IBM

Queue Machines: an unknown alternative

Arquimedes Canedo
IBM Research - Tokyo

About me

- Personal
  - Born in Mexico
  - 7 years in Japan
  - I like cooking, plan to open a restaurant after retirement from research
- Professional
  - BS.Eng. Instituto Politecnico Nacional, Mexico
  - M.Eng. Dr. Eng., University of Electro-Communications, Japan, 2008
    - Queue machines / Queue Compiler
  - Now Researcher at IBM Research – Tokyo
    - Synchronous dataflow languages / System Simulation

Queues...

Transport

Mathematics

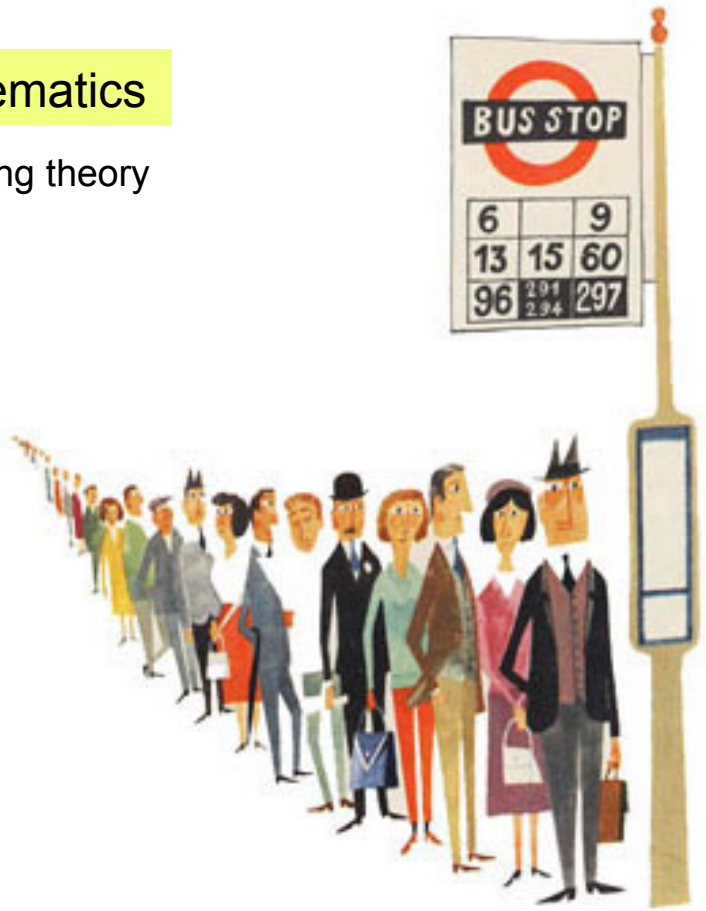
Queueing theory

Communications

Computers

Data structures
Simulation
Networks

Microprocessors
Virtual Machines



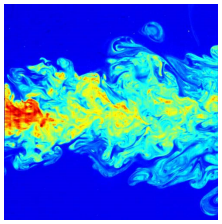
Operations research

First Come, First Served (FCFS)
First In, First Out (FIFO)

This tutorial...

- Basics of Queue Computing.
- Why queue machines matter ?
- Why should you care about queue machines ?
- Give you a new perspective.
- Encourage you to explore new ideas based on queue machines.
- How queue computing principles can be applied to other areas.

Queue Machines



- Desktop computing
- Number crunching applications
- Parallel processing
  - High instruction-level parallelism
- Streaming applications
  - The essence of queues
- Embedded devices
  - Compact program representation
- Mobile devices
  - Low power consumption
- Building blocks of many-core systems
- Software virtual machines

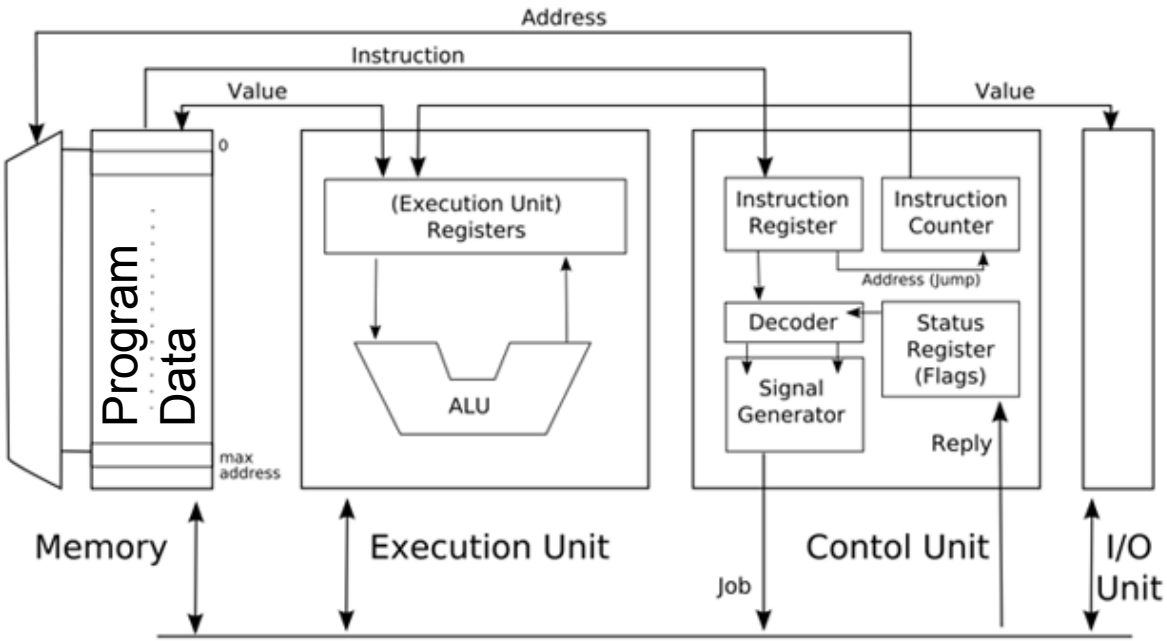
Computer Architecture Basics: Von Neumann Model

Programs

Instructions

Instruction pipeline

- Fetch
- Decode
- Execute
- Memory Access
- Write Back



Source: Department of Informatics Technische Universität München
www.lrr.in.tum.de/~jasmin/neumann.html

