1+expr13]	*(int*)(mem + r[rs1] + const13) = r[rd];
st rd,[rs1-expr13]	*(int*)(mem + r[rs1] - const13) = r[rd];
st rd,[expr13+rs1]	*(int*)(mem + r[rs1] + const13) = r[rd];
st rd,[expr13]	*(int*)(mem + const13) = r[rd];
	Store doubleword
std rd,[rs1]	*(int*)(mem + r[rs1]) = r[rd]; *(int*)(mem + r[rs1] + 4) = r[rd+1];
std rd,[rs1+rs2]	*(int*)(mem + r[rs1] + r[rs2]) = r[rd]; *(int*)(mem + r[rs1] + r[rs2] + 4) = r[rd+1];
std rd,[rs1+expr13]	*(int*)(mem + r[rs1] + const13) = r[rd]; *(int*)(mem + r[rs1] + const13 + 4) = r[rd+1];
std rd,[rs1-expr13]	*(int*)(mem + r[rs1] - const13) = r[rd]; *(int*)(mem + r[rs1] - const13 + 4) = r[rd+1];
std rd,[expr13+rs1]	*(int*)(mem + r[rs1] + const13) = r[rd]; *(int*)(mem + r[rs1] + const13 + 4) = r[rd+1];
std rd,[expr13]	*(int*)(mem + const13) = r[rd]; *(int*)(mem + const13 + 4) = r[rd+1];

Shift Mnemonics (Format 3)

	Shift left logical
sll rs1,rs2,rd	r[rd] = r[rs1] << r[rs2];
sll rs1,expr13,rd	r[rd] = r[rs1] << const13;
	Shift right logical
srl rs1,rs2,rd	r[rd] = (unsigned int)r[rs1] >> r[rs2];
srl rs1,expr13,rd	r[rd] = (unsigned int)r[rs1] >> const13;
	Shift right arithmetic
sra rs1,rs2,rd	r[rd] = r[rs1] >> r[rs2];
sra rs1,expr13,rd	r[rd] = r[rs1] >> const13;

Arithmetic Mnemonics (Format 3)

