Computer Structure and Language

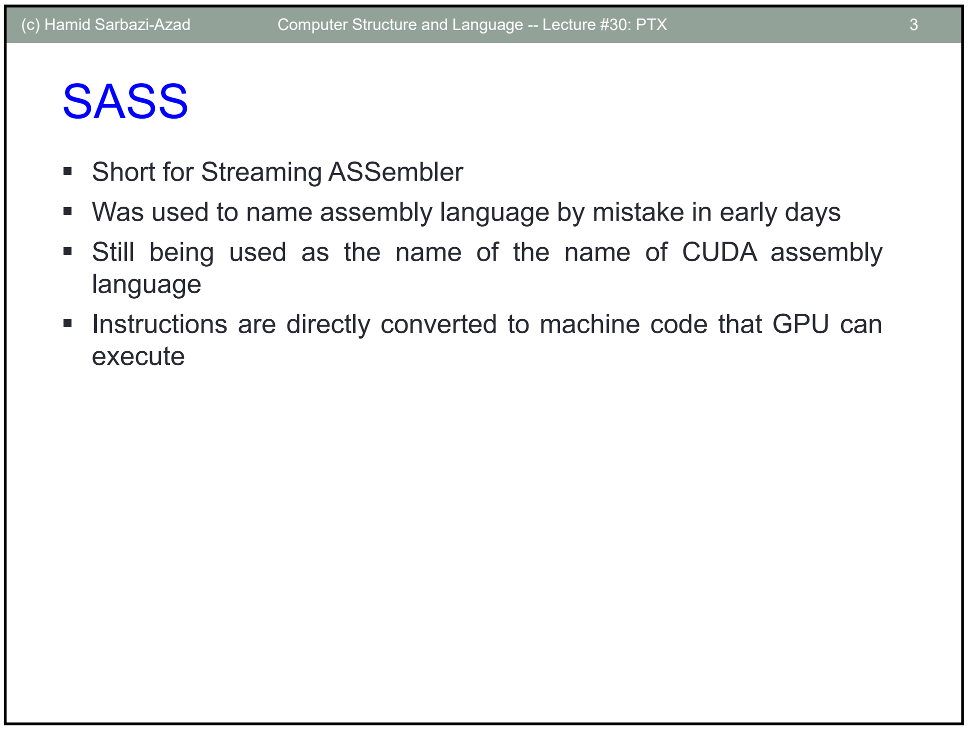
Hamid Sarbazi-Azad
Department of Computer Engineering
Sharif University of Technology (SUT)
Tehran, Iran

1

GPU & CUDA

- GPU
  - Specialized hardware with special features
  - Designed for massively parallel operations
  - Ideal for many scientific applications in addition to graphics
- CUDA
  - A parallel computing platform and programming model
  - Developed by NVIDIA

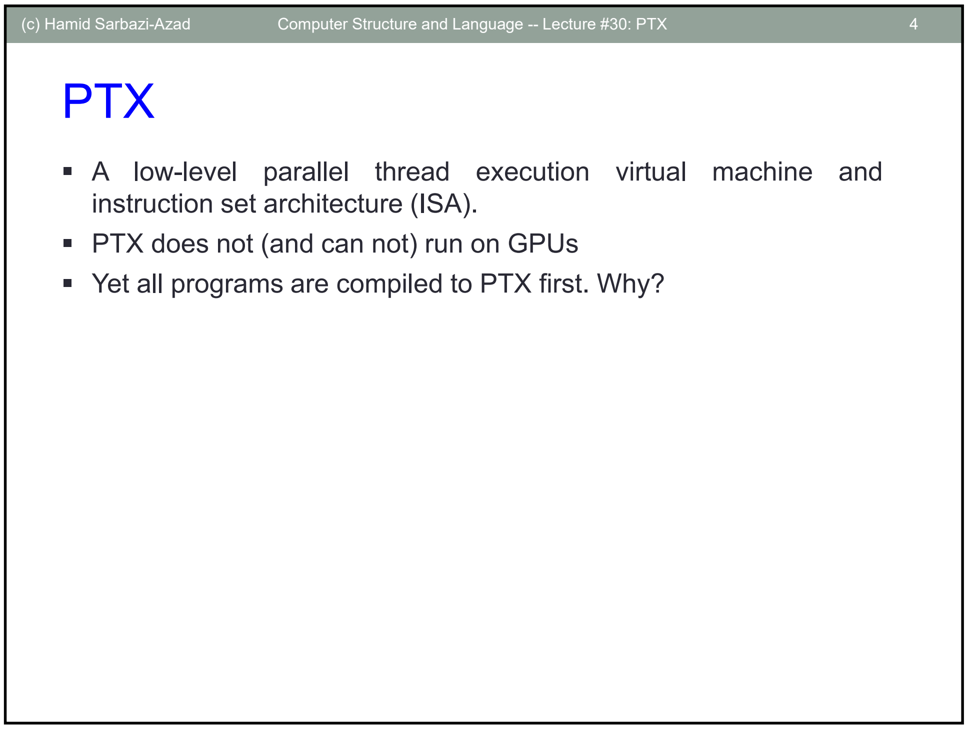
2



SASS

- Short for Streaming ASSembler
- Was used to name assembly language by mistake in early days
- Still being used as the name of the name of CUDA assembly language
- Instructions are directly converted to machine code that GPU can execute

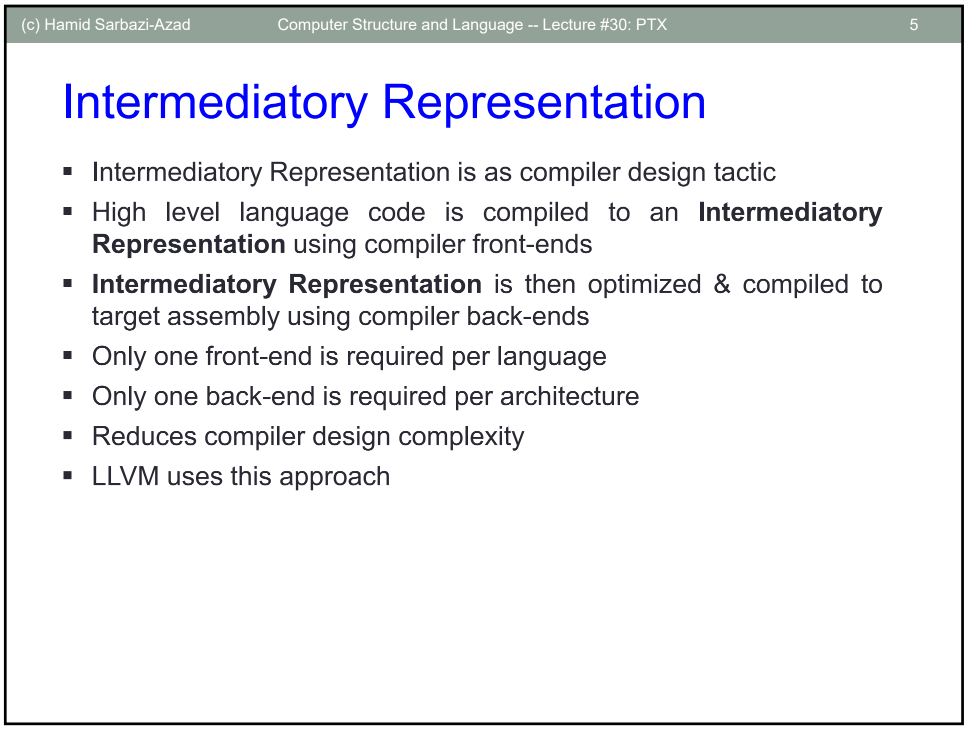
3



PTX

- A low-level parallel thread execution virtual machine and instruction set architecture (ISA).
- PTX does not (and can not) run on GPUs
- Yet all programs are compiled to PTX first. Why?

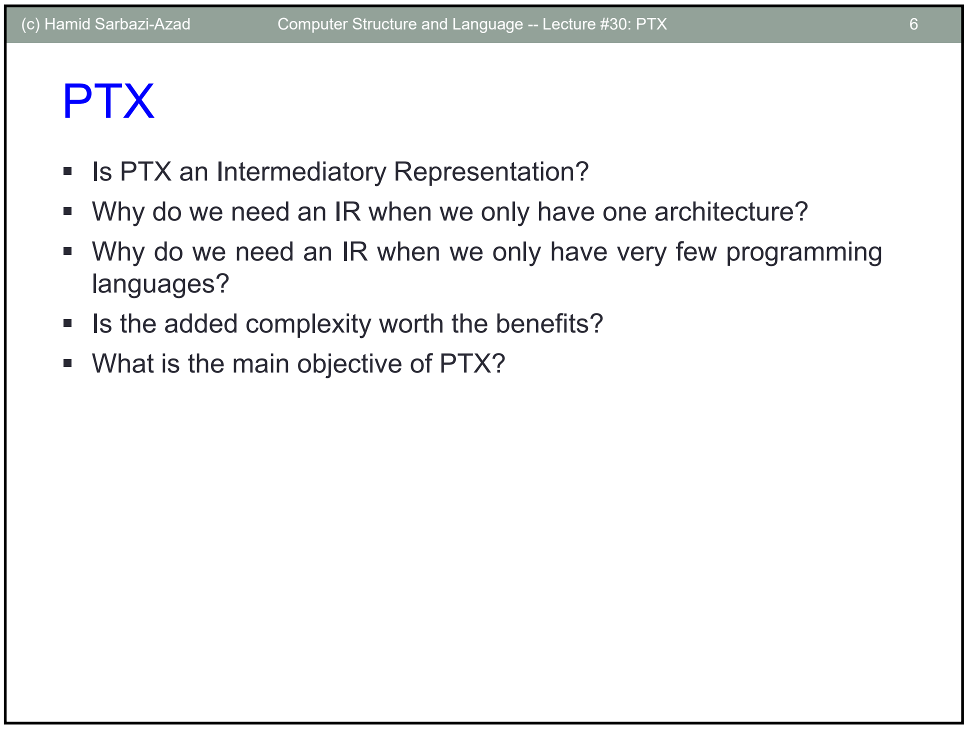
4



Intermediatory Representation

- Intermediatory Representation is as compiler design tactic
- High level language code is compiled to an **Intermediatory Representation** using compiler front-ends
- **Intermediatory Representation** is then optimized & compiled to target assembly using compiler back-ends
- Only one front-end is required per language
- Only one back-end is required per architecture
- Reduces compiler design complexity
- LLVM uses this approach

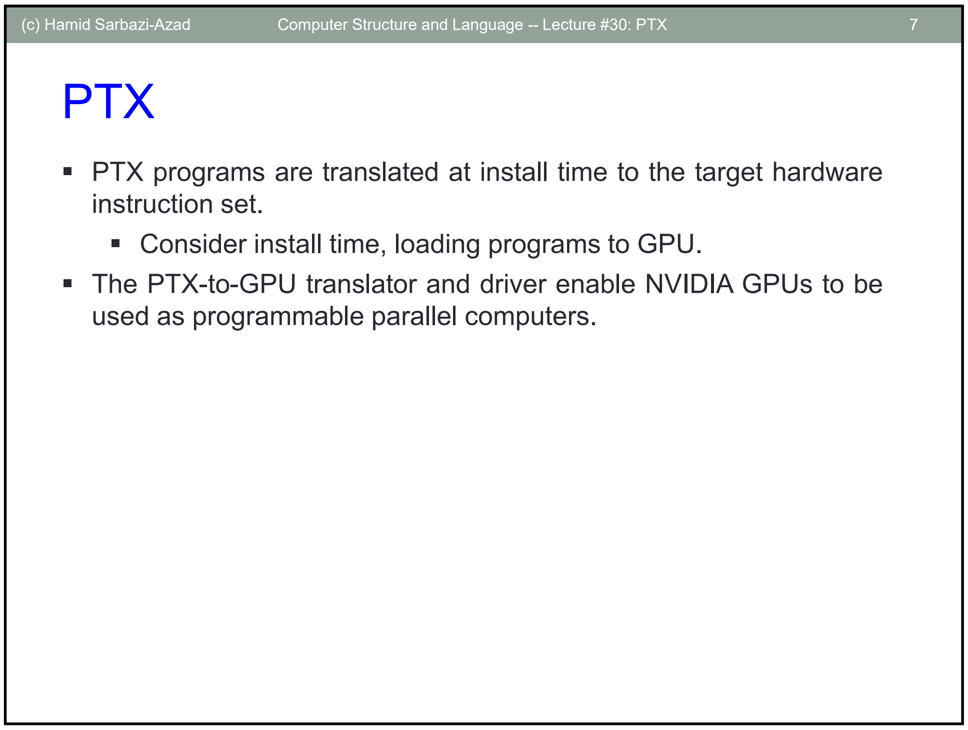
5



PTX

- Is PTX an Intermediatory Representation?
- Why do we need an IR when we only have one architecture?
- Why do we need an IR when we only have very few programming languages?
- Is the added complexity worth the benefits?
- What is the main objective of PTX?

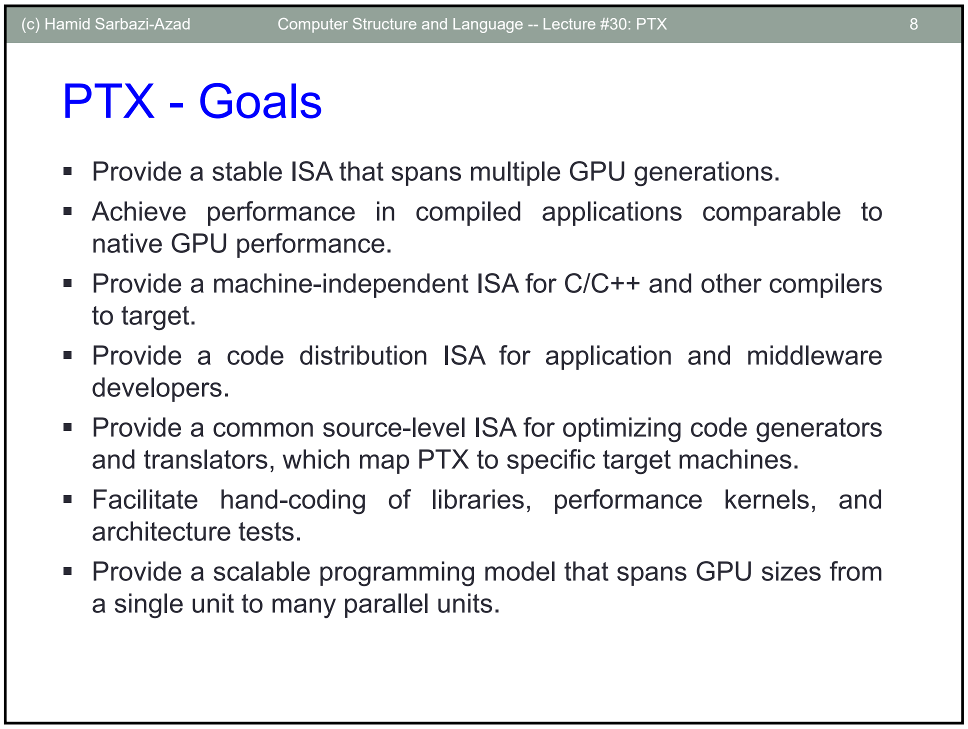
6



PTX

- PTX programs are translated at install time to the target hardware instruction set.
  - Consider install time, loading programs to GPU.
- The PTX-to-GPU translator and driver enable NVIDIA GPUs to be used as programmable parallel computers.

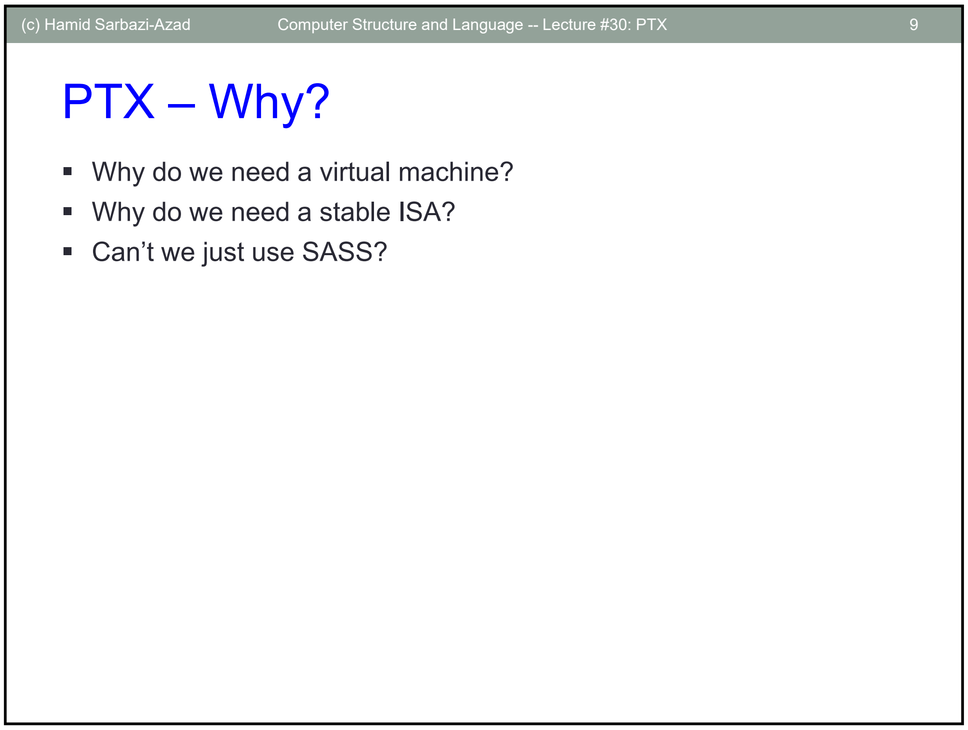
7



PTX - Goals

- Provide a stable ISA that spans multiple GPU generations.
- Achieve performance in compiled applications comparable to native GPU performance.
- Provide a machine-independent ISA for C/C++ and other compilers to target.
- Provide a code distribution ISA for application and middleware developers.
- Provide a common source-level ISA for optimizing code generators and translators, which map PTX to specific target machines.
- Facilitate hand-coding of libraries, performance kernels, and architecture tests.
- Provide a scalable programming model that spans GPU sizes from a single unit to many parallel units.

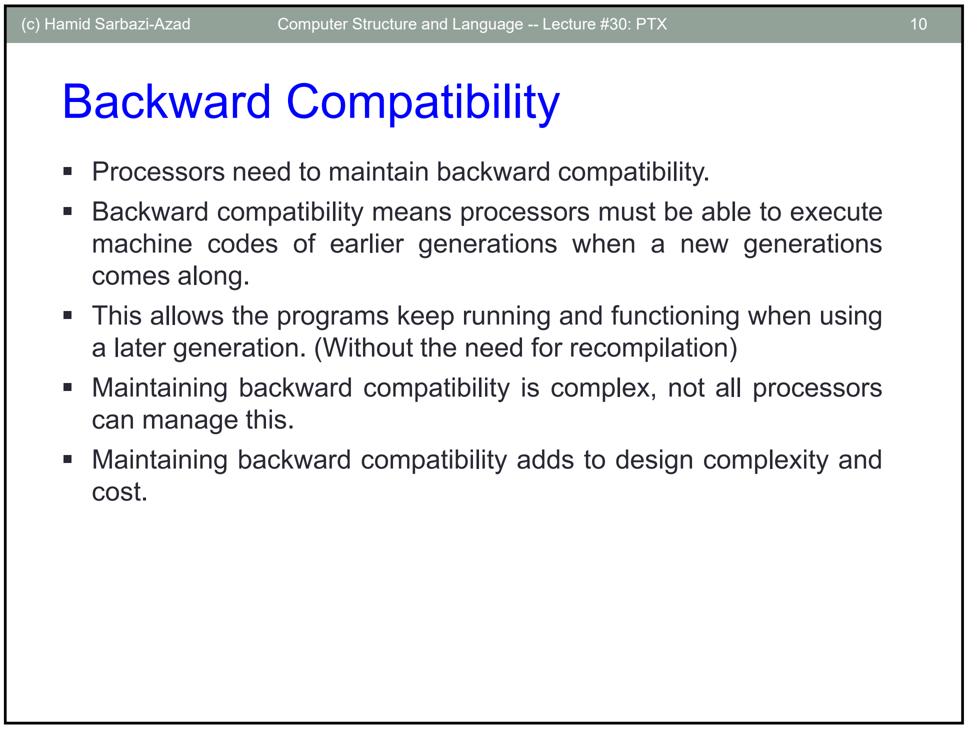
8



PTX – Why?