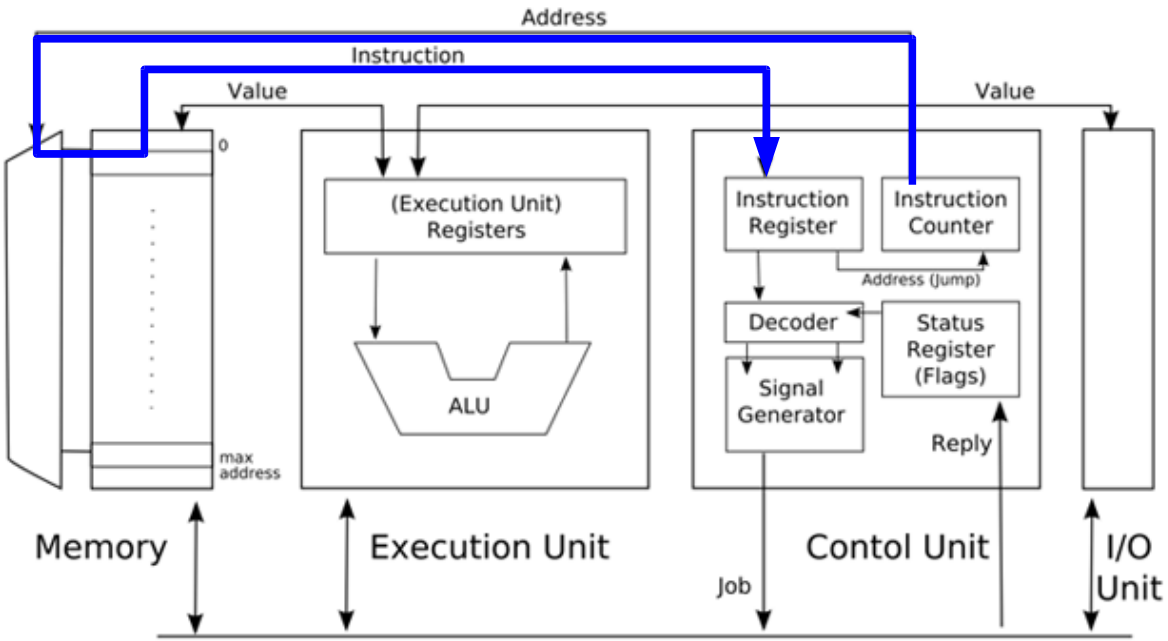
Computer Architecture Basics: Von Neumann Model

Programs

Instructions

Instruction pipeline

- Fetch
- Decode
- Execute
- Memory Access
- Write Back



Source: Department of Informatics Technische Universität München
www.lrr.in.tum.de/~jasmin/neumann.html

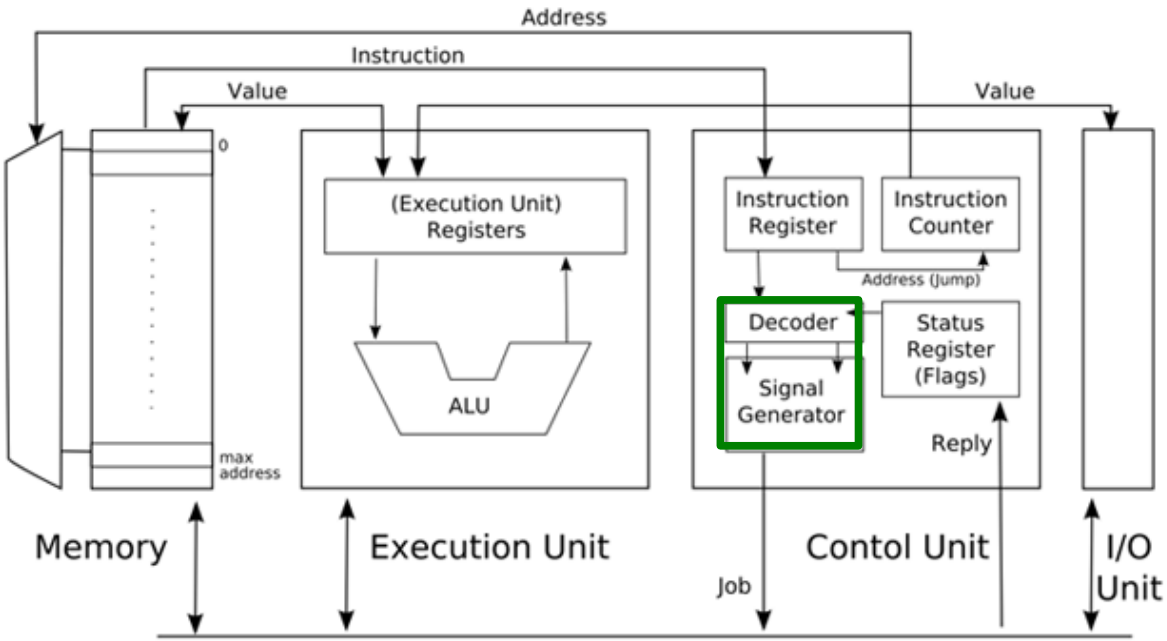
Computer Architecture Basics: Von Neumann Model

Programs

Instructions

Instruction pipeline

- Fetch
- Decode
- Execute
- Memory Access
- Write Back



Source: Department of Informatics Technische Universität München
www.lrr.in.tum.de/~jasmin/neumann.html

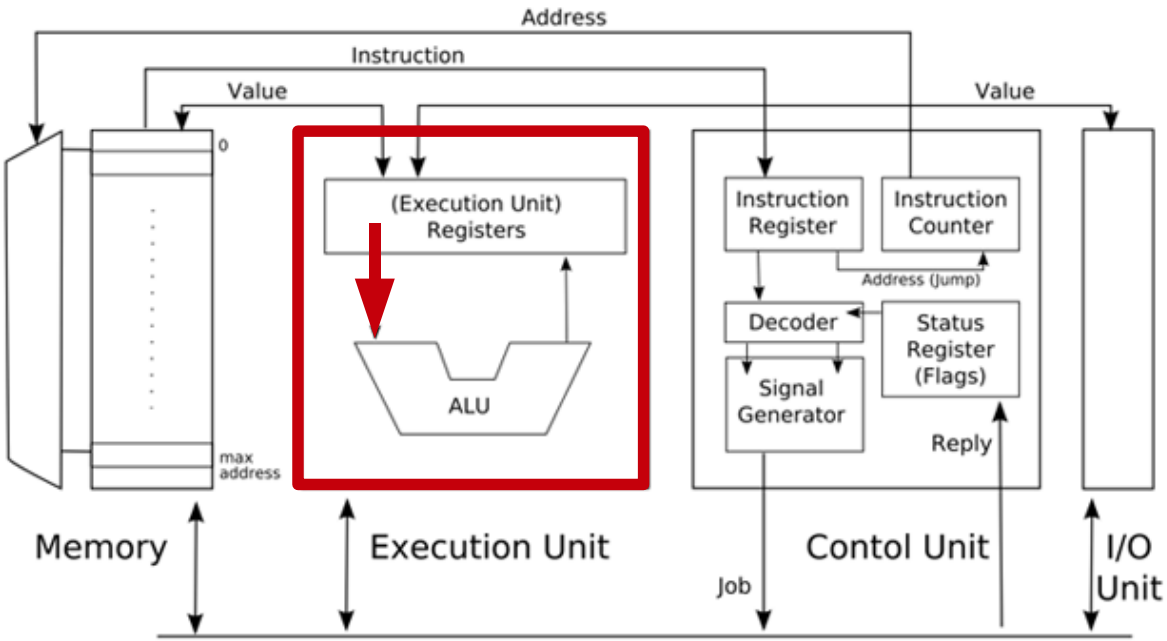
Computer Architecture Basics: Von Neumann Model