	Add
add rs1,rs2,rd	r[rd] = r[rs1] + r[rs2];
add rs1,expr13,rd	r[rd] = r[rs1] + const13;
	Add, and set condition codes
addcc rs1,rs2,rd	r[rd] = r[rs1] + r[rs2]; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & r[rs2]<0 & r[rd]>0) | (r[rs1]>0 & r[rs2]>0 & r[rd]<0); C = (r[rs1]<0 & r[rs2]<0) | (r[rd]>0 & (r[rs1]<0 | r[rs2]<0));
addcc rs1,expr13,rd	r[rd] = r[rs1] + const13; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & const13<0 & r[rd]>0) | (r[rs1]>0 & const13>0 & r[rd]<0); C = (r[rs1]<0 & const13<0) | (r[rd]>0 & (r[rs1]<0 | const13<0));
	Add extended
addx rs1,rs2,rd	r[rd] = r[rs1] + r[rs2] + C;
addx rs1,expr13,rd	r[rd] = r[rs1] + const13 + C;
	Add extended, and set condition codes
addxcc rs1,rs2,rd	r[rd] = r[rs1] + r[rs2] + C; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & r[rs2]<0 & r[rd]>0) | (r[rs1]>0 & r[rs2]>0 & r[rd]<0); C = (r[rs1]<0 & r[rs2]<0) | (r[rd]>0 & (r[rs1]<0 | r[rs2]<0));
addxcc rs1,expr13,rd	r[rd] = r[rs1] + const13 + C; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & const13<0 & r[rd]>0) | (r[rs1]>0 & const13>0 & r[rd]<0); C = (r[rs1]<0 & const13<0) | (r[rd]>0 & (r[rs1]<0 | const13<0));
	Subtract
sub rs1,rs2,rd	r[rd] = r[rs1] – r[rs2];
sub rs1,expr13,rd	r[rd] = r[rs1] – const13;
	Subtract, and set condition codes
subcc rs1,rs2,rd	r[rd] = r[rs1] – r[rs2]; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & r[rs2]>0 & r[rd]>0) | (r[rs1]>0 & r[rs2]<0 & r[rd]<0); C = (r[rs1]>0 & r[rs2]<0) | (r[rd]<0 & (r[rs1]>0 | r[rs2]<0));
subcc rs,expr13,rd	r[rd] = r[rs1] – const13; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & const13>0 & r[rd]>0) | (r[rs1]>0 & const13<0 & r[rd]<0); C = (r[rs1]>0 & const13<0) | (r[rd]<0 & (r[rs1]>0 | const13<0));
	Subtract extended
subx rs1,rs2,rd	r[rd] = r[rs1] – r[rs2] - C;
subx rs1,expr13,rd	r[rd] = r[rs1] – const13 - C;
	Subtract extended, and set condition codes
subxcc rs1,rs2,rd	r[rd] = r[rs1] – r[rs2] - C; N = r[rd]<0; Z = r[rd] == 0; V = (r[rs1]<0 & r[rs2]>0 & r[rd]>0) | (r[rs1]>0 & r[rs2]<0 & r[rd]<0); C = (r[rs1]>0 & r[rs2]<0) | (r[rd]<0 & (r[rs1]>0 | r[rs2]<0));
subxcc rs,expr13,rd	r[rd] = r[rs1] – const13 - C; N = r[rd]<0; Z = r[rd]==0; V = (r[rs1]<0 & const13>0 & r[rd]>0) | (r[rs1]>0 & const13<0 & r[rd]<0); C = (r[rs1]>0 & const13<0) | (r[rd]<0 & (r[rs1]>0 | const13<0));
	Negate
neg rs2,rd	Synthetic instruction for: sub %g0, rs2, rd
neg rd	Synthetic instruction for: sub %g0, rd, rd
	Increment
inc rd	Synthetic instruction for: add rd, 1, rd
inc expr13,rd	Synthetic instruction for: add rd, expr13, rd
	Increment, and set condition codes
inccc rd	Synthetic instruction for: addcc rd, 1, rd
inccc expr13,rd	Synthetic instruction for: addcc rd, is, rd
	Decrement
dec rd	Synthetic instruction for: sub rd, 1, rd
dec expr13,rd	Synthetic instruction for: sub rd, expr13, rd

	Decrement, and set condition codes
deccc rd	Synthetic instruction for: subcc rd, 1, rd
deccc expr13,rd	Synthetic instruction for: subcc rd, expr13, rd
	Compare
cmp rs,rs2	Synthetic instruction for: subcc rs, rs2, %g0
cmp rs,expr13	Synthetic instruction for: subcc rs, expr13, %g0

Logical Mnemonics (Format 3)