- Why do we need a virtual machine?
- Why do we need a stable ISA?
- Can’t we just use SASS?

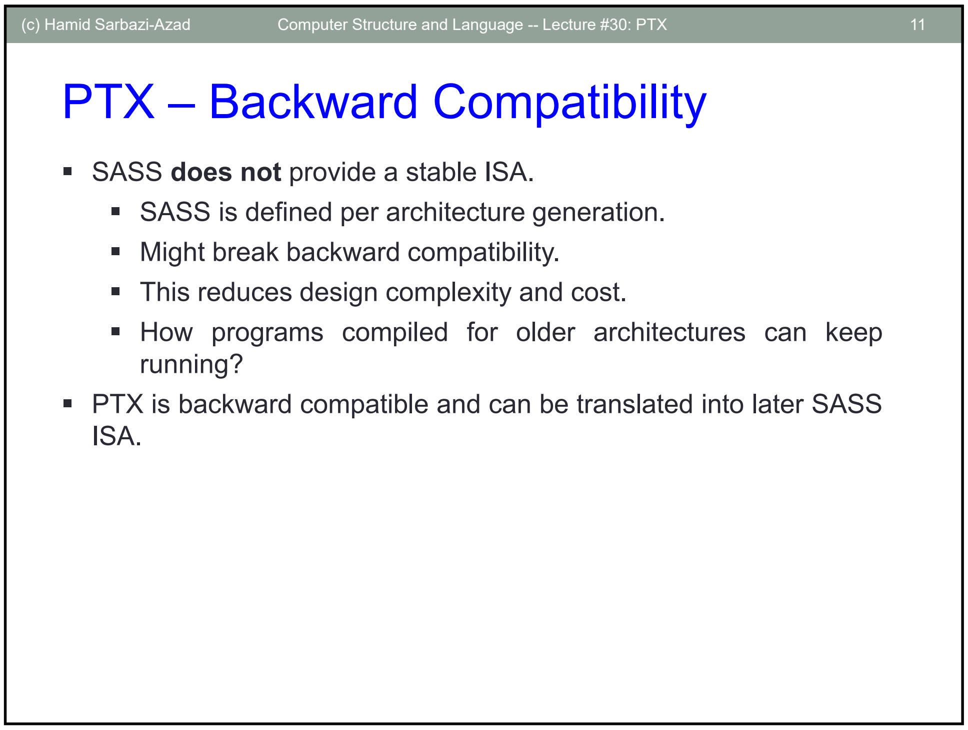
9



Backward Compatibility

- Processors need to maintain backward compatibility.
- Backward compatibility means processors must be able to execute machine codes of earlier generations when a new generations comes along.
- This allows the programs keep running and functioning when using a later generation. (Without the need for recompilation)
- Maintaining backward compatibility is complex, not all processors can manage this.
- Maintaining backward compatibility adds to design complexity and cost.

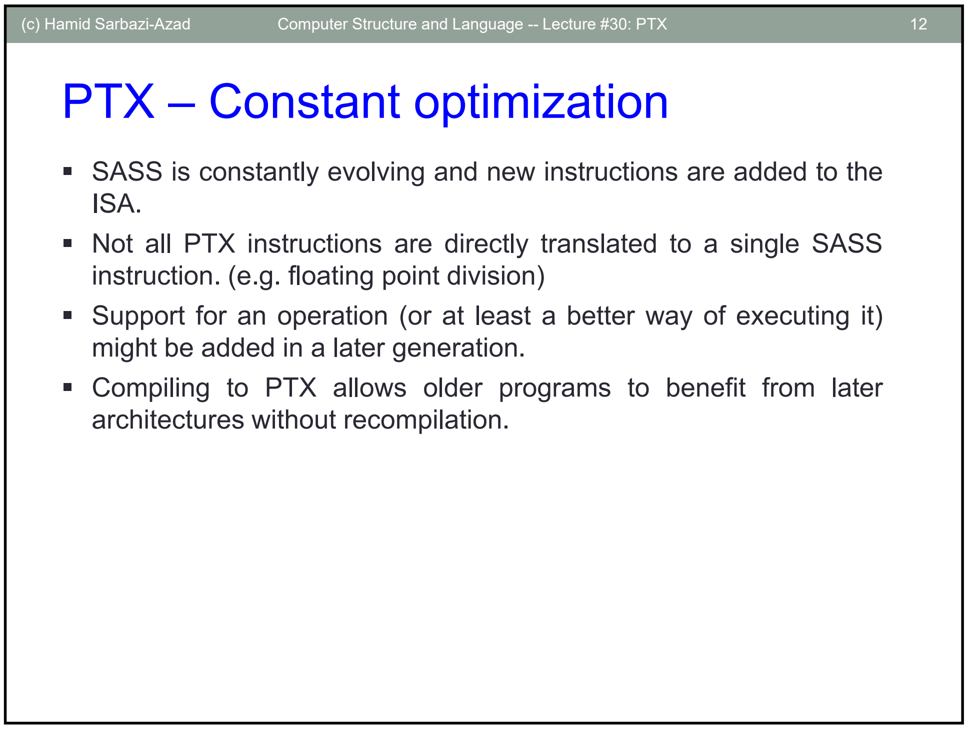
10



PTX – Backward Compatibility

- SASS **does not** provide a stable ISA.
  - SASS is defined per architecture generation.
  - Might break backward compatibility.
  - This reduces design complexity and cost.
  - How programs compiled for older architectures can keep running?
- PTX is backward compatible and can be translated into later SASS ISA.

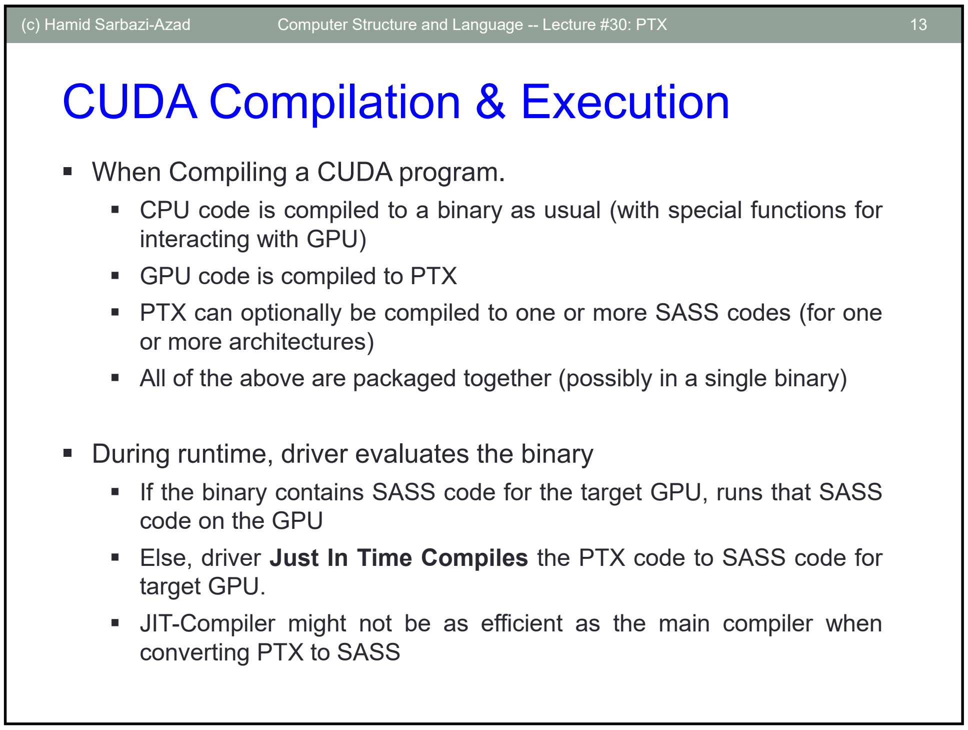
11



PTX – Constant optimization

- SASS is constantly evolving and new instructions are added to the ISA.
- Not all PTX instructions are directly translated to a single SASS instruction. (e.g. floating point division)
- Support for an operation (or at least a better way of executing it) might be added in a later generation.
- Compiling to PTX allows older programs to benefit from later architectures without recompilation.

12



CUDA Compilation & Execution

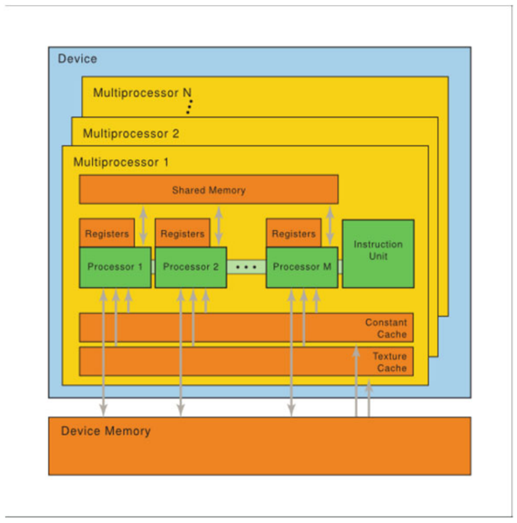
- When Compiling a CUDA program.
  - CPU code is compiled to a binary as usual (with special functions for interacting with GPU)
  - GPU code is compiled to PTX
  - PTX can optionally be compiled to one or more SASS codes (for one or more architectures)
  - All of the above are packaged together (possibly in a single binary)
- During runtime, driver evaluates the binary
  - If the binary contains SASS code for the target GPU, runs that SASS code on the GPU
  - Else, driver **Just In Time Compiles** the PTX code to SASS code for target GPU.
  - JIT-Compiler might not be as efficient as the main compiler when converting PTX to SASS

13

PTX PROGRAMMING

14

PTX Machine Model



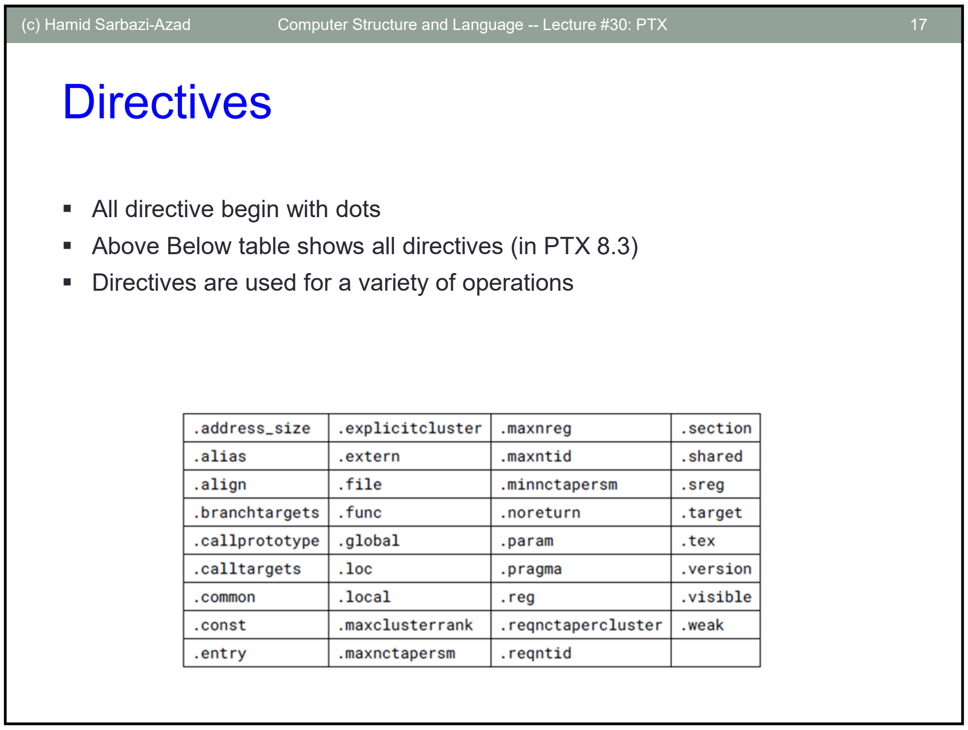
15

Syntax Example

```
        .reg     .b32 r1, r2;
        .global  .f32  array[N];

start:  mov.b32   r1, %tid.x;
        shl.b32   r1, r1, 2;          // shift thread id by 2 bits
        ld.global.b32 r2, array[r1];  // thread[tid] gets array[tid]
        add.f32   r2, r2, 0.5;        // add 1/2
```

- Very similar to other assemblies
- Supports some high level features (like array addressing)
- Has some special registers (like tid)
- Registers are defined like variables (with type specifiers)
- Instructions have a type specifier
- State space (memory region) is specified (in declaration, load & store)

16

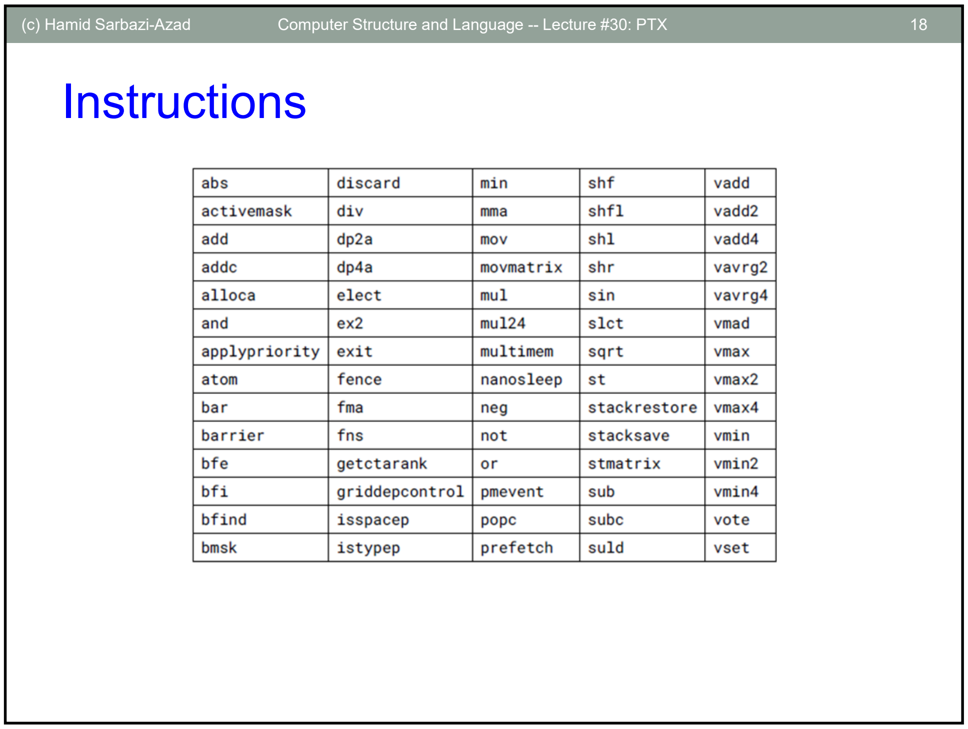


Directives

- All directive begin with dots
- Above Below table shows all directives (in PTX 8.3)
- Directives are used for a variety of operations

.address_size	.explicitcluster	.maxnreg	.section
.alias	.extern	.maxntid	.shared
.align	.file	.minnctapersm	.sreg
.branchtargets	.func	.noreturn	.target
.callprototype	.global	.param	.tex
.calltargets	.loc	.pragma	.version
.common	.local	.reg	.visible
.const	.maxclusterrank	.reqnctapercluster	.weak
.entry	.maxnctapersm	.reqntid	

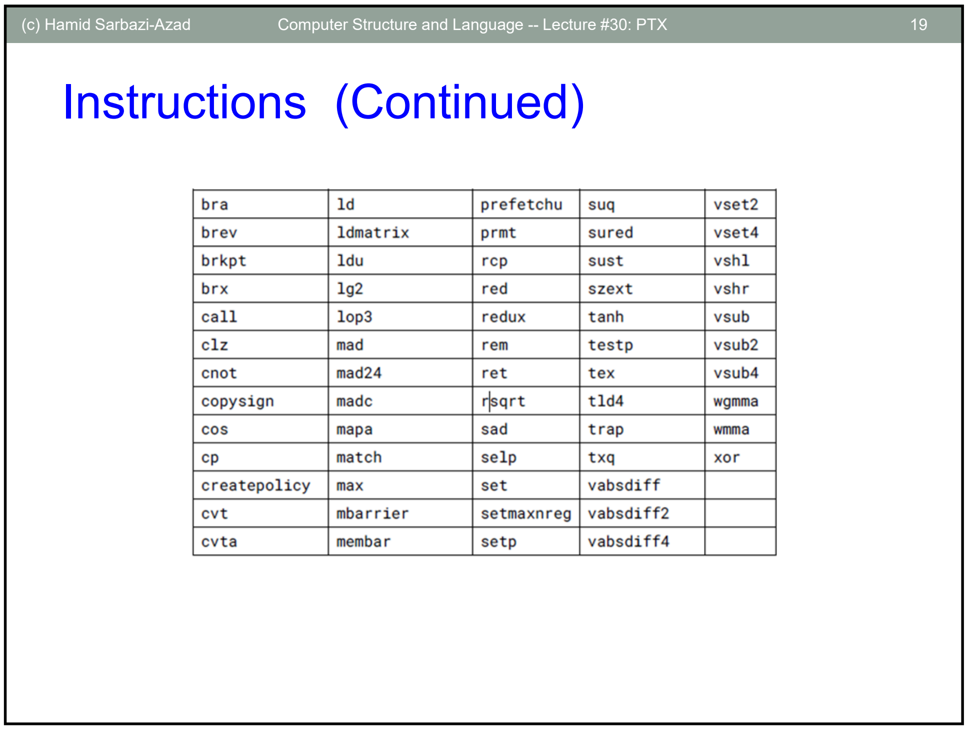
17



Instructions

abs	discard	min	shf	vadd
activemask	div	mma	shfl	vadd2
add	dp2a	mov	shl	vadd4
addc	dp4a	movmatrix	shr	vavrg2
alloca	elect	mul	sin	vavrg4
and	ex2	mul24	slct	vmad
applypriority	exit	multimem	sqrt	vmax
atom	fence	nanosleep	st	vmax2
bar	fma	neg	stackrestore	vmax4
barrier	fns	not	stacksave	vmin
bfe	getctarank	or	stmatrix	vmin2
bfi	griddepcontrol	pmevent	sub	vmin4
bfind	isspacep	popc	subc	vote
bmsk	istypep	prefetch	suld	vset