Programs

Instructions

Instruction pipeline

- Fetch
- Decode
- Execute
- Memory Access
- Write Back



Source: Department of Informatics Technische Universität München
www.lrr.in.tum.de/~jasmin/neumann.html

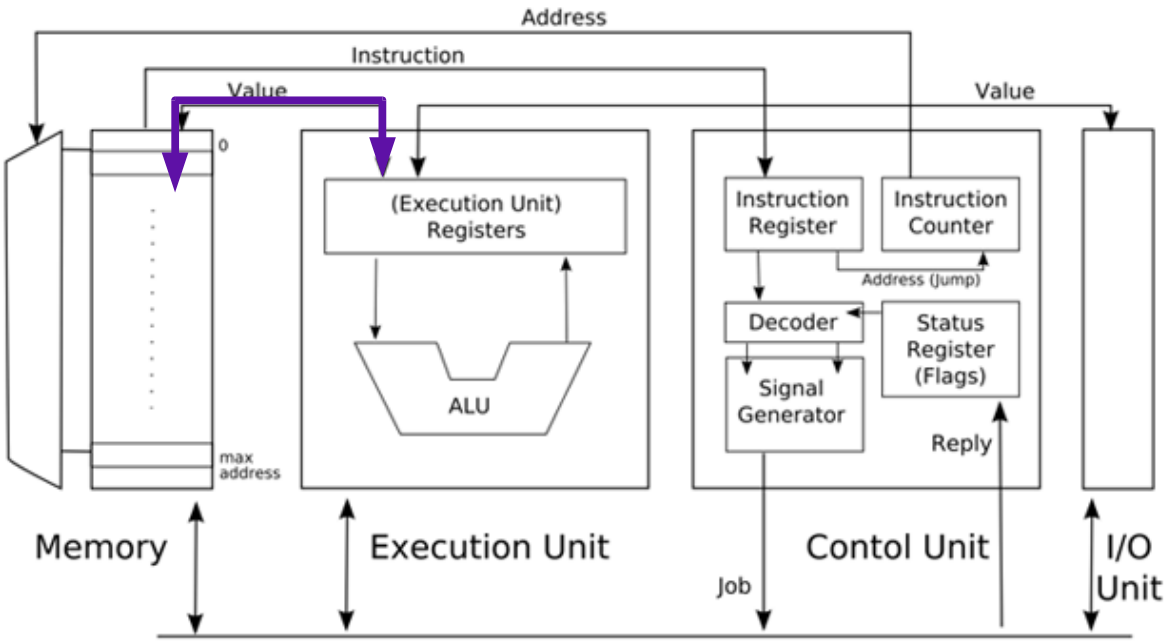
Computer Architecture Basics: Von Neumann Model

Programs

Instructions

Instruction pipeline

- Fetch
- Decode
- Execute
- Memory Access
- Write Back



Source: Department of Informatics Technische Universität München
www.lrr.in.tum.de/~jasmin/neumann.html

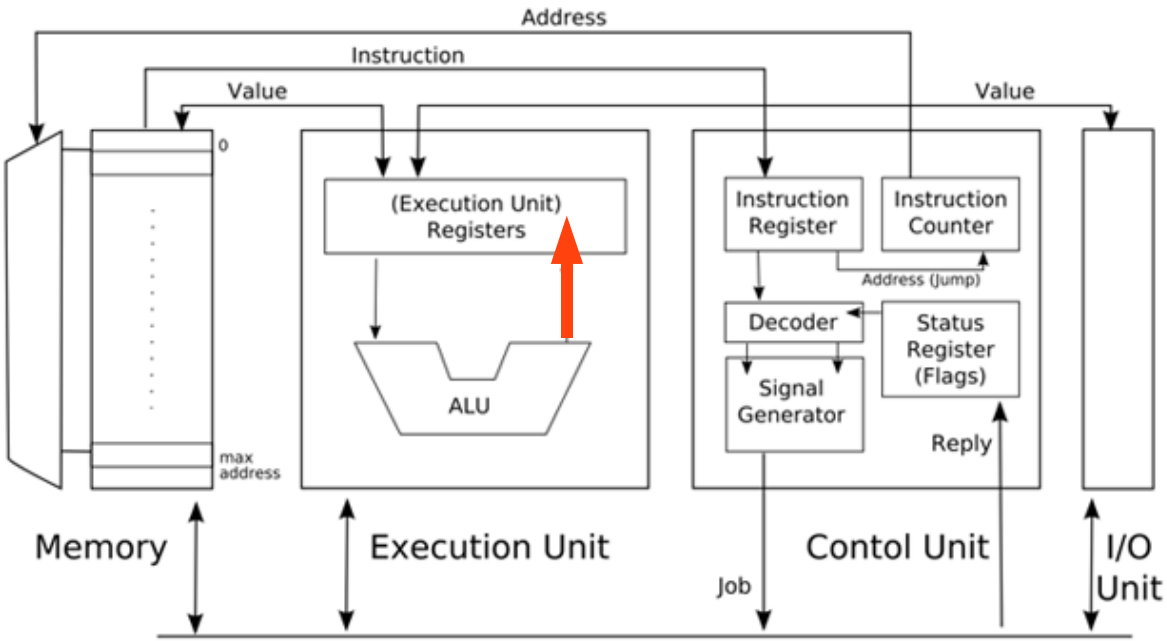
Computer Architecture Basics: Von Neumann Model

Programs

Instructions

Instruction pipeline

- Fetch
- Decode
- Execute
- Memory Access
- Write Back



Source: Department of Informatics Technische Universität München
www.lrr.in.tum.de/~jasmin/neumann.html

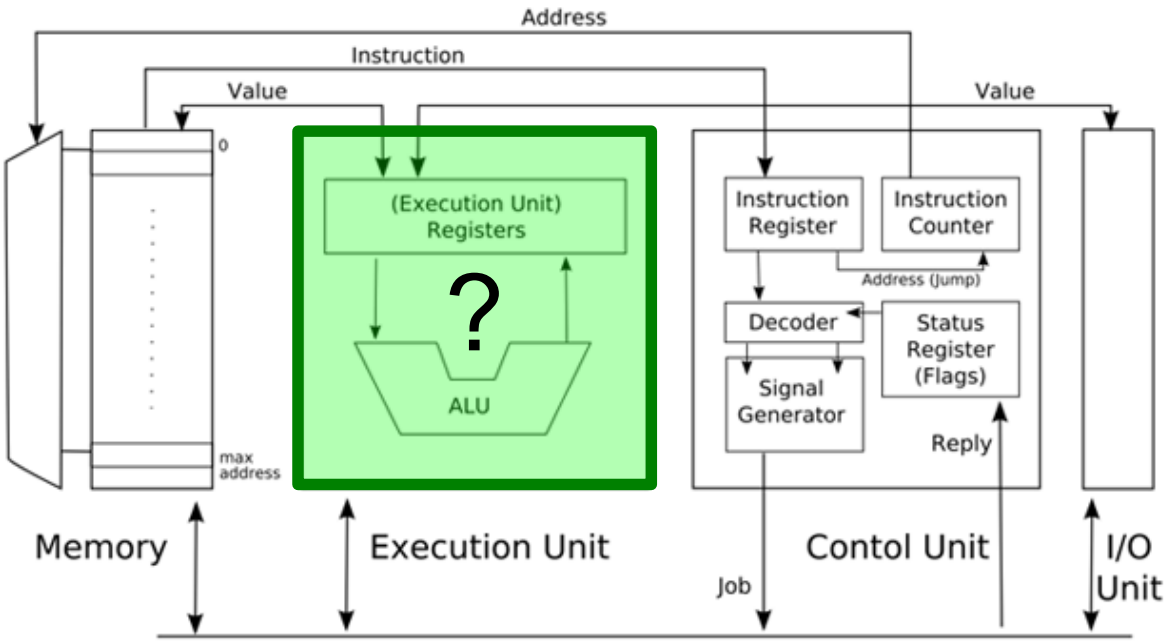
Computer Architecture Basics: Von Neumann Model