	And
and rs1,rs2,rd	r[rd] = r[rs1] & r[rs2]
and rs1,expr13,rd	r[rd] = r[rs1] & const13
	And, and set condition codes
andcc rs1,rs2,rd	r[rd] = r[rs1] & r[rs2]; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
andcc rs1,expr13,rd	r[rd] = r[rs1] & const13; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
	And negative
andn rs1,rs2,rd	r[rd] = r[rs1] & ~r[rs2];
andn rs1,expr13,rd	r[rd] = r[rs1] & ~expr13;
	And negative, and set condition codes
andncc rs1,rs2,rd	r[rd] = r[rs1] & ~r[rs2]; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
andncc rs1,expr13,rd	r[rd] = r[rs1] & ~const13; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
	Or
or rs1,rs2,rd	r[rd] = r[rs1] | r[rs2]
or rs1,expr13,rd	r[rd] = r[rs1] | expr13
	Or, and set condition codes
orcc rs1,rs2,rd	r[rd] = r[rs1] | r[rs2]; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
orcc rs1,expr13,rd	r[rd] = r[rs1] | const13; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
	Or negative
orn rs1,rs2,rd	r[rd] = r[rs1] | ~r[rs2];
orn rs1,expr13,rd	r[rd] = r[rs1] | ~expr13;
	Or negative, and set condition codes
orncc rs1,rs2,rd	r[rd] = r[rs1] | ~r[rs2]; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
orncc rs1,expr13,rd	r[rd] = r[rs1] | ~const13; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
	Exclusive or
xor rs1,rs2,rd	r[rd] = r[rs1] ^ r[rs2];
xor rs1,expr13,rd	r[rd] = r[rs1] ^ const13;
	Exclusive or, and set condition codes
xorcc rs1,rs2,rd	r[rd] = r[rs1] ^ r[rs2]; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
xorcc rs1,expr13,rd	r[rd] = r[rs1] ^ const13; N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
	Exclusive nor
xnor rs1,rs2,rd	r[rd] = ~(r[rs1] ^ r[rs2]);
xnor rs1,expr13,rd	r[rd] = ~(r[rs1] ^ expr13);
	Exclusive nor, and set condition codes
xnorcc rs1,rs2,rd	r[rd] = ~(r[rs1] ^ r[rs2]); N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
xnorcc rs1,expr13,rd	r[rd] = ~(r[rs1] ^ const13); N = r[rd]<0; Z = r[rd]==0; V=0; C=0;
	Clear
clr rd	Synthetic instruction for: or %g0, %g0, rd
	Move
mov rs2,rd	Synthetic instruction for: or %g0, rs2, rd
mov expr13,rd	Synthetic instruction for: or %g0, expr13, rd
	Test
tst rs2	Synthetic instruction for: orcc %g0, rs2, %g0
	Bit test
btst rs2,rs1	Synthetic instruction for: andcc rs1, rs2, %g0
btst expr13,rs1	Synthetic instruction for: andcc rs1, expr13, %g0
	Bit set
bset rs2,rd	Synthetic instruction for: or rd, rs2, rd
bset expr13,rd	Synthetic instruction for: or rd, expr13, rd
	Bit clear
bclr rs2,rd	Synthetic instruction for: andn rd, rs2, rd
bclr expr13,rd	Synthetic instruction for: andn rd, expr13, rd
	Bit toggle
btog rs2,rd	Synthetic instruction for: xor rd, rs2, rd
btog expr13,rd	Synthetic instruction for: xor rd, expr13, rd
	Not
not rs1,rd	Synthetic instruction for: xnor rs1, %g0, rd
not rd	Synthetic instruction for: xnor rd, %g0, rd

Integer Branch Mnemonics (Format 2)	
Unconditional branching:	

ba{,a} label22	**Branch to label always**
	pc = npc;
	npc = const22 << 2;
	if (a == 1) { pc = npc; npc += 4; }

Signed number branching:	

be{,a} label22	**Branch if equal**
	pc = npc;
	if (Z) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bne{,a} label22	**Branch if not equal**
	pc = npc;
	if (! Z) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bl{,a} label22	**Branch if less than**
	pc = npc;
	if (N ^ V) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

ble{,a} label22	**Branch if less than or equal to**
	pc = npc;
	if (Z | (N ^ V)) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bge{,a} label22	**Branch if greater than or equal to**
	pc = npc;
	if (!(N ^ V)) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bg{,a} label22	**Branch if greater than**
	pc = npc;
	if (!(Z | (N ^ V))) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

Unsigned number branching:	
blu{,a} label22	**Branch if less than (unsigned)** Synonym for: bcs{,a} label22

bleu{,a} label22	**Branch if less than or equal to (unsigned)**
	pc = npc;
	if (C | Z) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bgeu{,a} label22	**Branch if greater than or equal to (unsigned)** Synonym for: bcc{,a} label22

bgu{,a} label22	**Branch if greater than (unsigned)**
	pc = npc;
	if (!(C | Z)) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