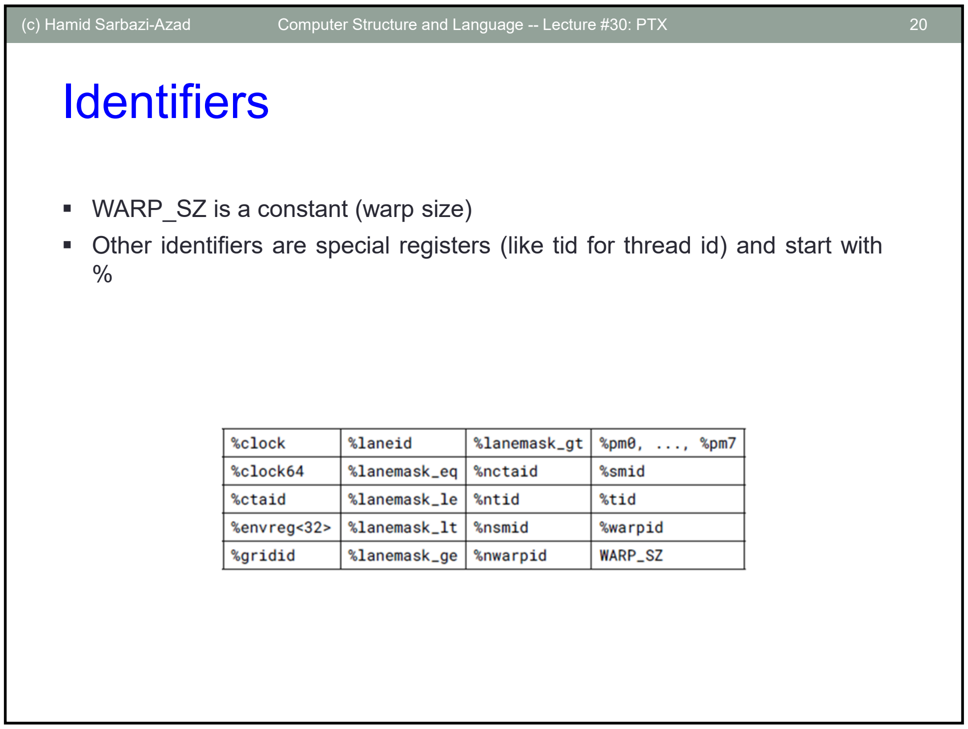
18



Instructions (Continued)

bra	ld	prefetchu	suq	vset2
brev	ldmatrix	prmt	sured	vset4
brkpt	ldu	rcp	sust	vshl
brx	lg2	red	szext	vshr
call	lop3	redux	tanh	vsub
clz	mad	rem	testp	vsub2
cnot	mad24	ret	tex	vsub4
copysign	madc	rsqrt	tld4	wgmma
cos	mapa	sad	trap	wmma
cp	match	selp	txq	xor
createpolicy	max	set	vabsdiff	
cvt	mbarrier	setmaxnreg	vabsdiff2	
cvta	membar	setp	vabsdiff4	

19

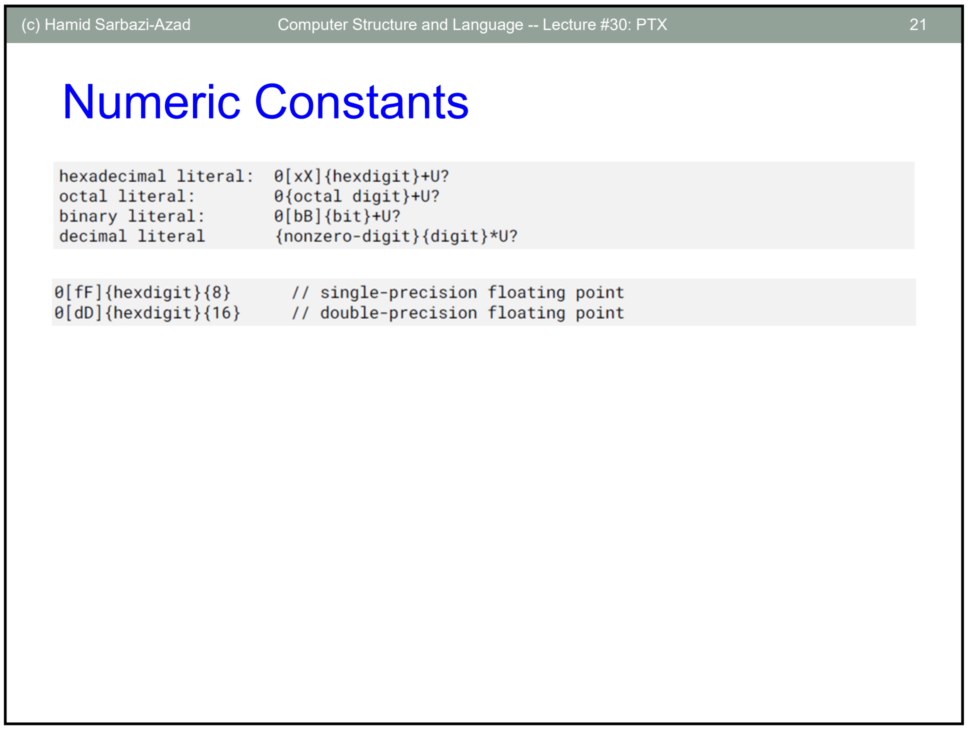


Identifiers

- WARP_SZ is a constant (warp size)
- Other identifiers are special registers (like tid for thread id) and start with %

%clock	%laneid	%lanemask_gt	%pm0, ..., %pm7
%clock64	%lanemask_eq	%nctaid	%smid
%ctaid	%lanemask_le	%ntid	%tid
%envreg<32>	%lanemask_lt	%nsmid	%warpid
%gridid	%lanemask_ge	%nwarpid	WARP_SZ

20

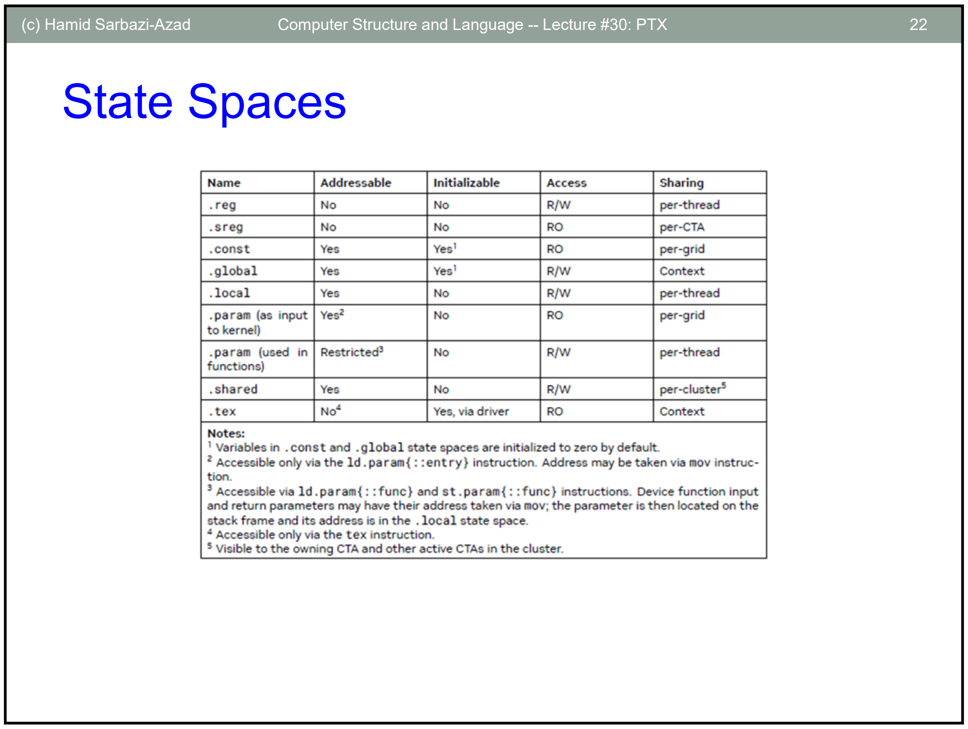


Numeric Constants

```
hexadecimal literal:  0[xX]{hexdigit}+U?
octal literal:        0{octal digit}+U?
binary literal:       0[bB]{bit}+U?
decimal literal       {nonzero-digit}{digit}*U?
```

```
0[fF]{hexdigit}{8}      // single-precision floating point
0[dD]{hexdigit}{16}     // double-precision floating point
```

21

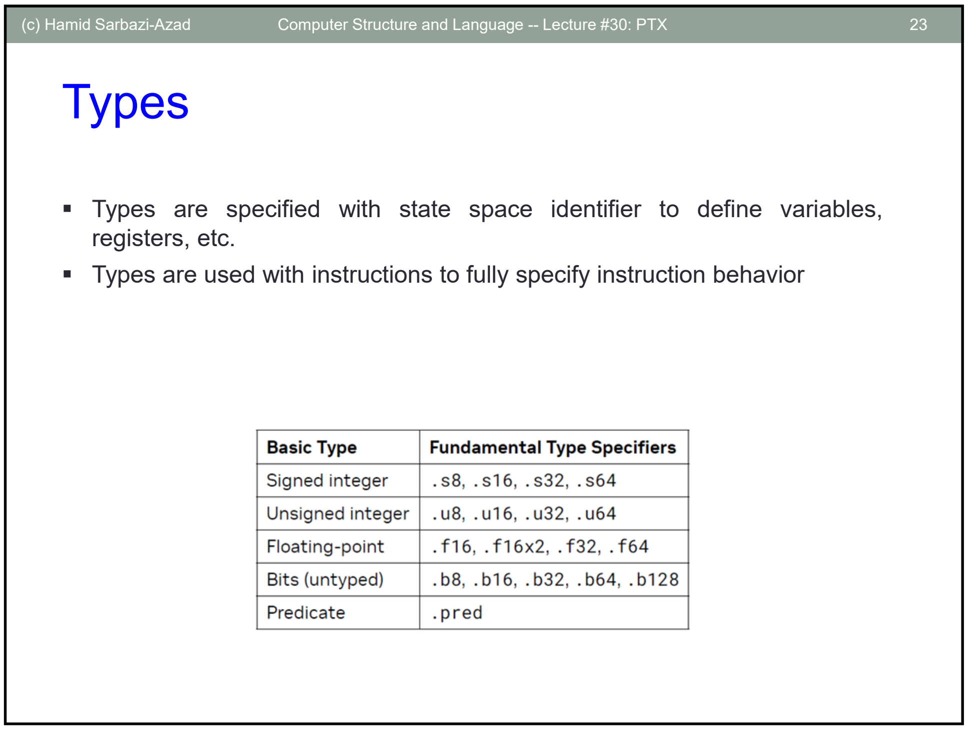


State Spaces

Name	Addressable	Initializable	Access	Sharing
.reg	No	No	R/W	per-thread
.sreg	No	No	RO	per-CTA
.const	Yes	Yes[1]	RO	per-grid
.global	Yes	Yes[1]	R/W	Context
.local	Yes	No	R/W	per-thread
.param (as input to kernel)	Yes[2]	No	RO	per-grid
.param (used in functions)	Restricted[3]	No	R/W	per-thread
.shared	Yes	No	R/W	per-cluster[5]
.tex	No[4]	Yes, via driver	RO	Context

Notes:
[1] Variables in .const and .global state spaces are initialized to zero by default.
[2] Accessible only via the ld.param{::entry} instruction. Address may be taken via mov instruction.
[3] Accessible via ld.param{::func} and st.param{::func} instructions. Device function input and return parameters may have their address taken via mov; the parameter is then located on the stack frame and its address is in the .local state space.
[4] Accessible only via the tex instruction.
[5] Visible to the owning CTA and other active CTAs in the cluster.

22

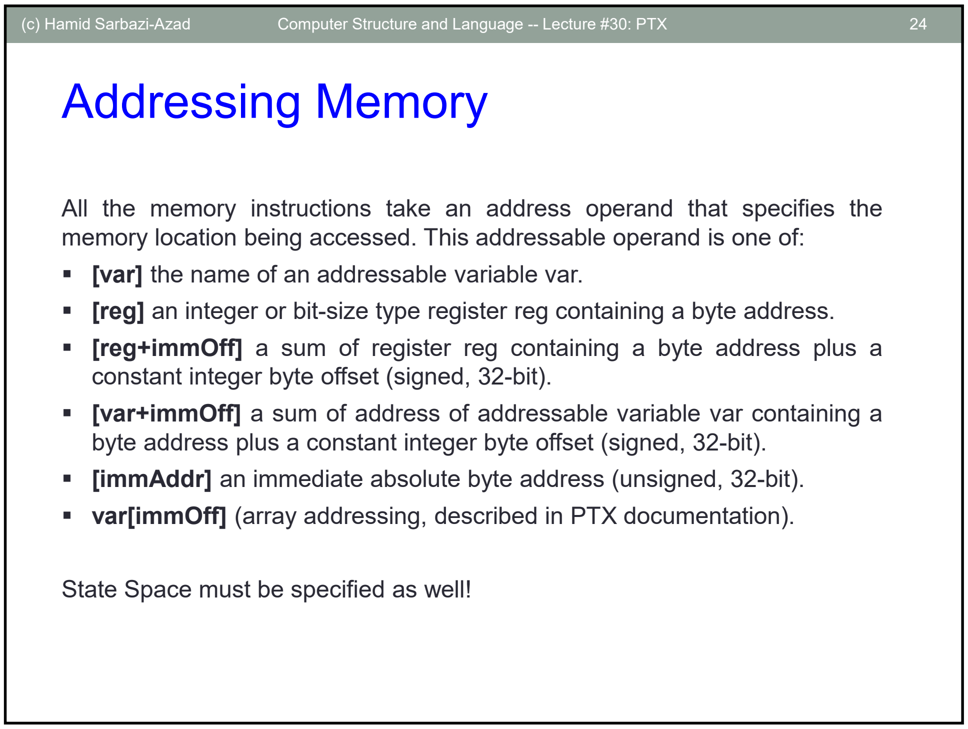


Types

- Types are specified with state space identifier to define variables, registers, etc.
- Types are used with instructions to fully specify instruction behavior

Basic Type	Fundamental Type Specifiers
Signed integer	.s8, .s16, .s32, .s64
Unsigned integer	.u8, .u16, .u32, .u64
Floating-point	.f16, .f16x2, .f32, .f64
Bits (untyped)	.b8, .b16, .b32, .b64, .b128
Predicate	.pred

23



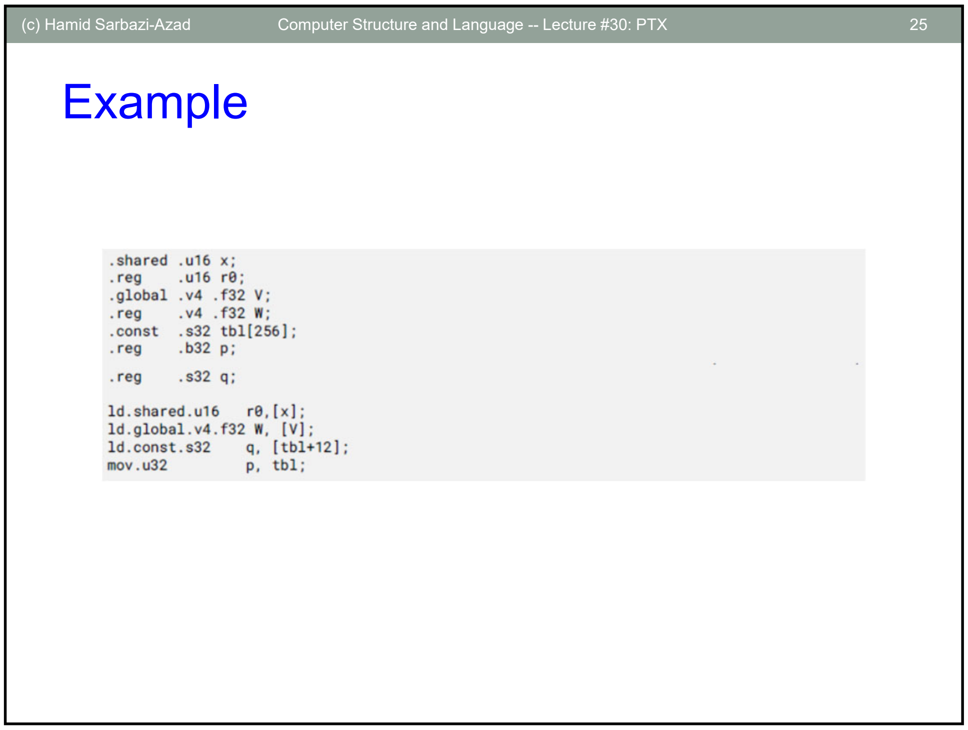
Addressing Memory

All the memory instructions take an address operand that specifies the memory location being accessed. This addressable operand is one of:

- **[var]** the name of an addressable variable var.
- **[reg]** an integer or bit-size type register reg containing a byte address.
- **[reg+immOff]** a sum of register reg containing a byte address plus a constant integer byte offset (signed, 32-bit).
- **[var+immOff]** a sum of address of addressable variable var containing a byte address plus a constant integer byte offset (signed, 32-bit).
- **[immAddr]** an immediate absolute byte address (unsigned, 32-bit).
- **var[immOff]** (array addressing, described in PTX documentation).

State Space must be specified as well!

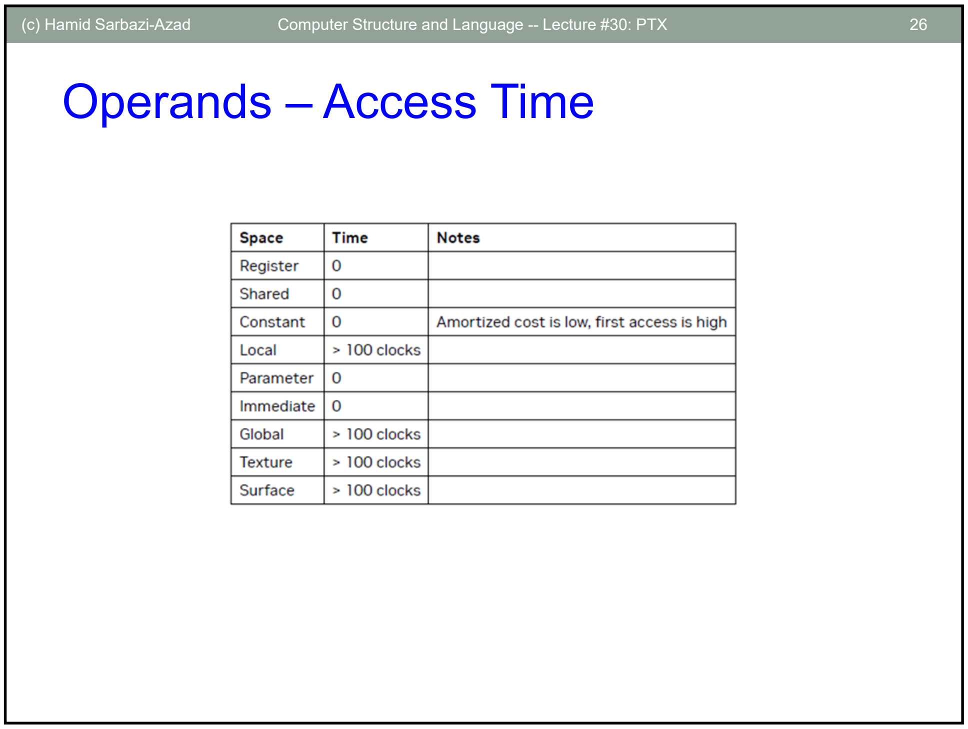
24



Example

```
.shared .u16 x;
.reg    .u16 r0;
.global .v4 .f32 V;
.reg    .v4 .f32 W;
.const  .s32 tbl[256];
.reg    .b32 p;

.reg    .s32 q;

ld.shared.u16   r0,[x];
ld.global.v4.f32 W, [V];
ld.const.s32    q, [tbl+12];
mov.u32         p, tbl;
```