Programs

Instructions

Instruction pipeline

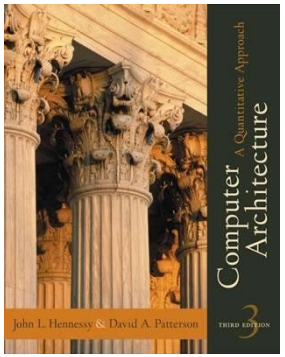
- Fetch
- Decode
- Execute
- Memory Access
- Write Back



Source: Department of Informatics Technische Universität München
www.lrr.in.tum.de/~jasmin/neumann.html

Instruction Set Architecture (ISA)

- Interface to the hardware
  - Instructions (opcode)
  - Registers
  - Data types
  - Addressing modes
  - Interrupts, etc.
- CISC vs RISC
- Variable-length vs fixed-length instructions



How important is the code density, Hw simplicity, etc?

Example of RISC instruction:

```
addu  r1, r2, r3 = 32bit
```

opcode

Destination operand

Source operands

How many instructions ? What kind ?

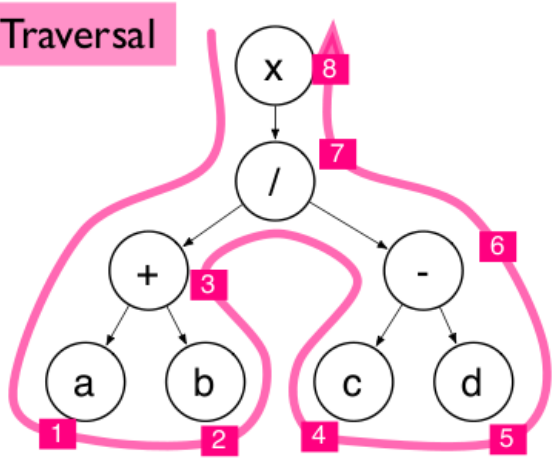
5-bit operands limit the addressable registers to 2^5=32

Register machines

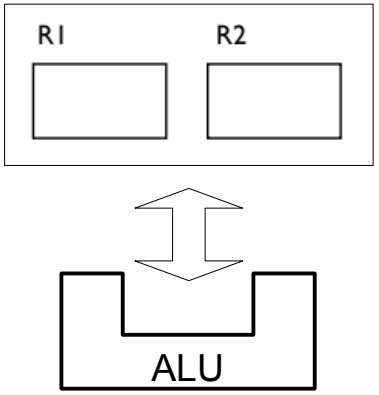
e.g.
Pentium,
SPARC,
ARM

"x = (a+b) / (c-d)"

Post-Order Traversal



Register Machines



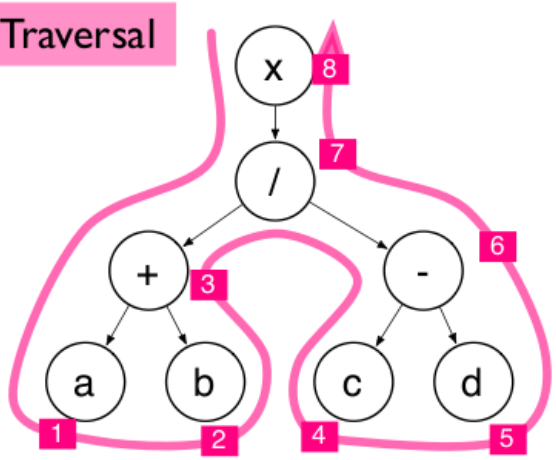

```
ld  r1, a
ld  r2, b
add r1, r1, r2

ld  r2, c
ld  r1, d
sub r2, r2, r1

div r1, r1, r2
st  r1, x
```

Register machines

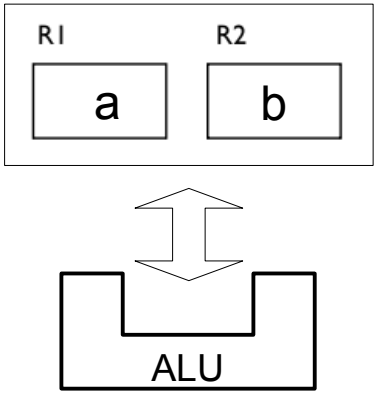
e.g.
Pentium,
SPARC,
ARM

"x = (a+b) / (c-d)"

Post-Order Traversal



Register Machines



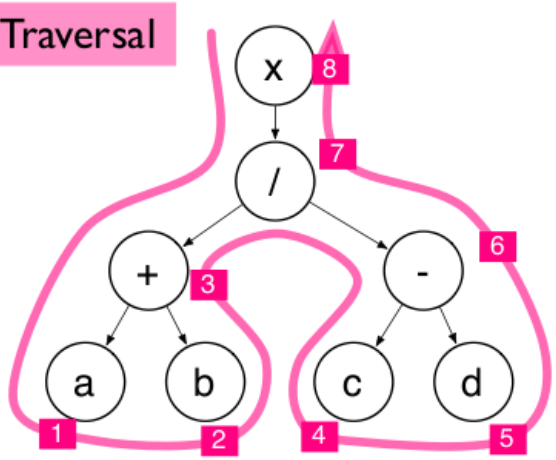

```
ld  r1, a
ld  r2, b
add r1, r1, r2

ld  r2, c
ld  r1, d
sub r2, r2, r1

div r1, r1, r2
st  r1, x
```

Register machines

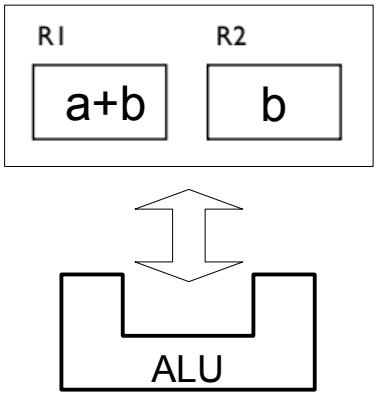
e.g.
Pentium,
SPARC,
ARM

"x = (a+b) / (c-d)"