Individual condition code branching:	

bpos{,a} label22	**Branch if positive (or zero)**
	pc = npc;
	if (! N) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bneg{,a} label22	**Branch if negative**
	pc = npc;
	if (N) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bcs{,a} label22	**Branch if C is set**
	pc = npc;
	if (C) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bcc{,a} label22	**Branch if C is clear**
	pc = npc;
	if (! C) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bvs{,a} label22	**Branch if V is set**
	pc = npc;
	if (V) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bvc{,a} label22	**Branch if V is clear**
	pc = npc;
	if (! V) npc = const22 << 2;
	else if (a == 0) npc += 4;
	else { pc += 4; npc += 8; }

bz{,a} label22	**Branch if zero** Synonym for: be{,a} label22
bnz{,a} label22	**Branch if not zero** Synonym for: bne{,a} label22

Control Mnemonics (Format 3)	

jmpl rs1,rd
jmpl rs1+rs2,rd
jmpl rs1+expr13,rd
jmpl rs1-expr13,rd
jmpl expr13+rs1,rd
jmpl expr13,rd
	Jump and link
	r[rd] = pc; pc = npc; npc = r[rs1];
	r[rd] = pc; pc = npc; npc = r[rs1] + r[rs2];
	r[rd] = pc; pc = npc; npc = r[rs1] + const13;
	r[rd] = pc; pc = npc; npc = r[rs1] - const13;
	r[rd] = pc; pc = npc; npc = r[rs1] + const13;
	r[rd] = pc; pc = npc; npc = const13;

jmp rs1 jmp rs1+rs2 jmp rs1+expr13 jmp rs1-expr13 jmp expr13+rs1 jmp expr13	**Jump** Synthetic instruction for jmpl rs1, %g0 Synthetic instruction for jmpl rs1+rs2, %g0 Synthetic instruction for jmpl rs1+expr13, %g0 Synthetic instruction for jmpl rs1-expr13, %g0 Synthetic instruction for jmpl expr13+rs1, %g0 Synthetic instruction for jmpl expr13, %g0
call rs1	**Call indirect** Synthetic instruction for: jmpl rs1, %o7
ret	**Return from subroutine** Synthetic instruction for: jmpl %i7 + 8, %g0
retl	**Return from leaf subroutine** Synthetic instruction for: jmpl %o7 + 8, %g0

save rs1,rs2,rd
 	
save rs1,expr13,rd
	Save register window and add
	temp = r[rs1] + r[rs2];
	save the register window
	r[rd] = temp;
	temp = r[rs1] + expr13;
	(save the register window)
	r[rd] = temp;

restore rs1,rs2,rd
 	
restore rs1,expr13,rd
	Restore register window and add
	temp = r[rs1] + r[rs2];
	(restore the register window)
	r[rd] = temp;
	temp = r[rs1] + r[rs2];
	(restore the register window)
	r[rd] = temp;

restore	**Restore register window and add** Synthetic instruction for: restore %g0, %g0, %g0

Control Mnemonics (Format 2)	
nop	**No operation**
sethi expr22,rd	**Set high-order bits** r[rd] = const22 << 10;
set expr32,rd	**Set** Synthetic instruction for: sethi %hi(const32), rd or rd, %lo(const32), rd where: %hi(const32) is equivalent to (const32 >> 10) %lo(const32) is equivalent to (const32 & 0x3ff)

Control Mnemonics (Format 1)	
	Call
call label30	r[o7] = pc; pc = npc; npc = pc + const30

Assembler Directives	
label:	Record the fact that label marks the current location within the current section
.section ".sectionname"	Make the sectionname section the current section
.skip n	Skip n bytes of memory in the current section
.align n	Increase the current section's location counter so it is evenly divisible by n
.byte bytevalue1, bytevalue2, ...	Allocate memory containing bytevalue1, bytevalue2, ... in the current section
.half halfvalue1, halfvalue2, ...	Allocate memory containing halfvalue1, halfvalue2, ... in the current section
.word wordvalue1, wordvalue2, ...	Allocate memory containing wordvalue1, wordvalue2, ... in the current section
.ascii "string1", "string2", ...	Allocate memory containing the characters from string1, string2, ... in the current section
.asciz "string1", "string2", ...	Allocate memory containing string1, string2, ..., where each string is NULL terminated, in the current section
.global label1, label2, ...	Mark label1, label2, ... so they are available to the linker