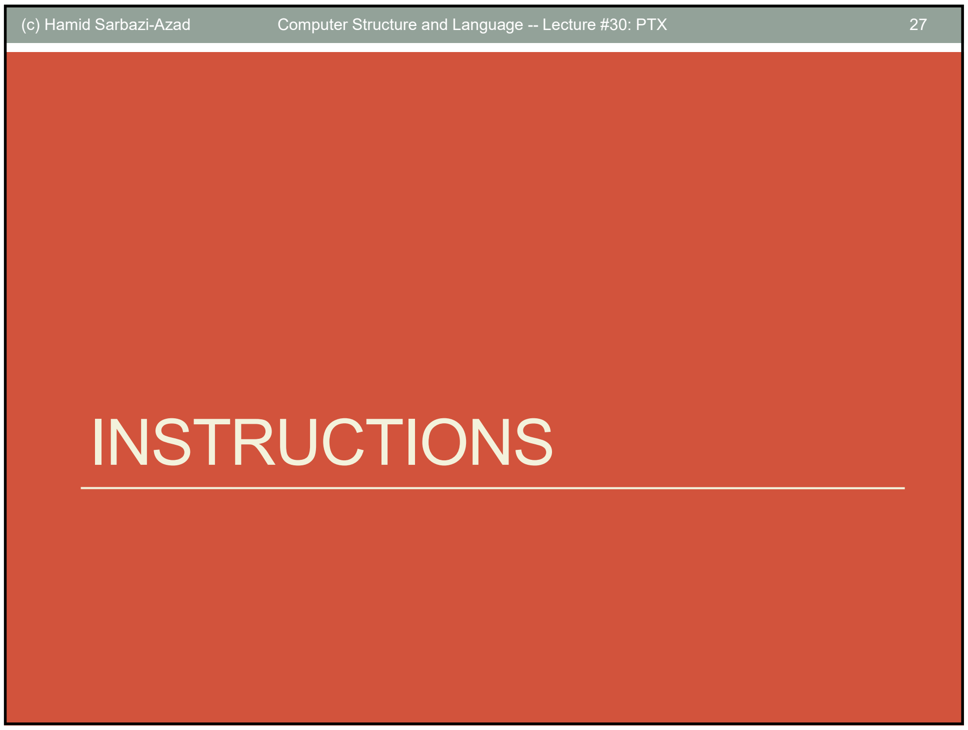
25



Operands – Access Time

Space	Time	Notes
Register	0	
Shared	0	
Constant	0	Amortized cost is low, first access is high
Local	> 100 clocks	
Parameter	0	
Immediate	0	
Global	> 100 clocks	
Texture	> 100 clocks	
Surface	> 100 clocks	

26


(c) Hamid Sarbazi-Azad Computer Structure and Language -- Lecture #30: PTX 27

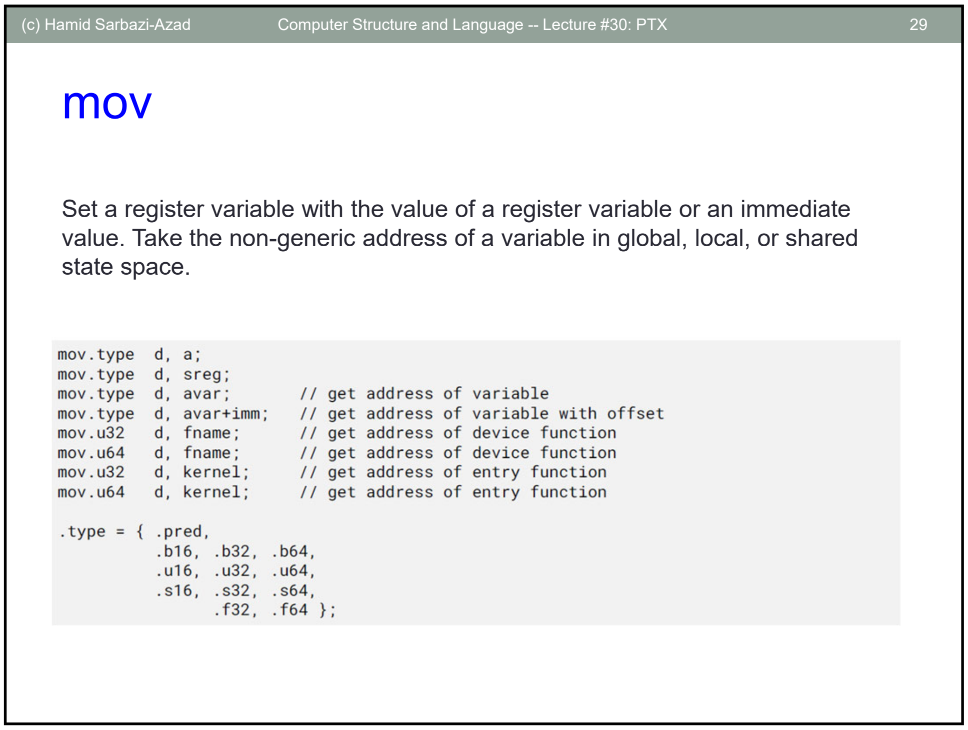
INSTRUCTIONS

27

(c) Hamid Sarbazi-Azad Computer Structure and Language -- Lecture #30: PTX 28

DATA MOVEMENT

28

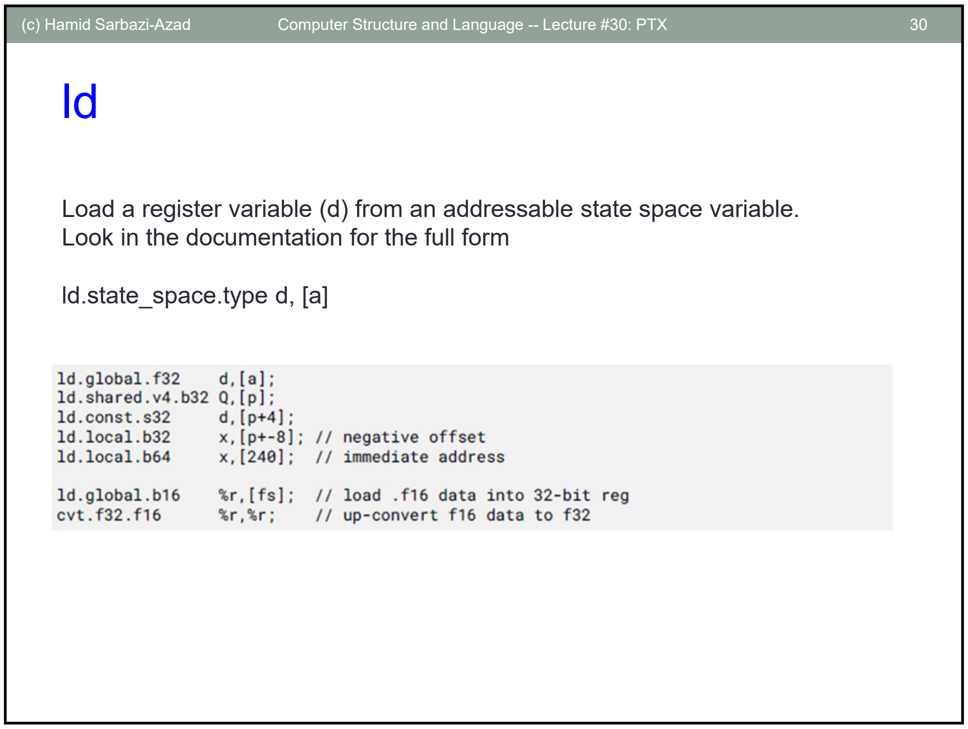


mov

Set a register variable with the value of a register variable or an immediate value. Take the non-generic address of a variable in global, local, or shared state space.

```
mov.type  d, a;
mov.type  d, sreg;
mov.type  d, avar;       // get address of variable
mov.type  d, avar+imm;   // get address of variable with offset
mov.u32   d, fname;      // get address of device function
mov.u64   d, fname;      // get address of device function
mov.u32   d, kernel;     // get address of entry function
mov.u64   d, kernel;     // get address of entry function

.type = { .pred,
          .b16, .b32, .b64,
          .u16, .u32, .u64,
          .s16, .s32, .s64,
                .f32, .f64 };
```

29



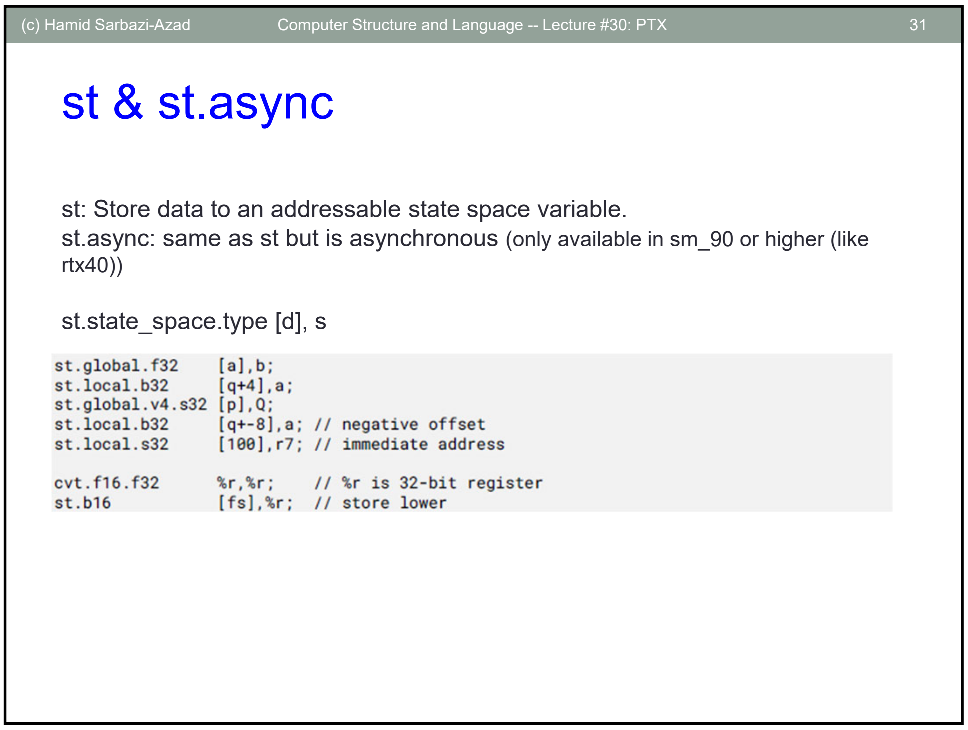
ld

Load a register variable (d) from an addressable state space variable.
Look in the documentation for the full form

ld.state_space.type d, [a]

```
ld.global.f32    d,[a];
ld.shared.v4.b32 Q,[p];
ld.const.s32     d,[p+4];
ld.local.b32     x,[p+-8]; // negative offset
ld.local.b64     x,[240];  // immediate address

ld.global.b16    %r,[fs];  // load .f16 data into 32-bit reg
cvt.f32.f16      %r,%r;    // up-convert f16 data to f32
```

30



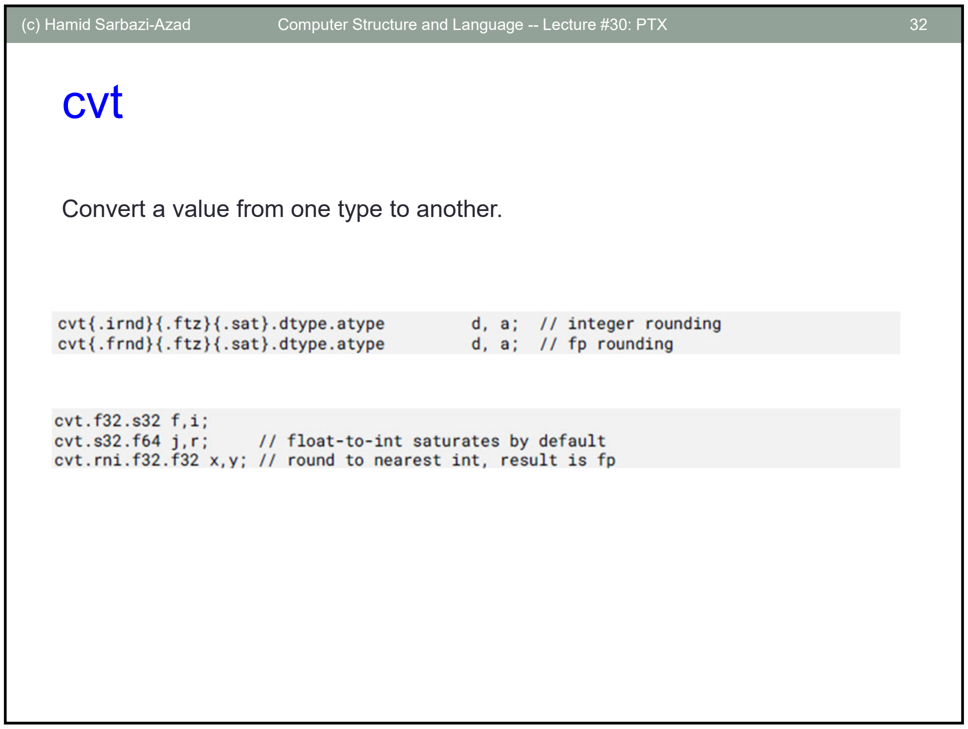
st & st.async

st: Store data to an addressable state space variable.
st.async: same as st but is asynchronous (only available in sm_90 or higher (like rtx40))

st.state_space.type [d], s

```
st.global.f32    [a],b;
st.local.b32     [q+4],a;
st.global.v4.s32 [p],Q;
st.local.b32     [q+-8],a; // negative offset
st.local.s32     [100],r7; // immediate address

cvt.f16.f32      %r,%r;    // %r is 32-bit register
st.b16           [fs],%r;  // store lower
```

31



cvt

Convert a value from one type to another.

```
cvt{.irnd}{.ftz}{.sat}.dtype.atype         d, a;  // integer rounding
cvt{.frnd}{.ftz}{.sat}.dtype.atype         d, a;  // fp rounding
```

```
cvt.f32.s32 f,i;
cvt.s32.f64 j,r;     // float-to-int saturates by default
cvt.rni.f32.f32 x,y; // round to nearest int, result is fp
```

32

cvt - notes

Integer rounding is required for float-to-integer conversions, and for same-size float-to-float conversions where the value is rounded to an integer. Integer rounding is illegal in all other instances.

Integer rounding modifiers:

- **.rni** round to nearest integer, choosing even integer if source is equidistant between two integers
- **.rzi** round to nearest integer in the direction of zero
- **.rmi** round to nearest integer in direction of negative infinity
- **.rpi** round to nearest integer in direction of positive infinity

Saturation modifier:

- **.sat** For integer destination types, .sat limits the result to MININT..MAXINT for the size of the operation. Note that saturation applies to both signed and unsigned integer types.
- **.sat** modifier is illegal in cases where saturation is not possible based on the source and destination types.

33

cvt – notes (cont.)

Floating-point rounding is required for float-to-float conversions that result in loss of precision, and for integer-to-float conversions. Floating-point rounding is illegal in all other instances.

Floating-point rounding modifiers:

- **.rn** mantissa LSB rounds to nearest even
- **.rna** mantissa LSB rounds to nearest, ties away from zero
- **.rz** mantissa LSB rounds towards zero
- **.rm** mantissa LSB rounds towards negative infinity
- **.rp** mantissa LSB rounds towards positive infinity

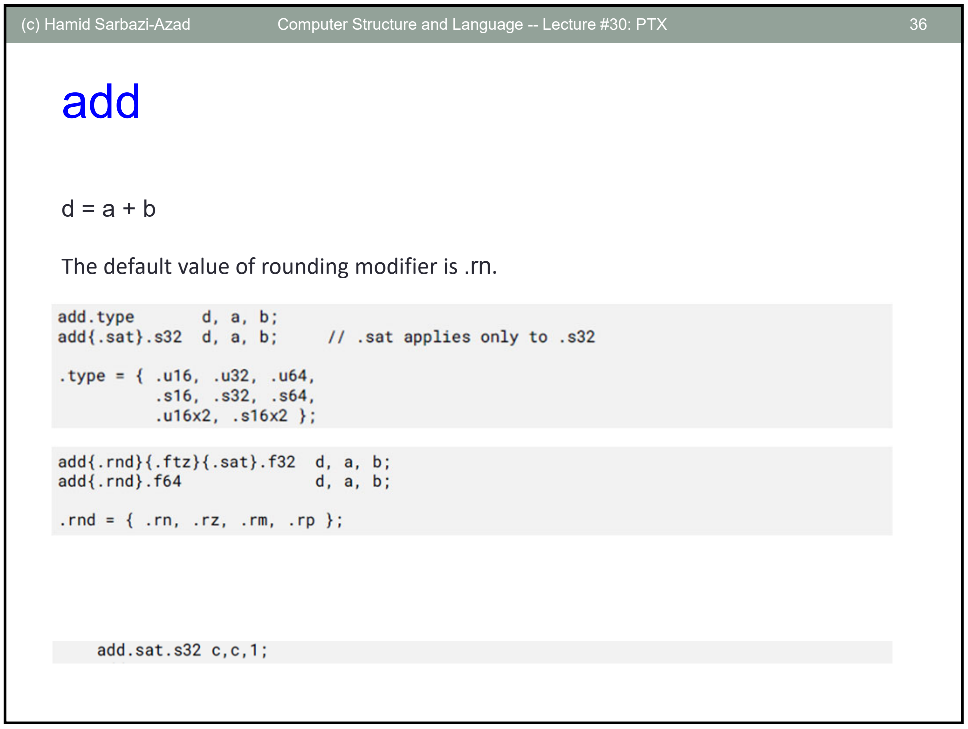
Saturation modifier:

- **.sat** For floating-point destination types, .sat limits the result to the range [0.0, 1.0]. NaN results are flushed to positive zero. Applies to .f16, .f32, and .f64 types.