Post-Order Traversal



Register Machines



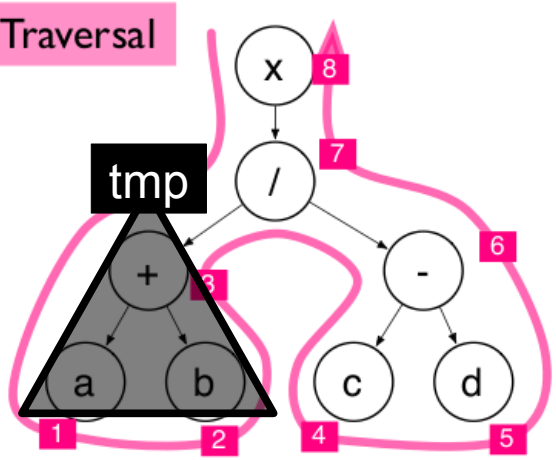

```
ld  r1, a
ld  r2, b
add r1, r1, r2

ld  r2, c
ld  r1, d
sub r2, r2, r1

div r1, r1, r2
st  r1, x
```

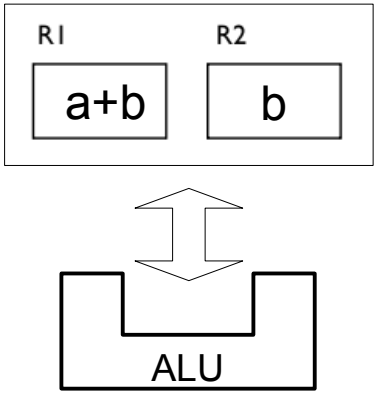
Register machines

e.g.
Pentium,
SPARC,
ARM

"x = (a+b) / (c-d)"

Post-Order Traversal





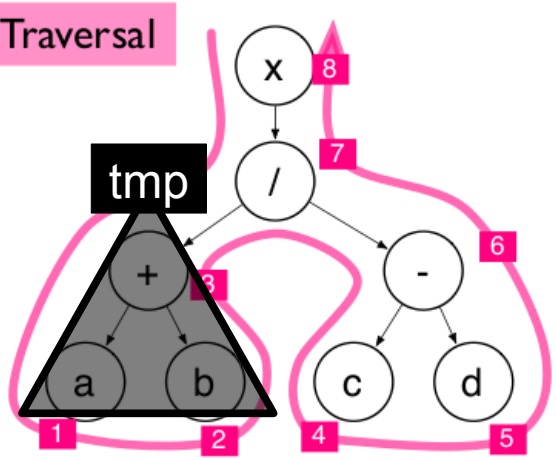

```
ld  r1, a
ld  r2, b
add r1, r1, r2
st  r1, tmp
ld  r2, c
ld  r1, d
sub r2, r2, r1

div r1, r1, r2
st  r1, x
```

Register machines

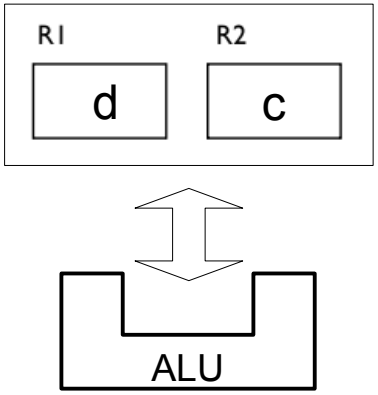
e.g.
Pentium,
SPARC,
ARM

"x = (a+b) / (c-d)"

Post-Order Traversal



Register Machines



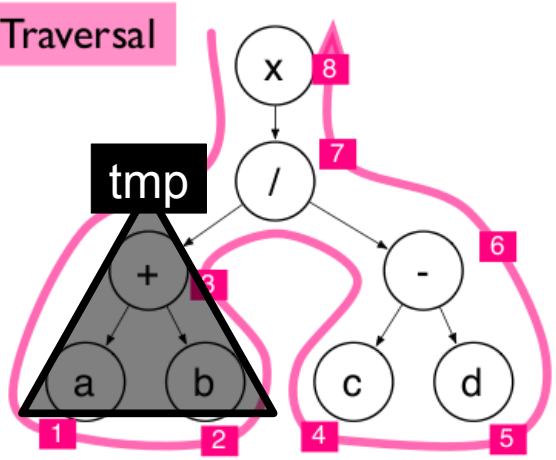

```
ld  r1, a
ld  r2, b
add r1, r1, r2
st  r1, tmp
ld  r2, c
ld  r1, d
sub r2, r2, r1

div r1, r1, r2
st  r1, x
```

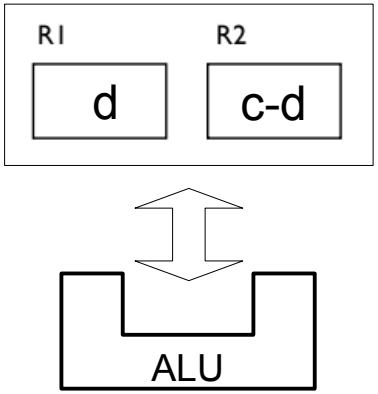
Register machines

e.g.
Pentium,
SPARC,
ARM

"x = (a+b) / (c-d)"

Post-Order Traversal





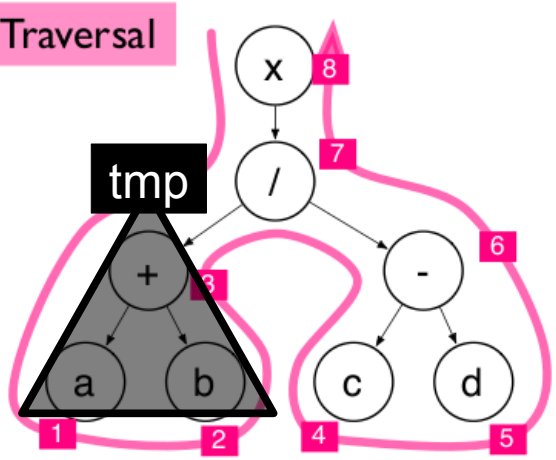

```
ld  r1, a
ld  r2, b
add r1, r1, r2
st  r1, tmp
ld  r2, c
ld  r1, d
sub r2, r2, r1

div r1, r1, r2
st  r1, x
```

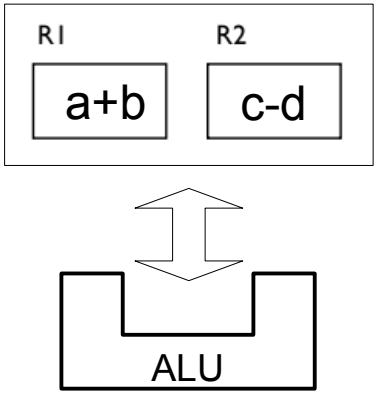
Register machines

e.g.
Pentium,
SPARC,
ARM

"x = (a+b) / (c-d)"