34

ARITHMETIC

35



add

d = a + b

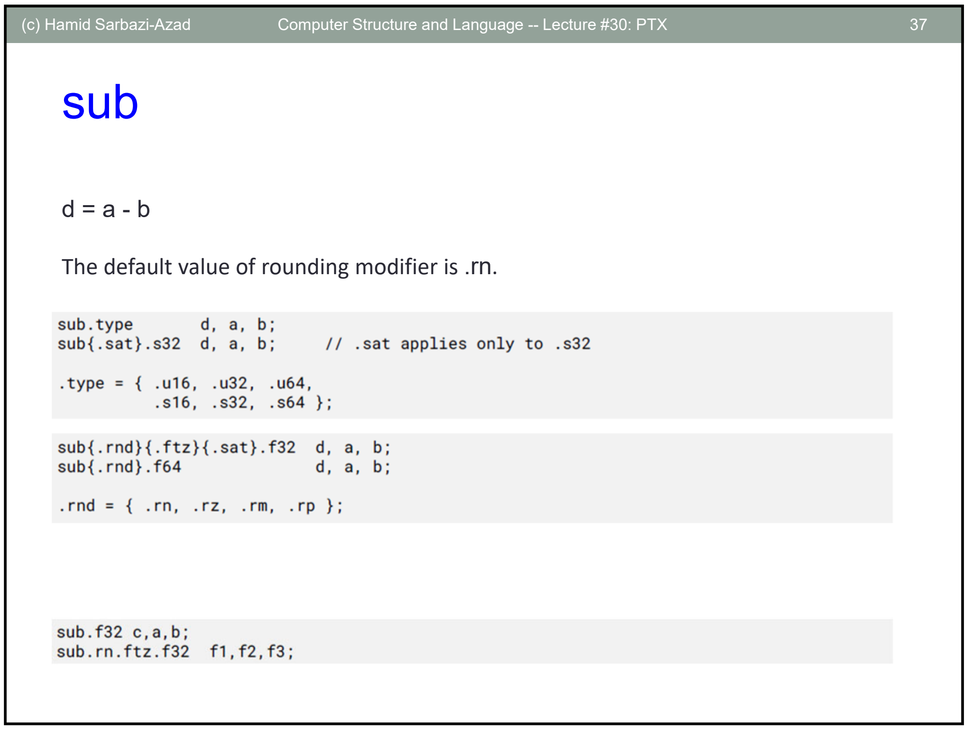
The default value of rounding modifier is .rn.

```
add.type       d, a, b;
add{.sat}.s32  d, a, b;     // .sat applies only to .s32

.type = { .u16, .u32, .u64,
          .s16, .s32, .s64,
          .u16x2, .s16x2 };
```

```
add{.rnd}{.ftz}{.sat}.f32  d, a, b;
add{.rnd}.f64              d, a, b;

.rnd = { .rn, .rz, .rm, .rp };
```

```
    add.sat.s32 c,c,1;
```

36



sub

d = a - b

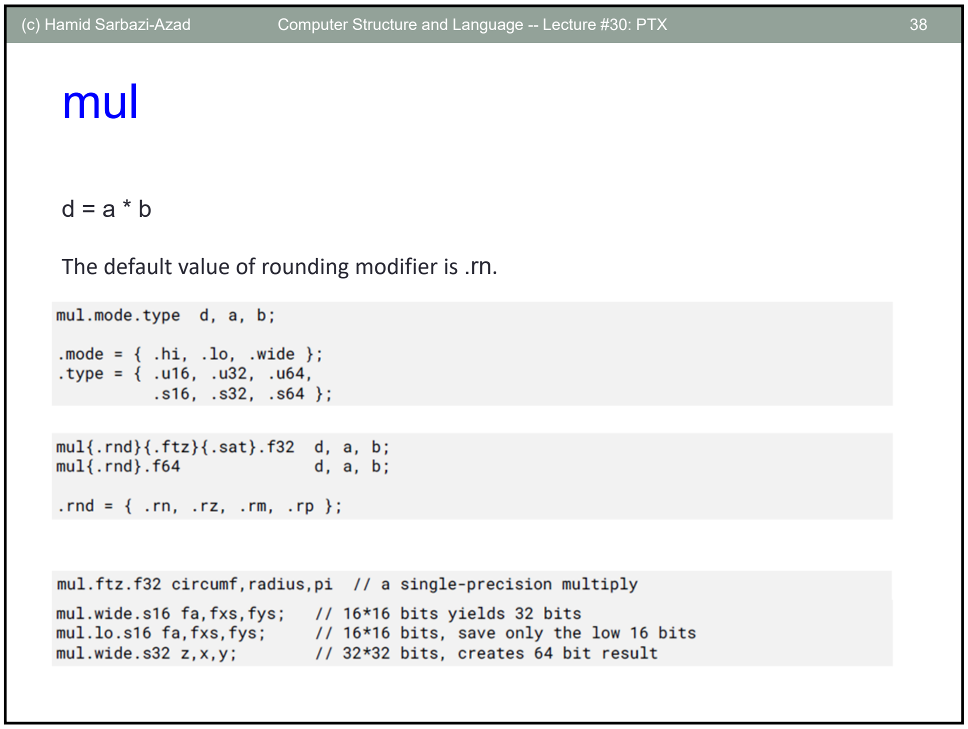
The default value of rounding modifier is .rn.

```
sub.type       d, a, b;
sub{.sat}.s32  d, a, b;     // .sat applies only to .s32

.type = { .u16, .u32, .u64,
          .s16, .s32, .s64 };
```

```
sub{.rnd}{.ftz}{.sat}.f32  d, a, b;
sub{.rnd}.f64              d, a, b;

.rnd = { .rn, .rz, .rm, .rp };
```

```
sub.f32 c,a,b;
sub.rn.ftz.f32  f1,f2,f3;
```

37



mul

d = a * b

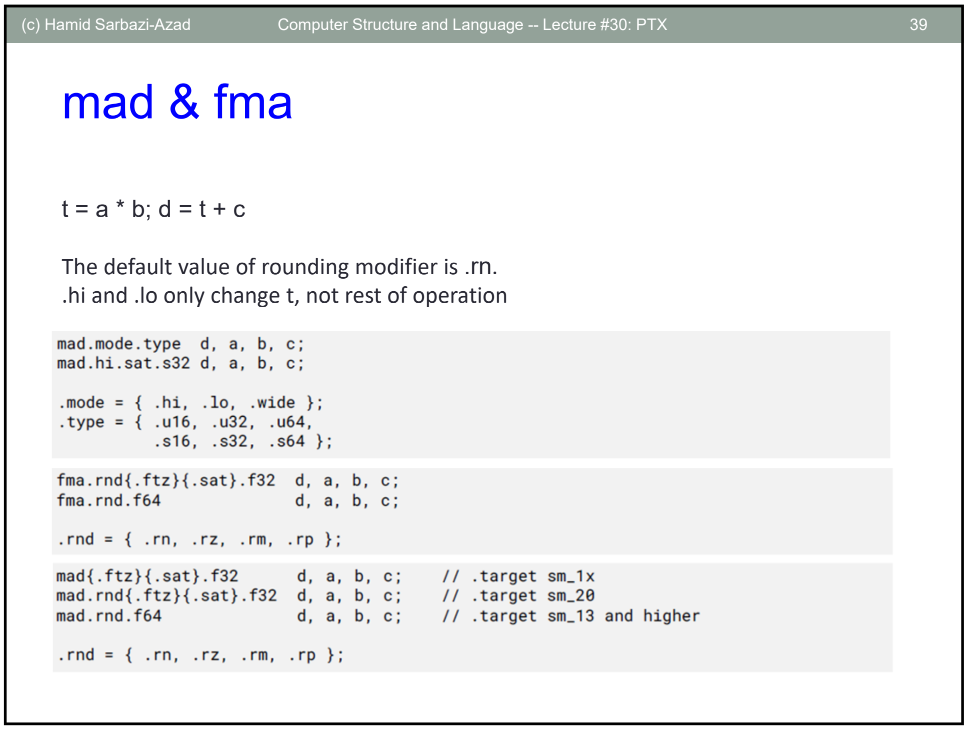
The default value of rounding modifier is .rn.

```
mul.mode.type  d, a, b;

.mode = { .hi, .lo, .wide };
.type = { .u16, .u32, .u64,
          .s16, .s32, .s64 };
```

```
mul{.rnd}{.ftz}{.sat}.f32  d, a, b;
mul{.rnd}.f64              d, a, b;

.rnd = { .rn, .rz, .rm, .rp };
```

```
mul.ftz.f32 circumf,radius,pi  // a single-precision multiply
mul.wide.s16 fa,fxs,fys;   // 16*16 bits yields 32 bits
mul.lo.s16 fa,fxs,fys;     // 16*16 bits, save only the low 16 bits
mul.wide.s32 z,x,y;        // 32*32 bits, creates 64 bit result
```

38



mad & fma

t = a * b; d = t + c

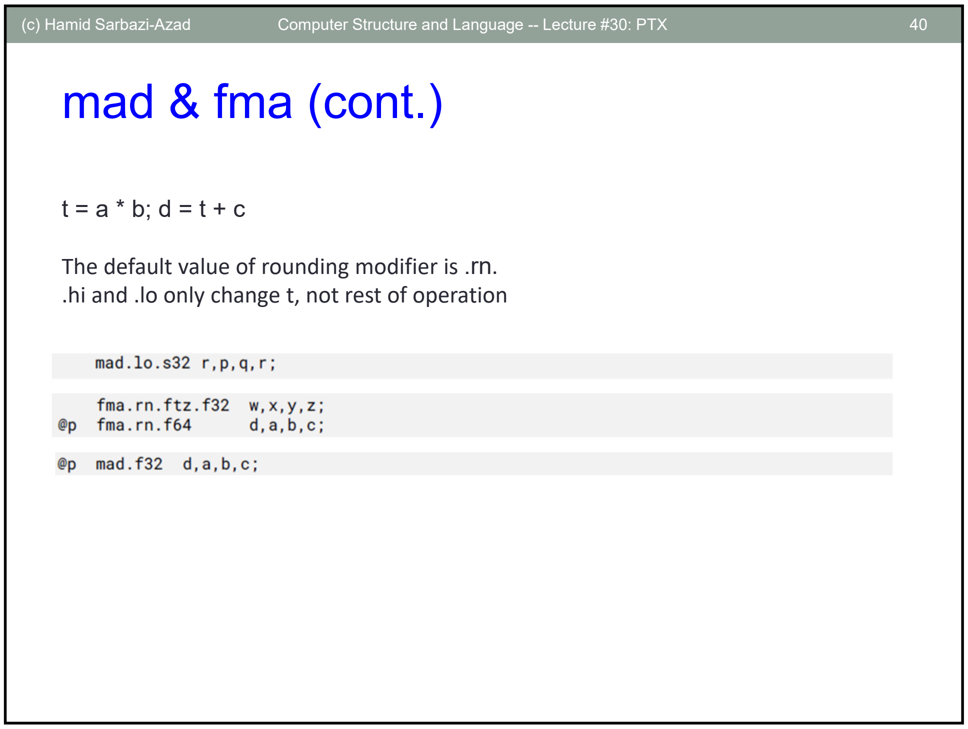
The default value of rounding modifier is .rn.
.hi and .lo only change t, not rest of operation

```
mad.mode.type  d, a, b, c;
mad.hi.sat.s32 d, a, b, c;

.mode = { .hi, .lo, .wide };
.type = { .u16, .u32, .u64,
          .s16, .s32, .s64 };
```

```
fma.rnd{.ftz}{.sat}.f32  d, a, b, c;
fma.rnd.f64              d, a, b, c;

.rnd = { .rn, .rz, .rm, .rp };
```

```
mad{.ftz}{.sat}.f32      d, a, b, c;    // .target sm_1x
mad.rnd{.ftz}{.sat}.f32  d, a, b, c;    // .target sm_20
mad.rnd.f64              d, a, b, c;    // .target sm_13 and higher

.rnd = { .rn, .rz, .rm, .rp };
```

39



mad & fma (cont.)

t = a * b; d = t + c

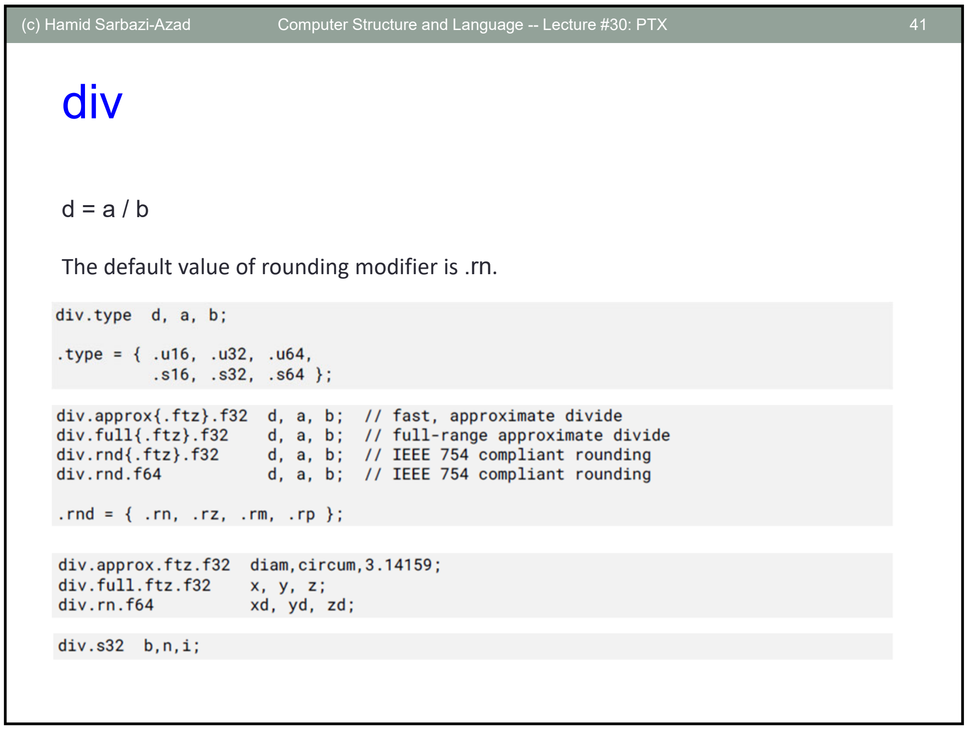
The default value of rounding modifier is .rn.
.hi and .lo only change t, not rest of operation

```
    mad.lo.s32 r,p,q,r;
```

```
    fma.rn.ftz.f32  w,x,y,z;
@p  fma.rn.f64      d,a,b,c;
```

```
@p  mad.f32  d,a,b,c;
```

40



div

d = a / b

The default value of rounding modifier is .rn.

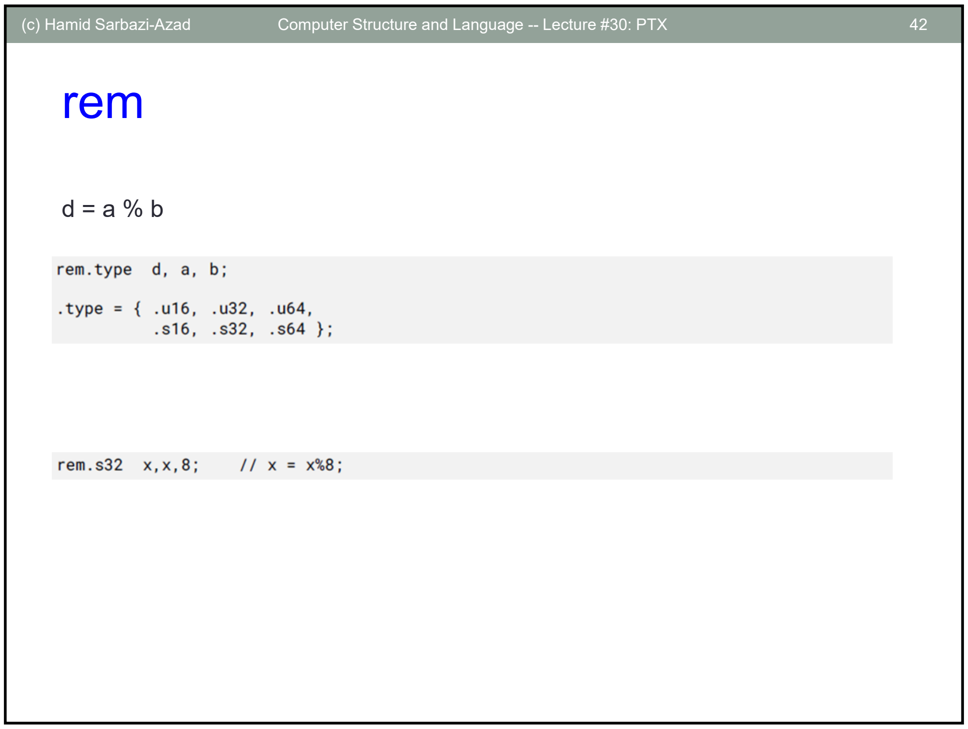
```
div.type  d, a, b;

.type = { .u16, .u32, .u64,
          .s16, .s32, .s64 };
```

```
div.approx{.ftz}.f32  d, a, b;  // fast, approximate divide
div.full{.ftz}.f32    d, a, b;  // full-range approximate divide
div.rnd{.ftz}.f32     d, a, b;  // IEEE 754 compliant rounding
div.rnd.f64           d, a, b;  // IEEE 754 compliant rounding

.rnd = { .rn, .rz, .rm, .rp };
```

```
div.approx.ftz.f32  diam,circum,3.14159;
div.full.ftz.f32    x, y, z;
div.rn.f64          xd, yd, zd;
```

```
div.s32  b,n,i;
```

41



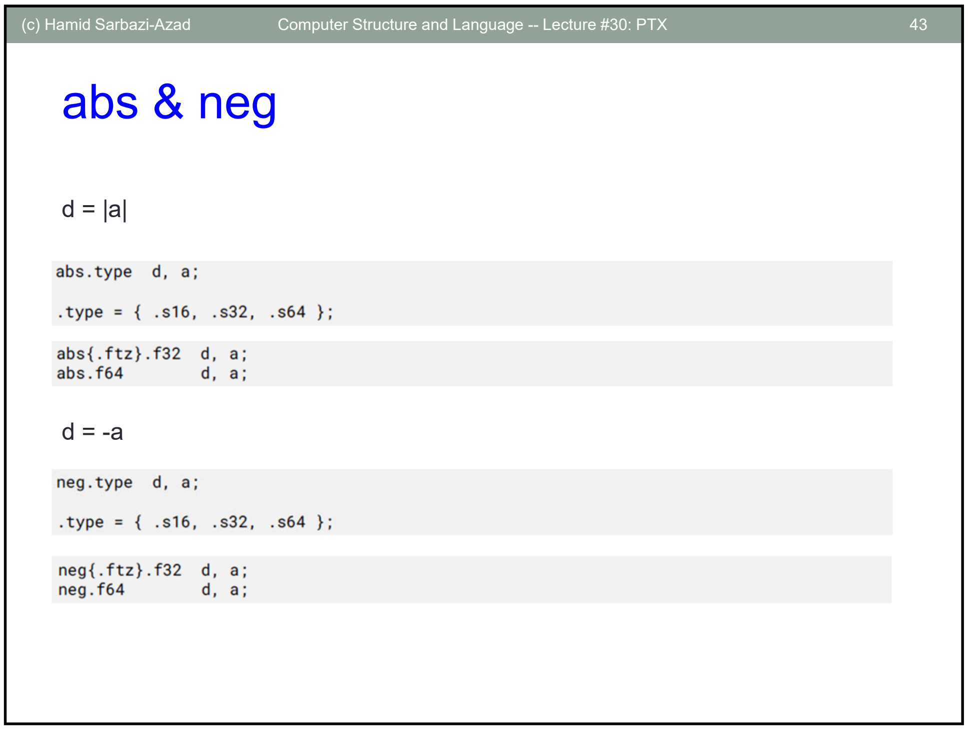
rem

d = a % b

```
rem.type  d, a, b;

.type = { .u16, .u32, .u64,
          .s16, .s32, .s64 };
```

```
rem.s32  x,x,8;    // x = x%8;
```

42



abs & neg

d = |a|

```
abs.type  d, a;

.type = { .s16, .s32, .s64 };
```

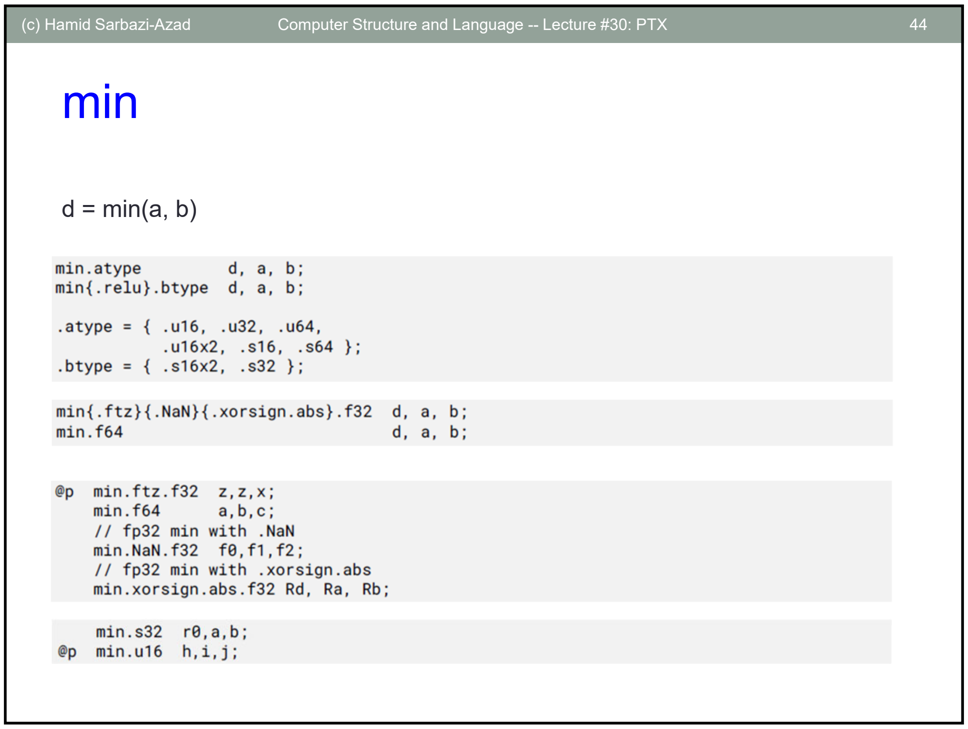
```
abs{.ftz}.f32  d, a;
abs.f64        d, a;
```

d = -a

```
neg.type  d, a;

.type = { .s16, .s32, .s64 };
```

```
neg{.ftz}.f32  d, a;
neg.f64        d, a;
```