Post-Order Traversal





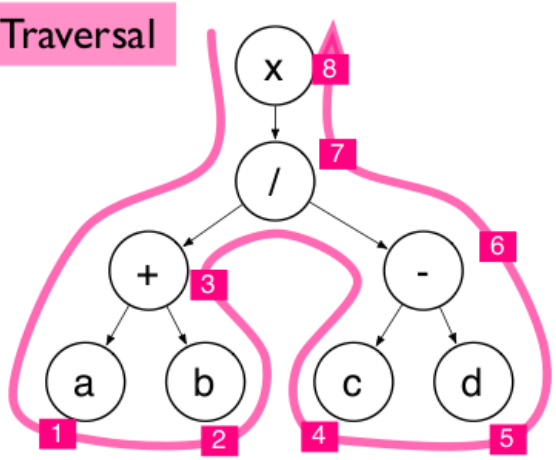

```
ld  r1, a
ld  r2, b
add r1, r1, r2
st  r1, tmp
ld  r2, c
ld  r1, d
sub r2, r2, r1
ld  r1, tmp
div r1, r1, r2
st  r1, x
```

Register machines

e.g.
Pentium,
SPARC,
ARM

"x = (a+b) / (c-d)"

Post-Order Traversal



Register Machines

Renaming Registers

RX RY

Architectural Registers

R1 R2

(a+b)/ (c-d)

c-d

ALU

```
ld   r1, a
ld   r2, b
add  r1, r1, r2
st   r1, tmp
ld   r2, c
ld   r1, d
sub  r2, r2, r1
ld   r1, tmp
div  r1, r1, r2
st   r1, x
```

False dependencies

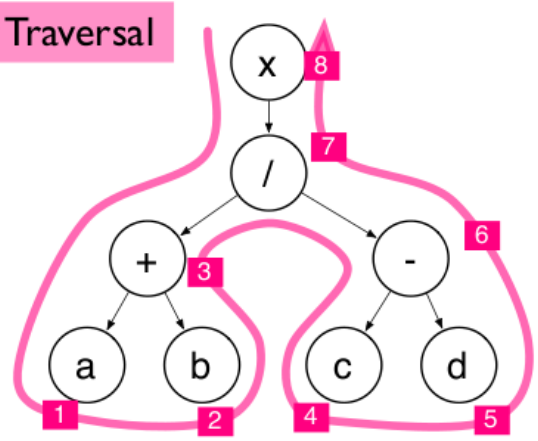
Register renaming (OOO)

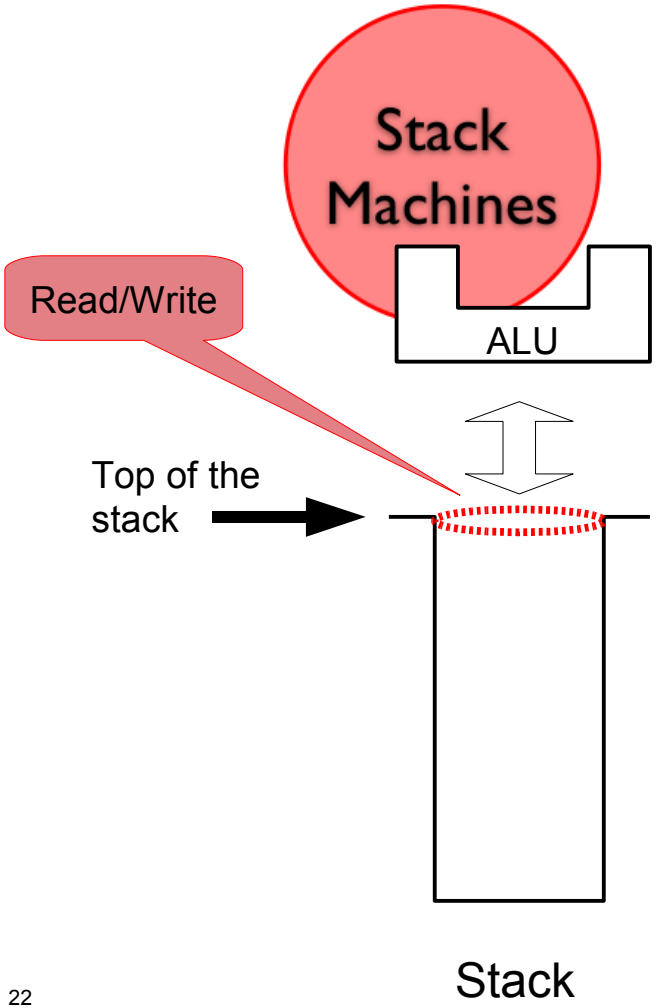
Stack machines

"x = (a+b) / (c-d)"

e.g. Forth, Java

Post-Order Traversal





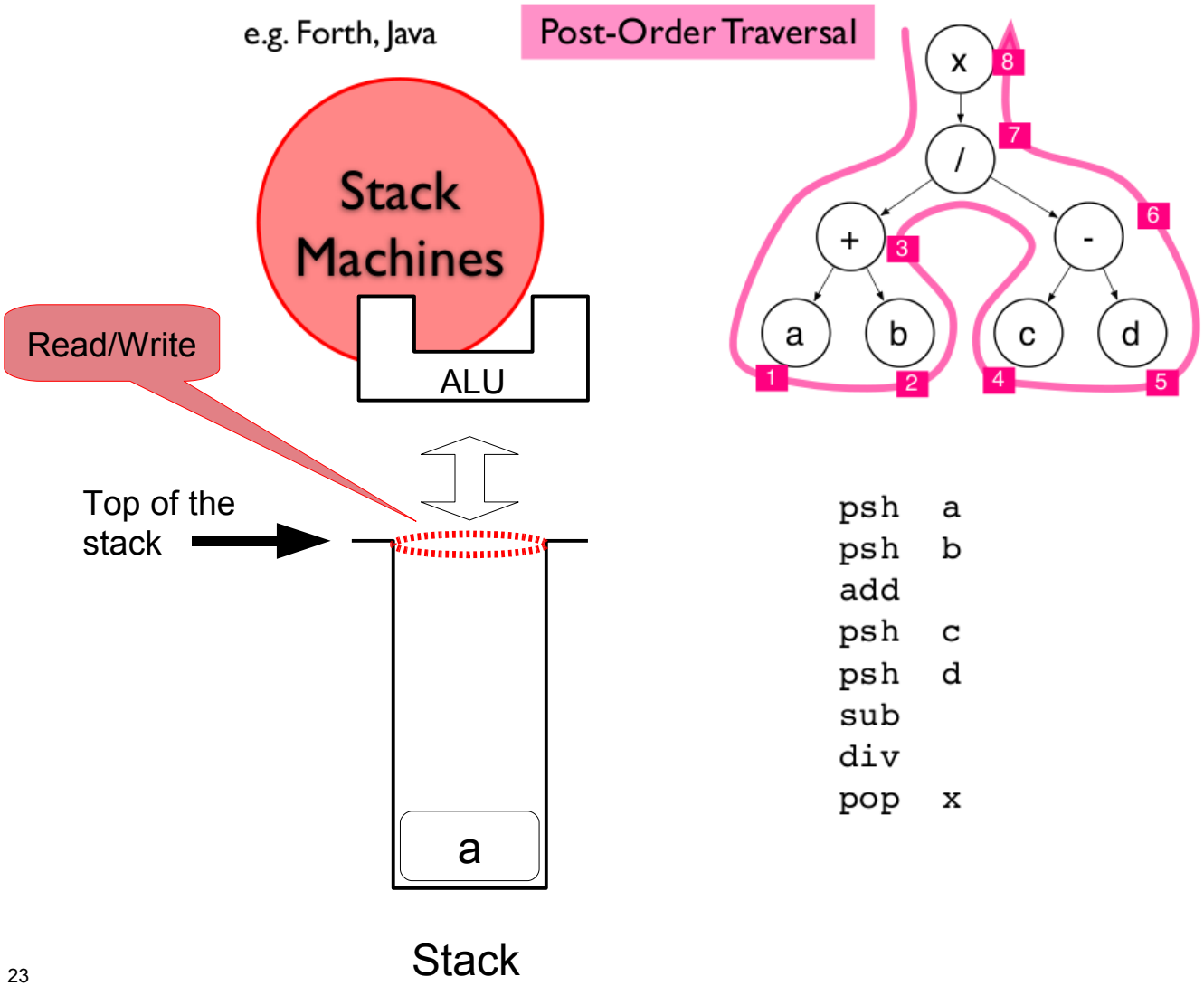

```
psh  a
psh  b
add
psh  c
psh  d
sub
div
pop  x
```