43



min

d = min(a, b)

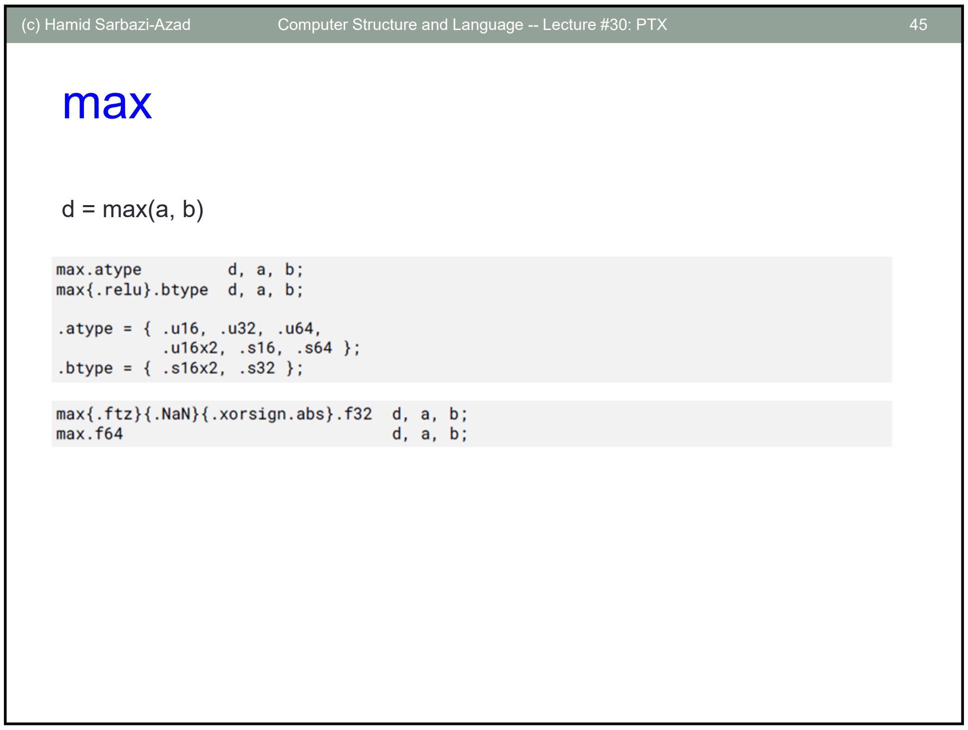
```
min.atype         d, a, b;
min{.relu}.btype  d, a, b;

.atype = { .u16, .u32, .u64,
           .u16x2, .s16, .s64 };
.btype = { .s16x2, .s32 };
```

```
min{.ftz}{.NaN}{.xorsign.abs}.f32  d, a, b;
min.f64                            d, a, b;
```

```
@p  min.ftz.f32  z,z,x;
    min.f64      a,b,c;
    // fp32 min with .NaN
    min.NaN.f32  f0,f1,f2;
    // fp32 min with .xorsign.abs
    min.xorsign.abs.f32 Rd, Ra, Rb;
```

```
    min.s32  r0,a,b;
@p  min.u16  h,i,j;
```

44



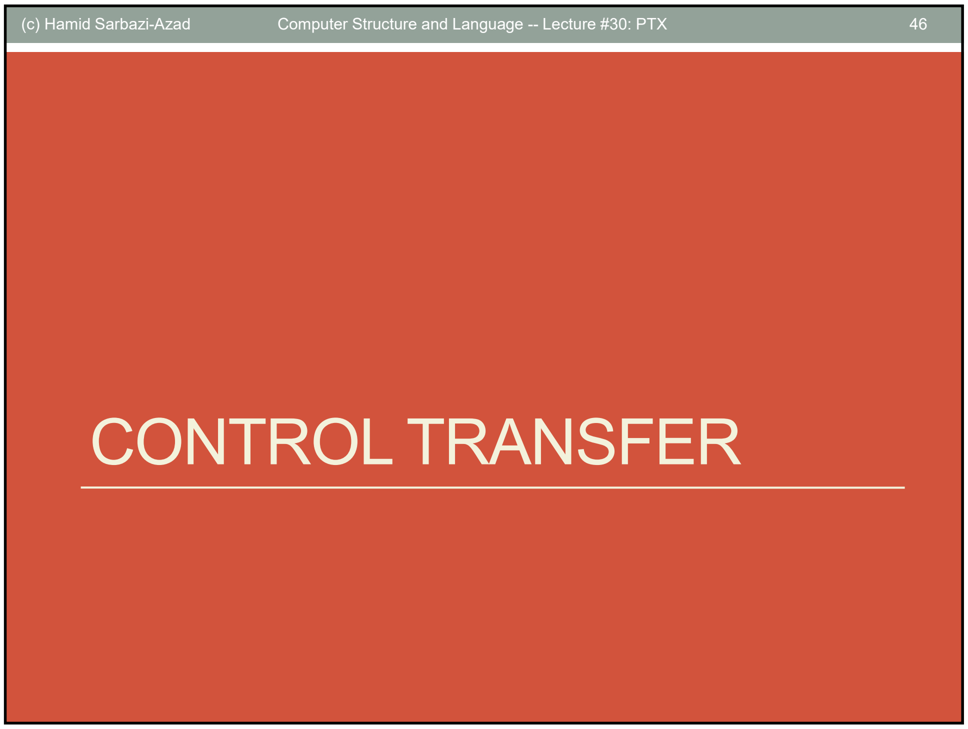
max

d = max(a, b)

```
max.atype         d, a, b;
max{.relu}.btype  d, a, b;

.atype = { .u16, .u32, .u64,
           .u16x2, .s16, .s64 };
.btype = { .s16x2, .s32 };
```

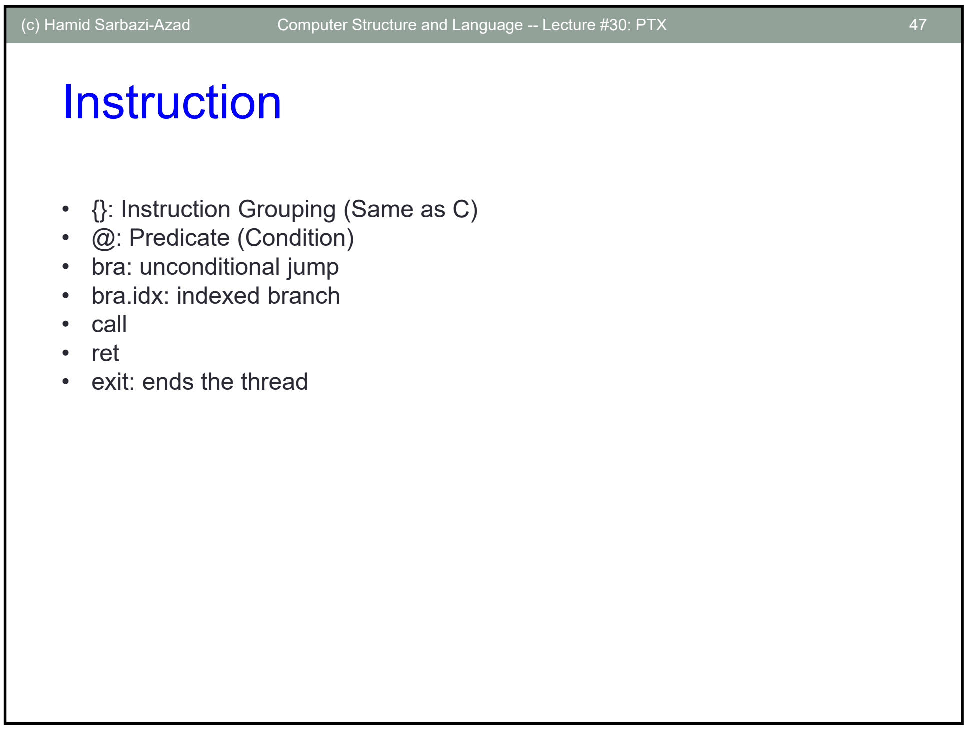
```
max{.ftz}{.NaN}{.xorsign.abs}.f32  d, a, b;
max.f64                            d, a, b;
```

45



CONTROL TRANSFER

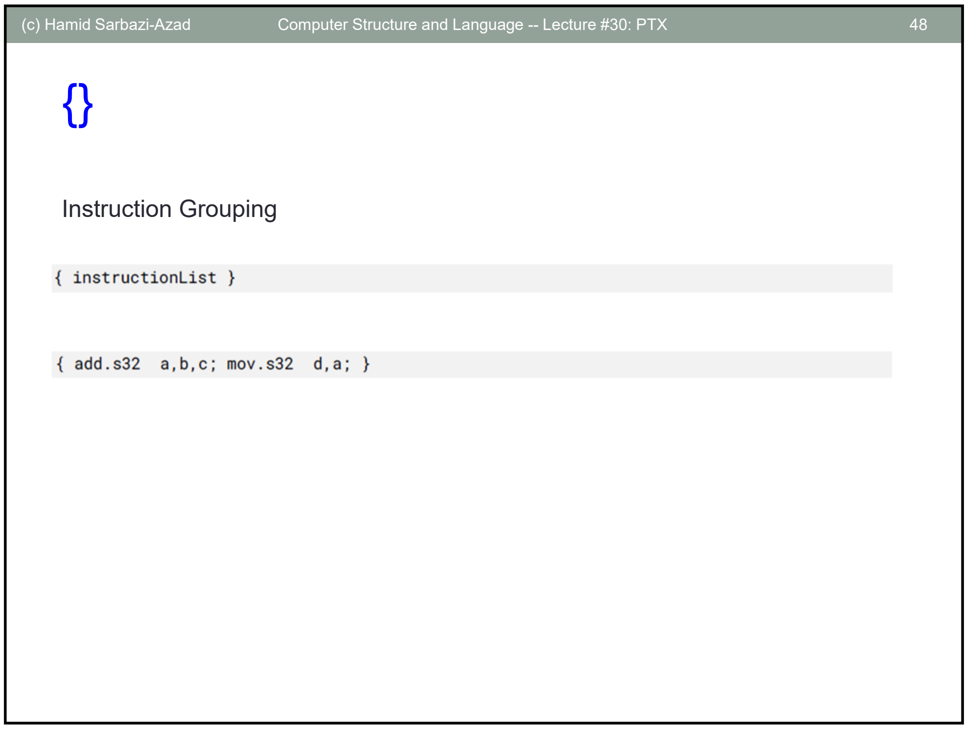
46



Instruction

- {}: Instruction Grouping (Same as C)
- @: Predicate (Condition)
- bra: unconditional jump
- bra.idx: indexed branch
- call
- ret
- exit: ends the thread

47



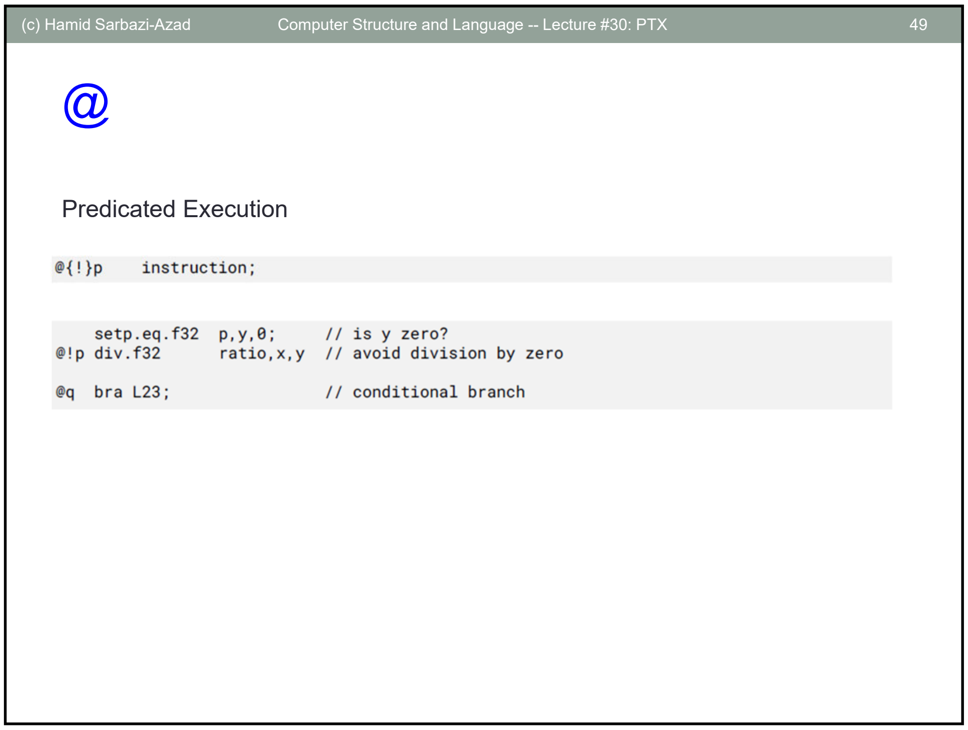
{}

Instruction Grouping

```
{ instructionList }
```

```
{ add.s32  a,b,c; mov.s32  d,a; }
```

48



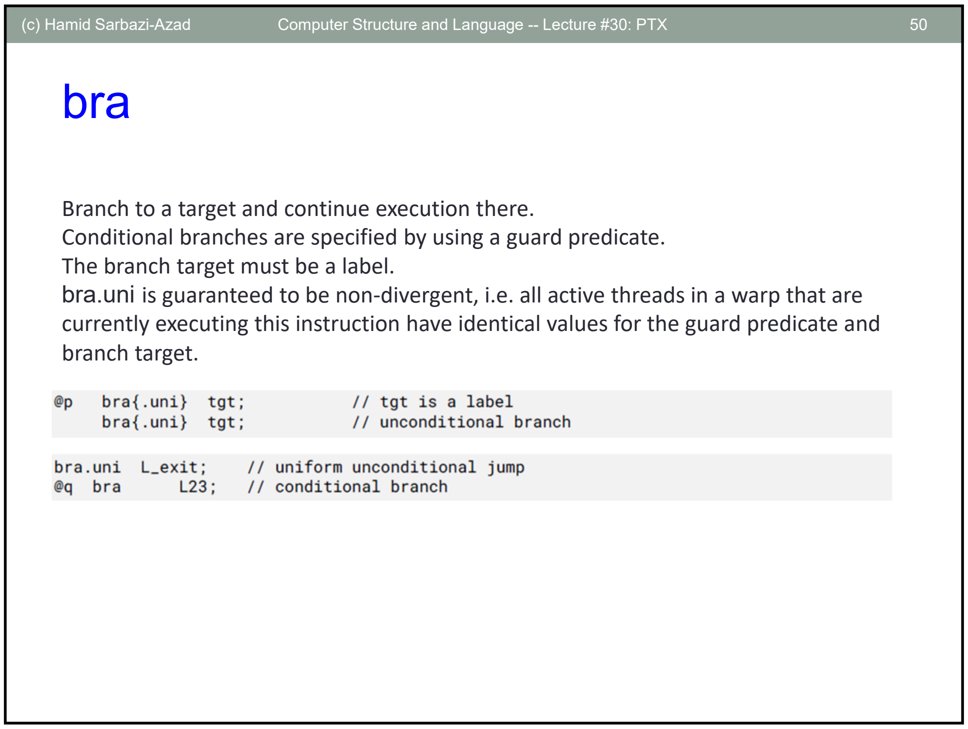
@

Predicated Execution

```
@{!}p    instruction;
```

```
    setp.eq.f32  p,y,0;     // is y zero?
@!p div.f32      ratio,x,y  // avoid division by zero

@q  bra L23;                // conditional branch
```

49



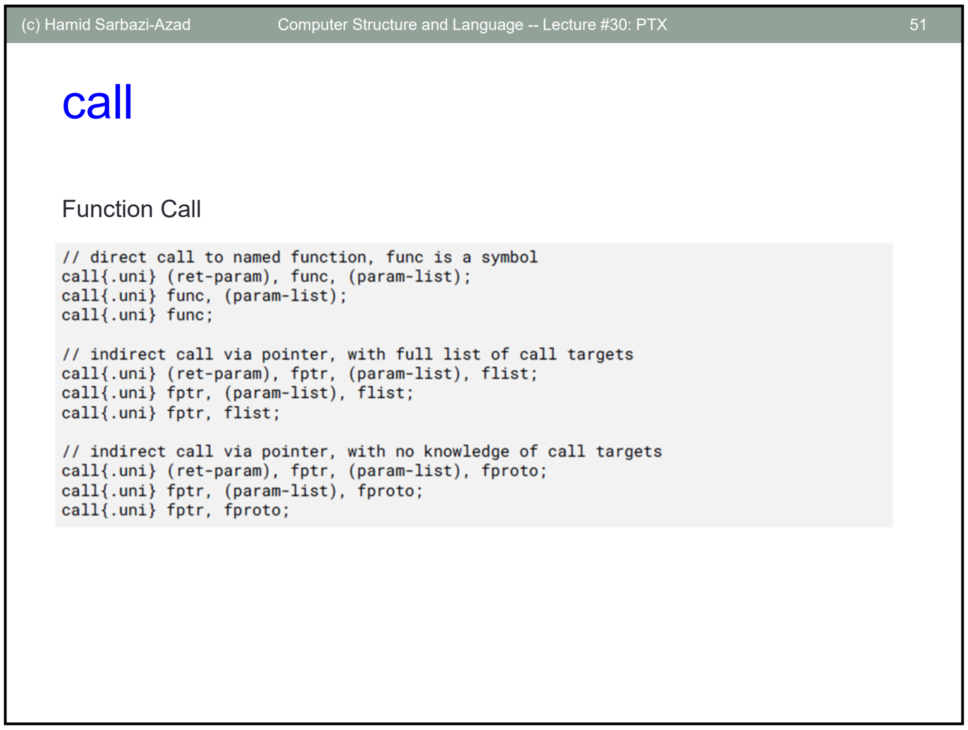
bra

Branch to a target and continue execution there.
Conditional branches are specified by using a guard predicate.
The branch target must be a label.
bra.uni is guaranteed to be non-divergent, i.e. all active threads in a warp that are currently executing this instruction have identical values for the guard predicate and branch target.

```
@p   bra{.uni}  tgt;           // tgt is a label
     bra{.uni}  tgt;           // unconditional branch
```

```
bra.uni  L_exit;    // uniform unconditional jump
@q  bra      L23;   // conditional branch
```