Stack machines

"x = (a+b) / (c-d)"

e.g. Forth, Java

Post-Order Traversal



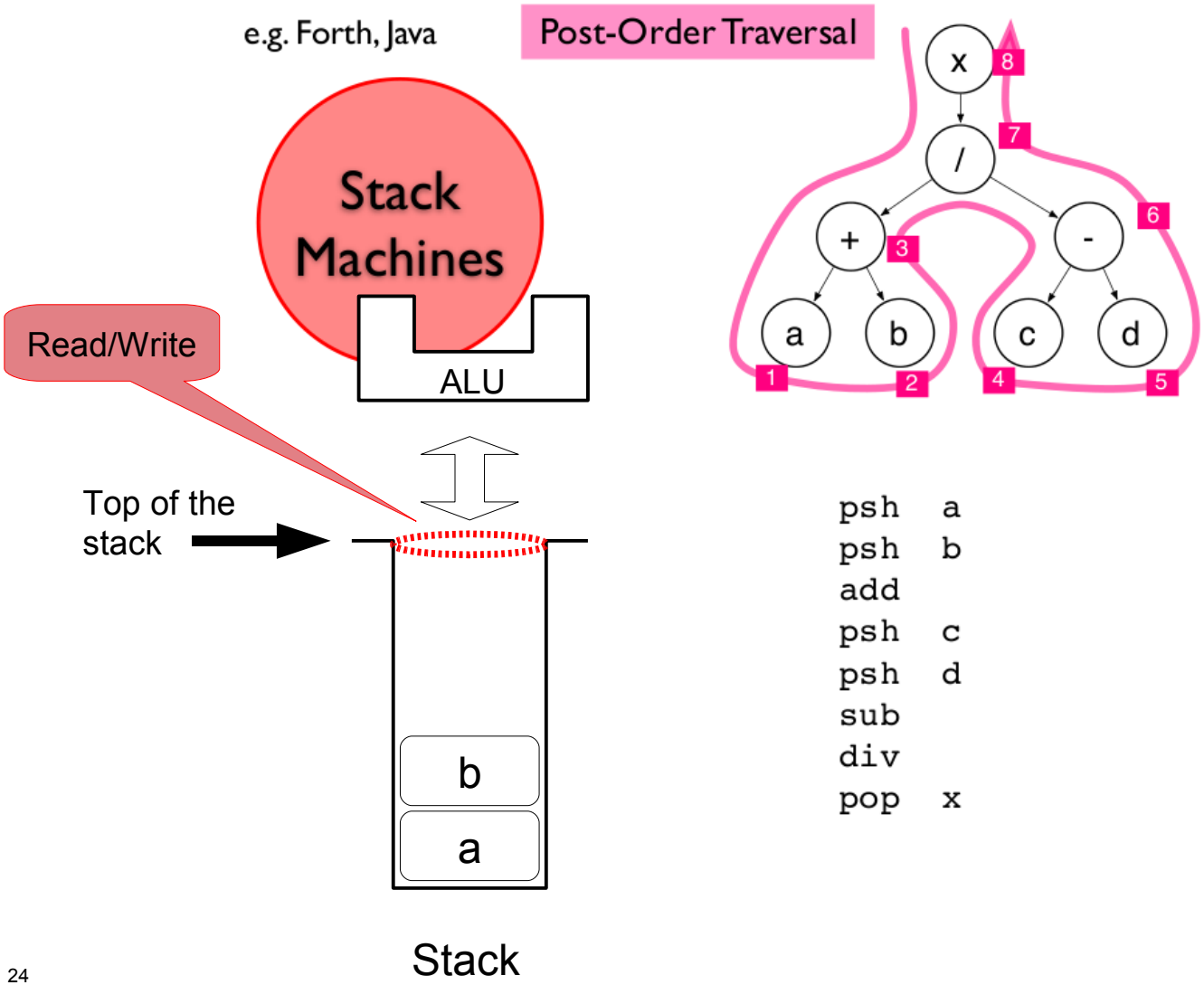

```
psh  a
psh  b
add
psh  c
psh  d
sub
div
pop  x
```

Stack machines

"x = (a+b) / (c-d)"