50

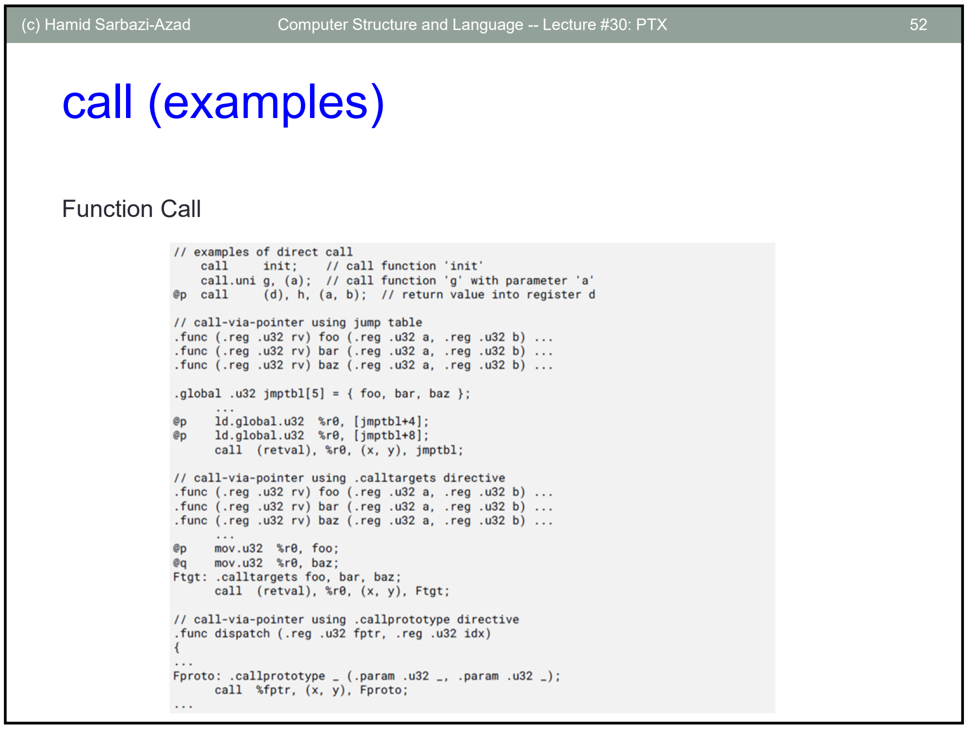


call

Function Call

```
// direct call to named function, func is a symbol
call{.uni} (ret-param), func, (param-list);
call{.uni} func, (param-list);
call{.uni} func;

// indirect call via pointer, with full list of call targets
call{.uni} (ret-param), fptr, (param-list), flist;
call{.uni} fptr, (param-list), flist;
call{.uni} fptr, flist;

// indirect call via pointer, with no knowledge of call targets
call{.uni} (ret-param), fptr, (param-list), fproto;
call{.uni} fptr, (param-list), fproto;
call{.uni} fptr, fproto;
```

51

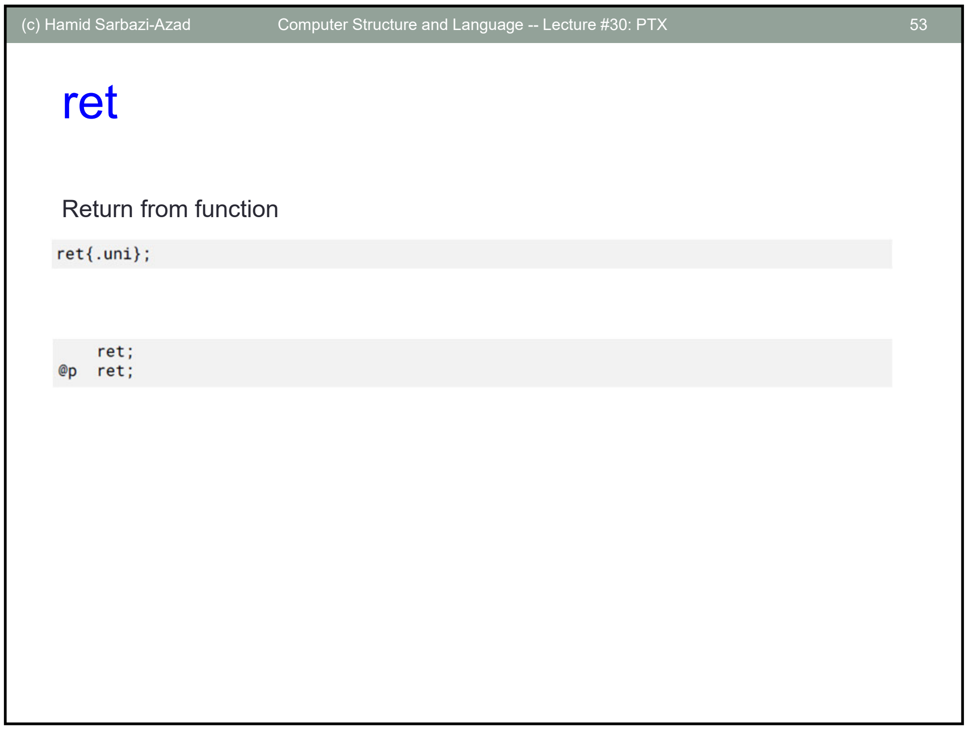


call (examples)

Function Call

```
// examples of direct call
    call     init;    // call function 'init'
    call.uni g, (a);  // call function 'g' with parameter 'a'
@p  call     (d), h, (a, b);  // return value into register d

// call-via-pointer using jump table
.func (.reg .u32 rv) foo (.reg .u32 a, .reg .u32 b) ...
.func (.reg .u32 rv) bar (.reg .u32 a, .reg .u32 b) ...
.func (.reg .u32 rv) baz (.reg .u32 a, .reg .u32 b) ...

.global .u32 jmptbl[5] = { foo, bar, baz };
      ...
@p    ld.global.u32  %r0, [jmptbl+4];
@p    ld.global.u32  %r0, [jmptbl+8];
      call  (retval), %r0, (x, y), jmptbl;

// call-via-pointer using .calltargets directive
.func (.reg .u32 rv) foo (.reg .u32 a, .reg .u32 b) ...
.func (.reg .u32 rv) bar (.reg .u32 a, .reg .u32 b) ...
.func (.reg .u32 rv) baz (.reg .u32 a, .reg .u32 b) ...
      ...
@p    mov.u32  %r0, foo;
@q    mov.u32  %r0, baz;
Ftgt: .calltargets foo, bar, baz;
      call  (retval), %r0, (x, y), Ftgt;

// call-via-pointer using .callprototype directive
.func dispatch (.reg .u32 fptr, .reg .u32 idx)
{
...
Fproto: .callprototype _ (.param .u32 _, .param .u32 _);
      call  %fptr, (x, y), Fproto;
...
```

52



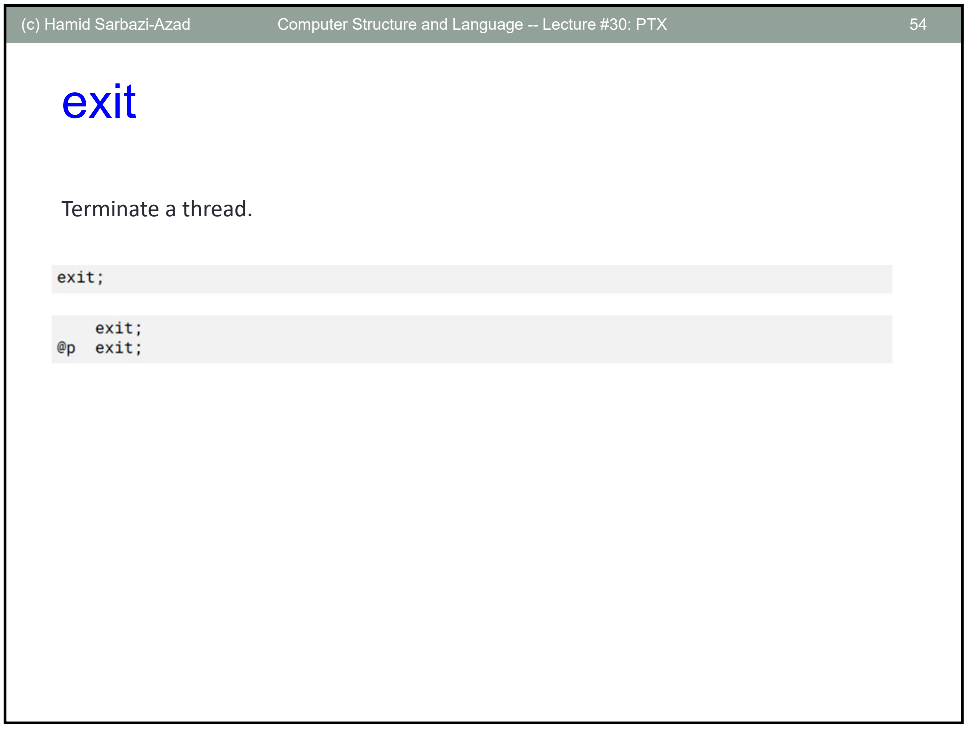
ret

Return from function

```
ret{.uni};
```

```
    ret;
@p  ret;
```

53



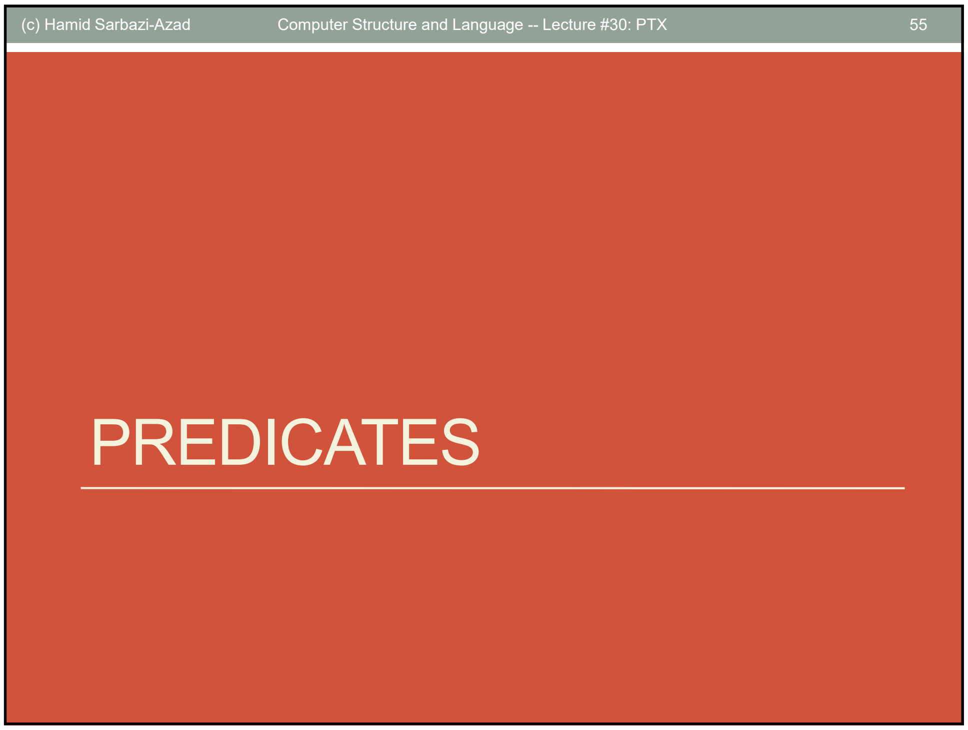
exit

Terminate a thread.

```
exit;
```

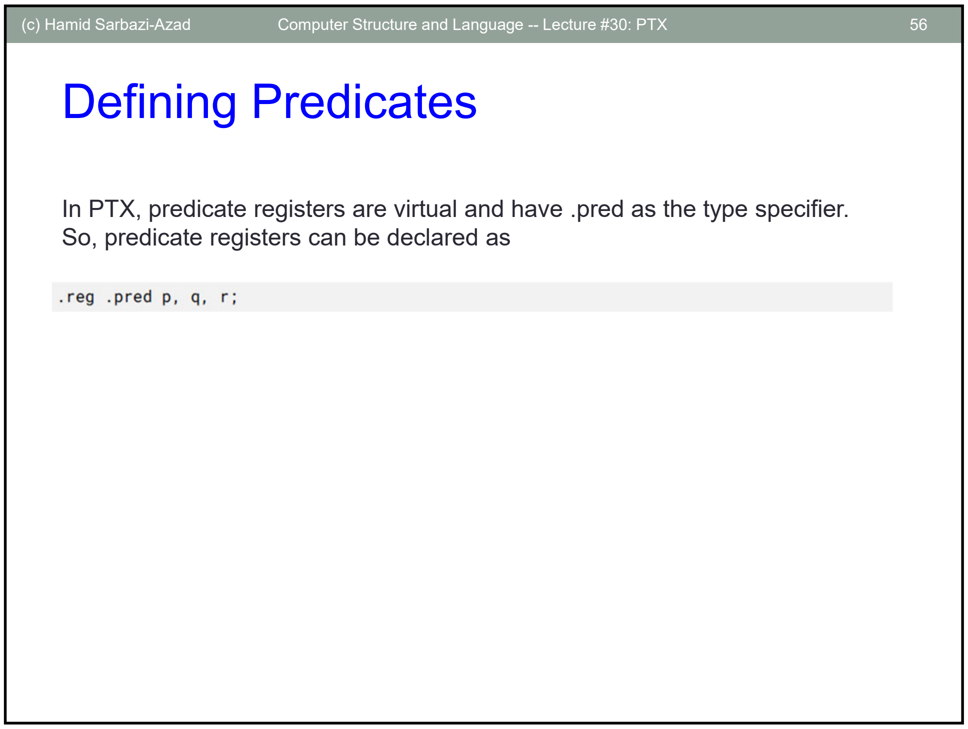
```
    exit;
@p  exit;
```

54



PREDICATES

55

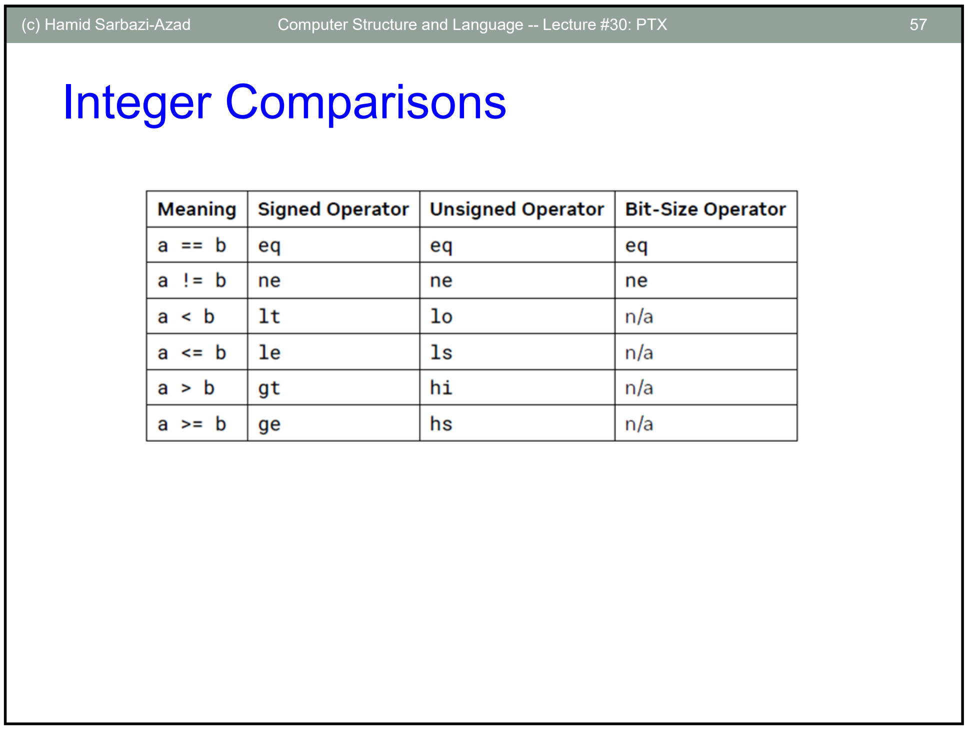


Defining Predicates

In PTX, predicate registers are virtual and have .pred as the type specifier. So, predicate registers can be declared as

```
.reg .pred p, q, r;
```

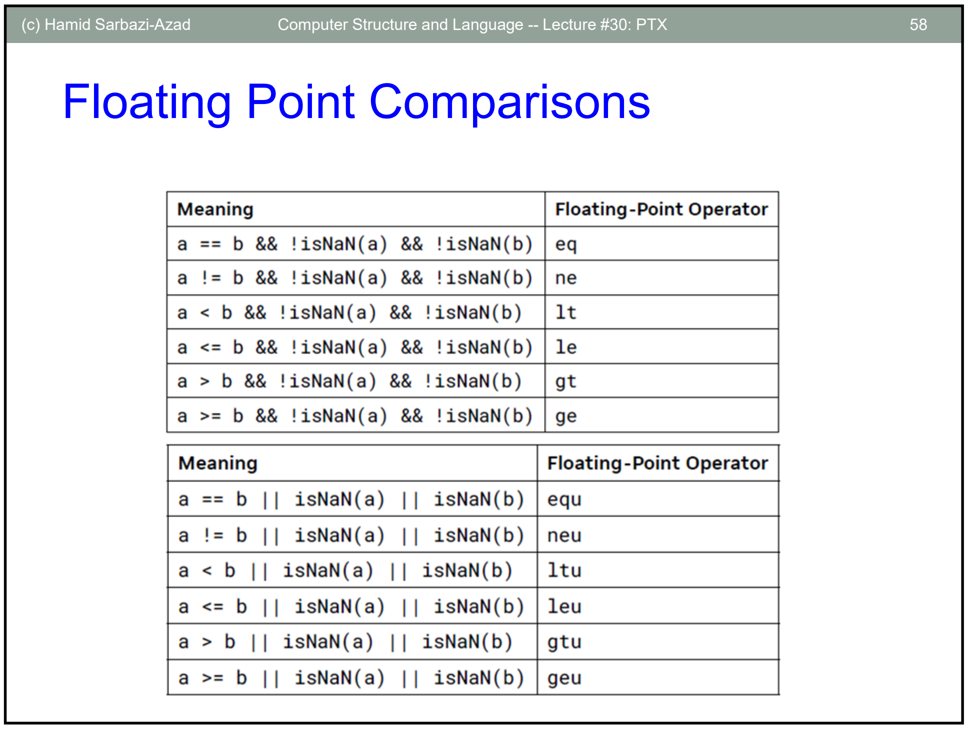
56



Integer Comparisons

Meaning	Signed Operator	Unsigned Operator	Bit-Size Operator
`a == b`	`eq`	`eq`	`eq`
`a != b`	`ne`	`ne`	`ne`
`a < b`	`lt`	`lo`	n/a
`a <= b`	`le`	`ls`	n/a
`a > b`	`gt`	`hi`	n/a
`a >= b`	`ge`	`hs`	n/a

57

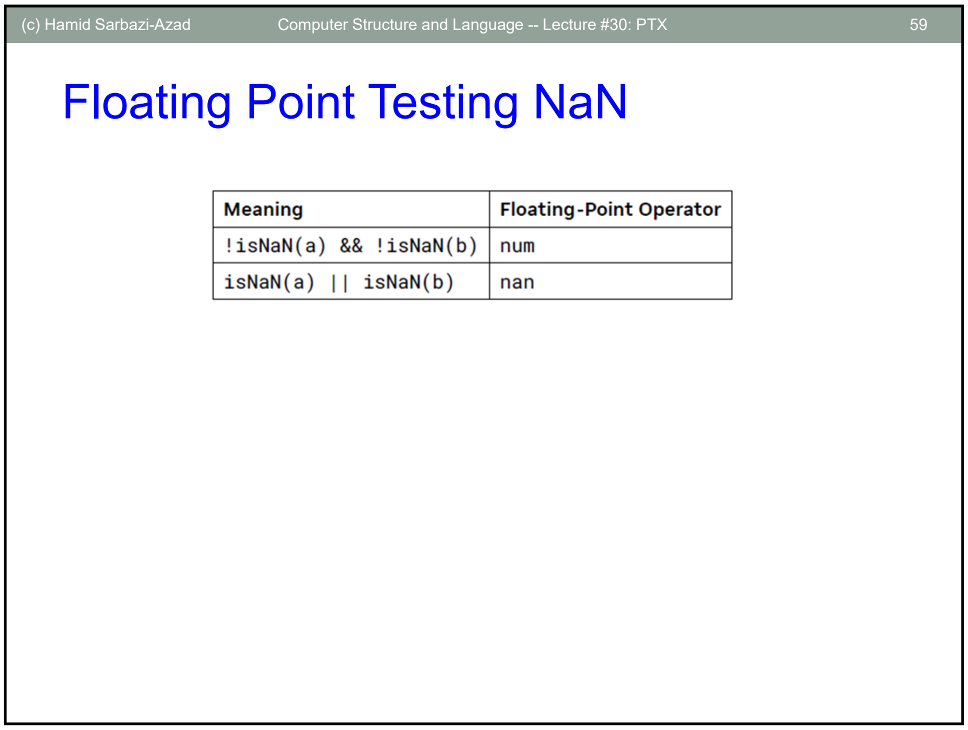


Floating Point Comparisons

Meaning	Floating-Point Operator
`a == b && !isNaN(a) && !isNaN(b)`	`eq`
`a != b && !isNaN(a) && !isNaN(b)`	`ne`
`a < b && !isNaN(a) && !isNaN(b)`	`lt`
`a <= b && !isNaN(a) && !isNaN(b)`	`le`
`a > b && !isNaN(a) && !isNaN(b)`	`gt`
`a >= b && !isNaN(a) && !isNaN(b)`	`ge`

Meaning	Floating-Point Operator
`a == b || isNaN(a) || isNaN(b)`	`equ`
`a != b || isNaN(a) || isNaN(b)`	`neu`
`a < b || isNaN(a) || isNaN(b)`	`ltu`
`a <= b || isNaN(a) || isNaN(b)`	`leu`
`a > b || isNaN(a) || isNaN(b)`	`gtu`
`a >= b || isNaN(a) || isNaN(b)`	`geu`

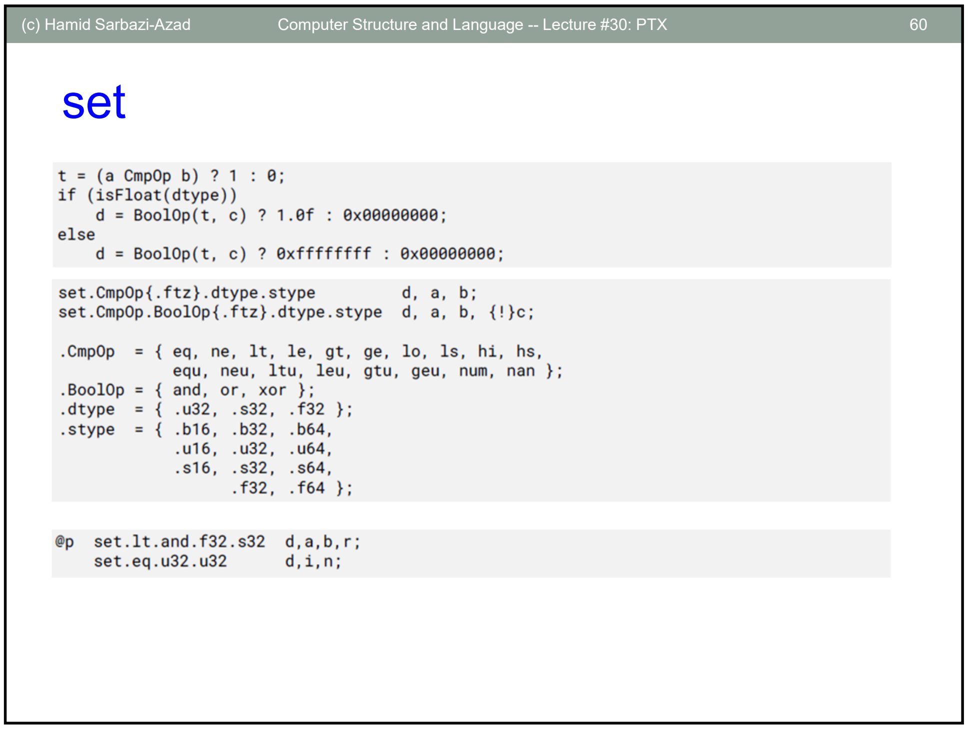
58



Floating Point Testing NaN

Meaning	Floating-Point Operator
`!isNaN(a) && !isNaN(b)`	num
`isNaN(a) || isNaN(b)`	nan

59



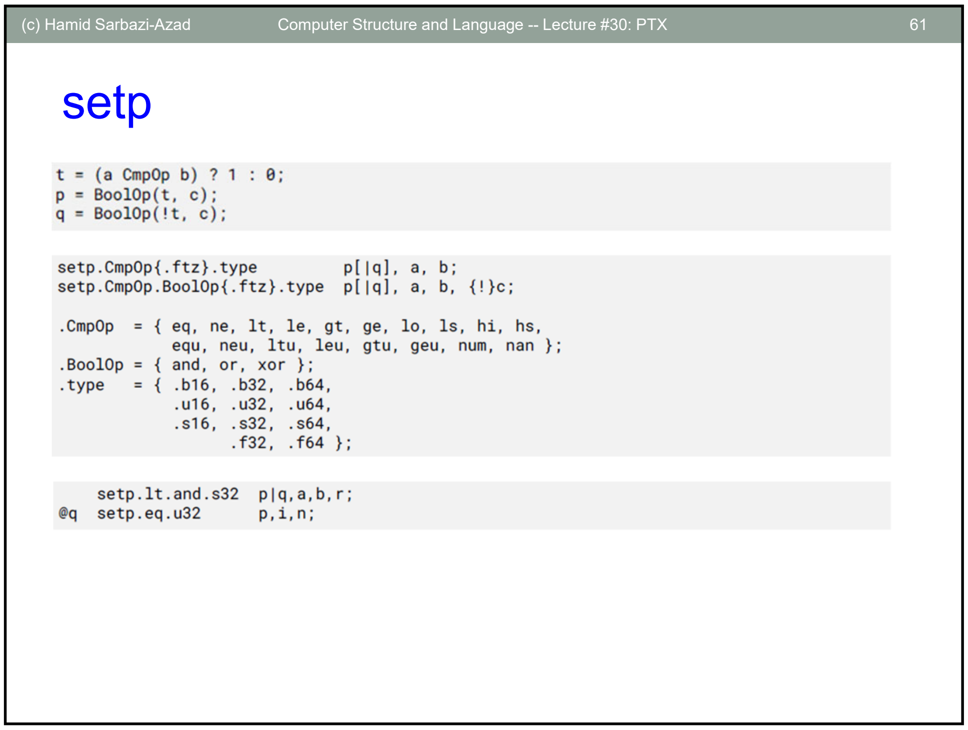
set

```
t = (a CmpOp b) ? 1 : 0;
if (isFloat(dtype))
    d = BoolOp(t, c) ? 1.0f : 0x00000000;
else
    d = BoolOp(t, c) ? 0xffffffff : 0x00000000;
```

```
set.CmpOp{.ftz}.dtype.stype         d, a, b;
set.CmpOp.BoolOp{.ftz}.dtype.stype  d, a, b, {!}c;

.CmpOp  = { eq, ne, lt, le, gt, ge, lo, ls, hi, hs,
            equ, neu, ltu, leu, gtu, geu, num, nan };
.BoolOp = { and, or, xor };
.dtype  = { .u32, .s32, .f32 };
.stype  = { .b16, .b32, .b64,
            .u16, .u32, .u64,
            .s16, .s32, .s64,
                  .f32, .f64 };
```

```
@p  set.lt.and.f32.s32  d,a,b,r;
    set.eq.u32.u32      d,i,n;
```

60



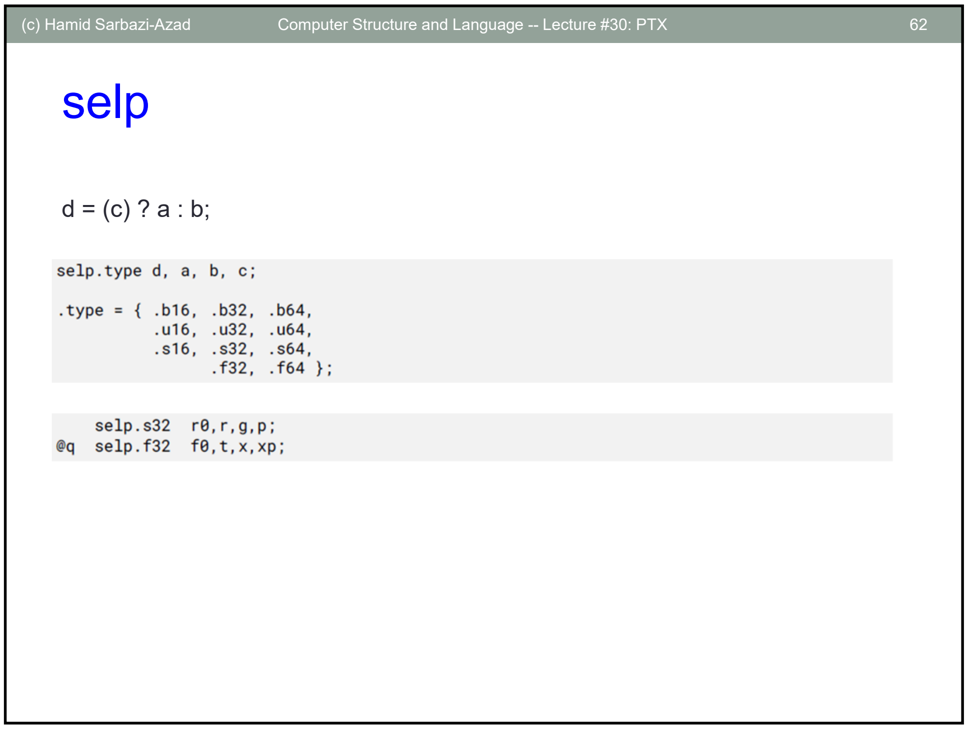
setp

```
t = (a CmpOp b) ? 1 : 0;
p = BoolOp(t, c);
q = BoolOp(!t, c);
```

```
setp.CmpOp{.ftz}.type         p[|q], a, b;
setp.CmpOp.BoolOp{.ftz}.type  p[|q], a, b, {!}c;

.CmpOp  = { eq, ne, lt, le, gt, ge, lo, ls, hi, hs,
            equ, neu, ltu, leu, gtu, geu, num, nan };
.BoolOp = { and, or, xor };
.type   = { .b16, .b32, .b64,
            .u16, .u32, .u64,
            .s16, .s32, .s64,
                  .f32, .f64 };
```

```
    setp.lt.and.s32  p|q,a,b,r;
@q  setp.eq.u32      p,i,n;
```

61



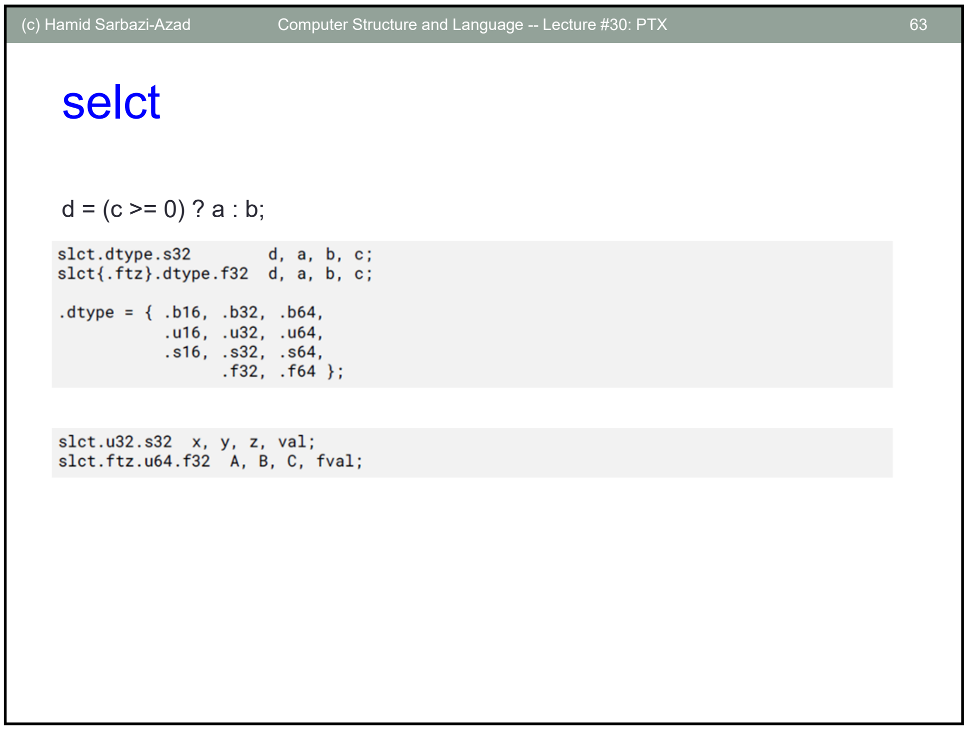
selp

d = (c) ? a : b;

```
selp.type d, a, b, c;

.type = { .b16, .b32, .b64,
          .u16, .u32, .u64,
          .s16, .s32, .s64,
                .f32, .f64 };
```

```
    selp.s32  r0,r,g,p;
@q  selp.f32  f0,t,x,xp;
```

62



selct

d = (c >= 0) ? a : b;

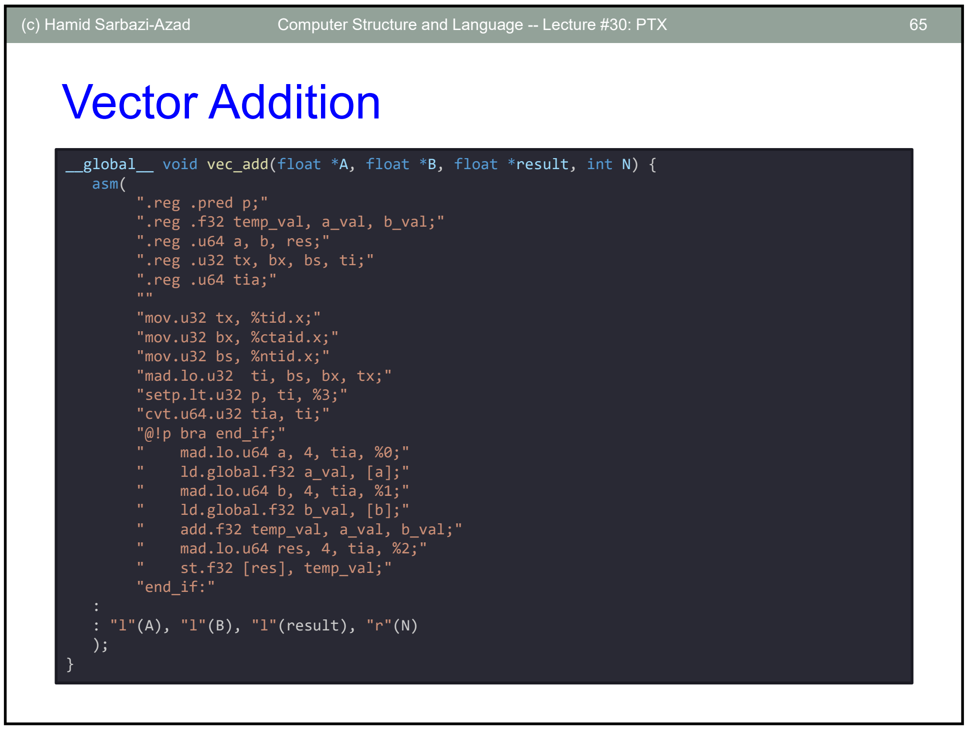
```
slct.dtype.s32        d, a, b, c;
slct{.ftz}.dtype.f32  d, a, b, c;

.dtype = { .b16, .b32, .b64,
           .u16, .u32, .u64,
           .s16, .s32, .s64,
                 .f32, .f64 };
```

```
slct.u32.s32  x, y, z, val;
slct.ftz.u64.f32  A, B, C, fval;
```

63

SAMPLE CODE

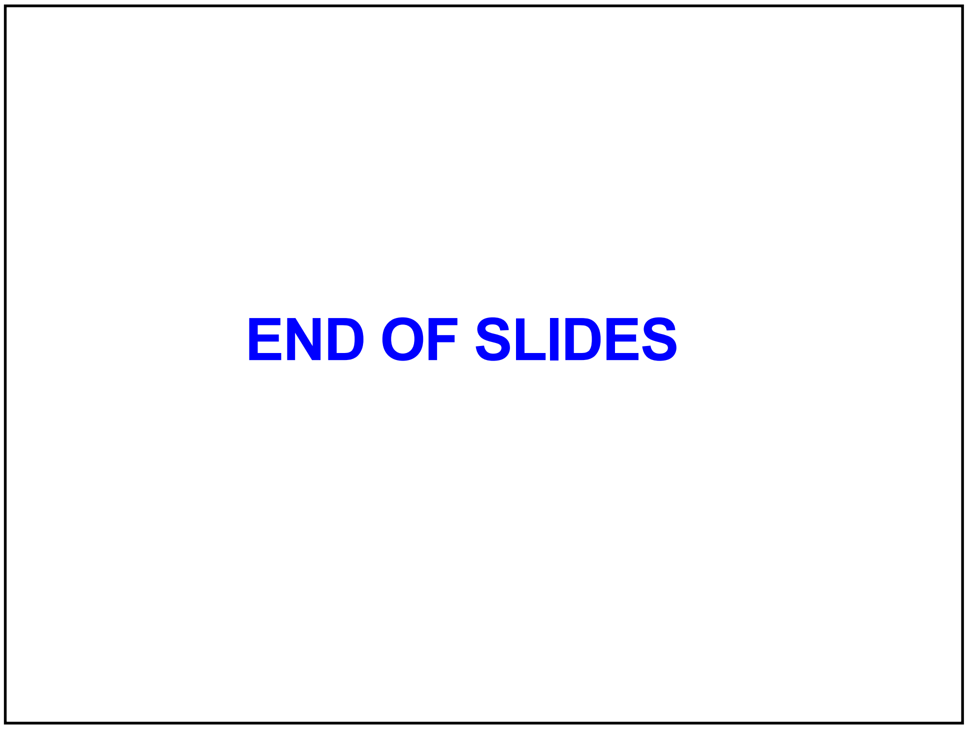
64



Vector Addition

```
__global__ void vec_add(float *A, float *B, float *result, int N) {
   asm(
        ".reg .pred p;"
        ".reg .f32 temp_val, a_val, b_val;"
        ".reg .u64 a, b, res;"
        ".reg .u32 tx, bx, bs, ti;"
        ".reg .u64 tia;"
        ""
        "mov.u32 tx, %tid.x;"
        "mov.u32 bx, %ctaid.x;"
        "mov.u32 bs, %ntid.x;"
        "mad.lo.u32  ti, bs, bx, tx;"
        "setp.lt.u32 p, ti, %3;"
        "cvt.u64.u32 tia, ti;"
        "@!p bra end_if;"
        "    mad.lo.u64 a, 4, tia, %0;"
        "    ld.global.f32 a_val, [a];"
        "    mad.lo.u64 b, 4, tia, %1;"
        "    ld.global.f32 b_val, [b];"
        "    add.f32 temp_val, a_val, b_val;"
        "    mad.lo.u64 res, 4, tia, %2;"
        "    st.f32 [res], temp_val;"
        "end_if:"
   :
   : "l"(A), "l"(B), "l"(result), "r"(N)
   );
}
```

65


END OF SLIDES

66