e.g. Forth, Java

Post-Order Traversal



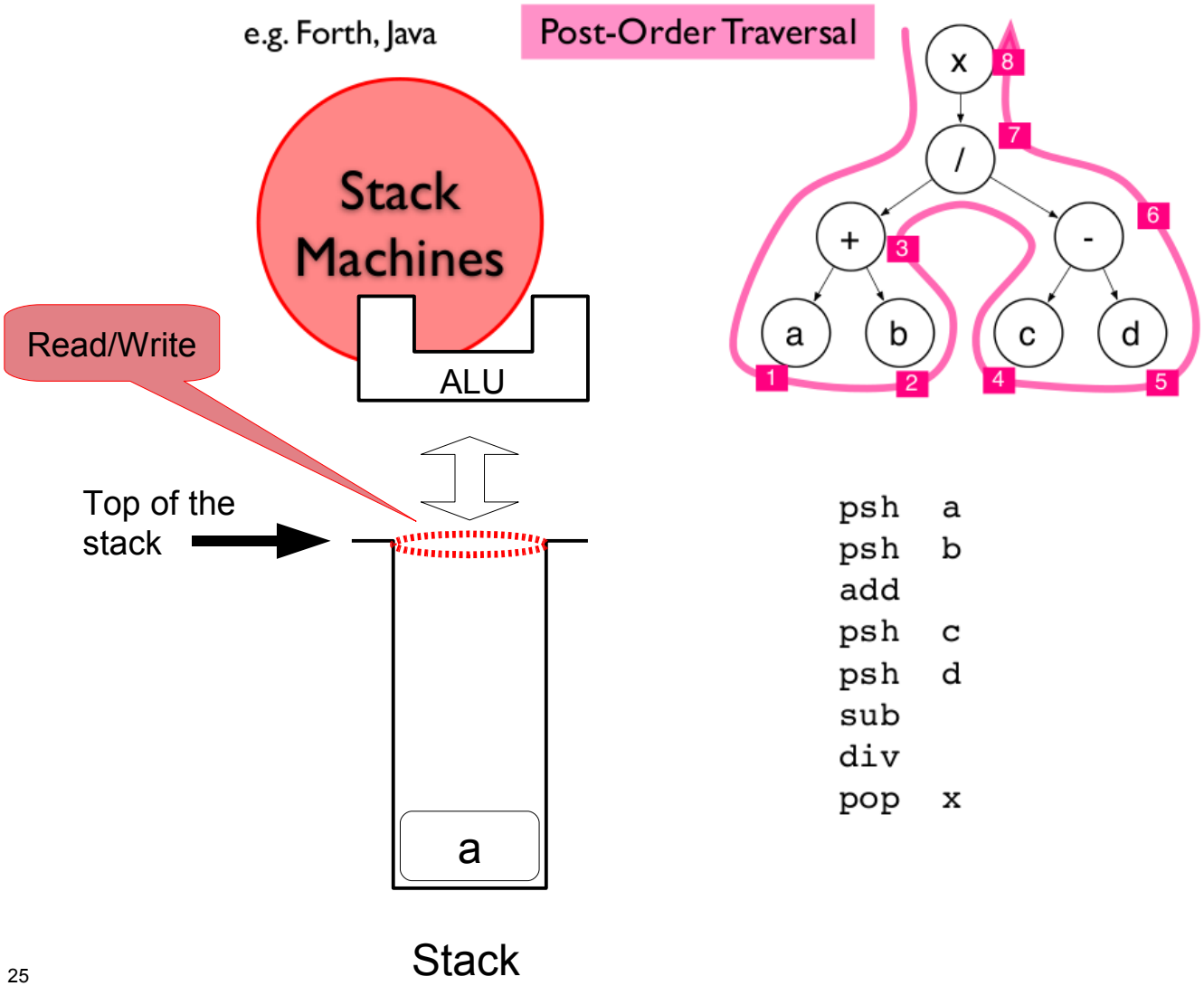

```
psh  a
psh  b
add
psh  c
psh  d
sub
div
pop  x
```

Stack machines

"x = (a+b) / (c-d)"

e.g. Forth, Java

Post-Order Traversal



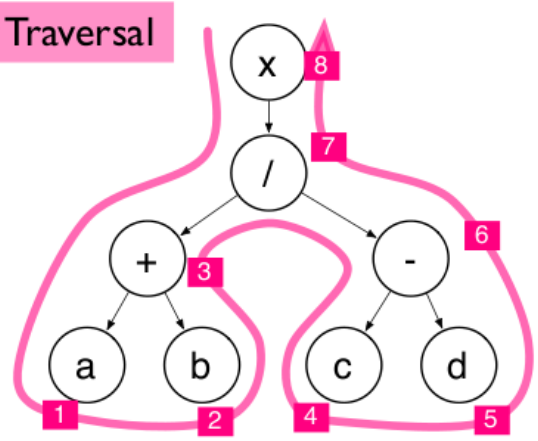

```
psh  a
psh  b
add
psh  c
psh  d
sub
div
pop  x
```

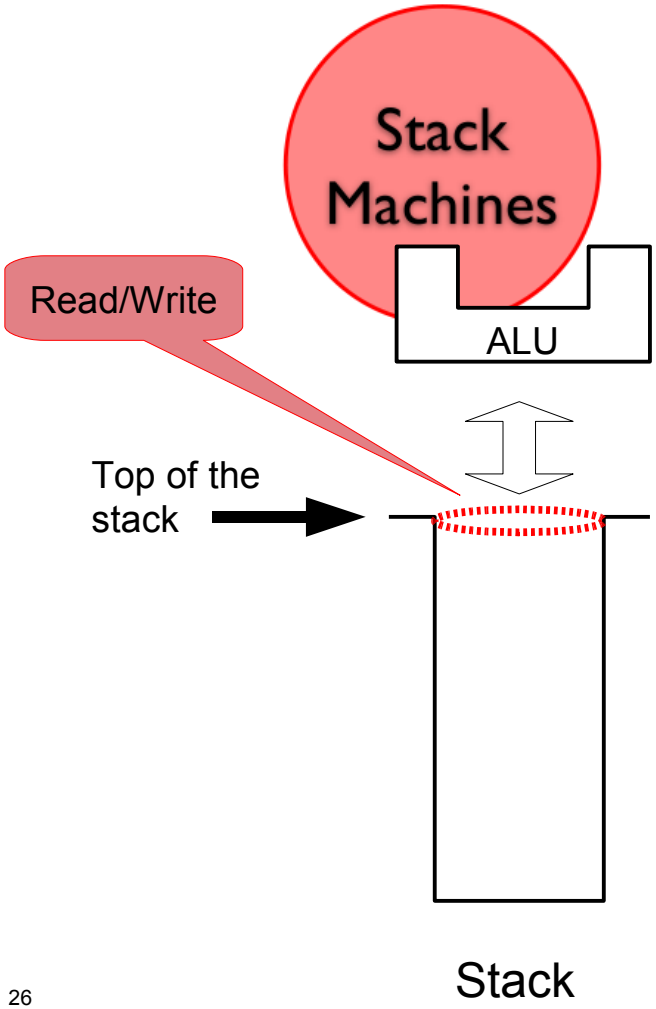
Stack machines

"x = (a+b) / (c-d)"

e.g. Forth, Java

Post-Order Traversal





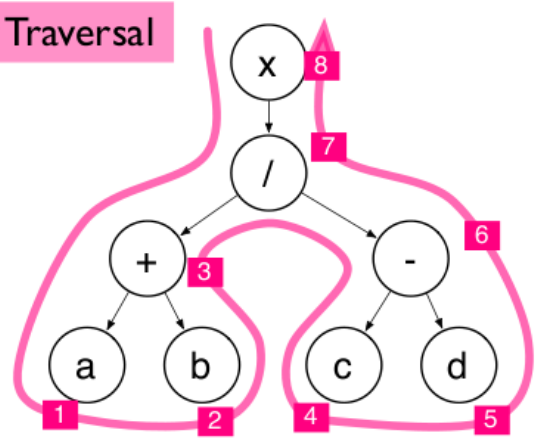

```
psh  a
psh  b
add
psh  c
psh  d
sub
div
pop  x
```

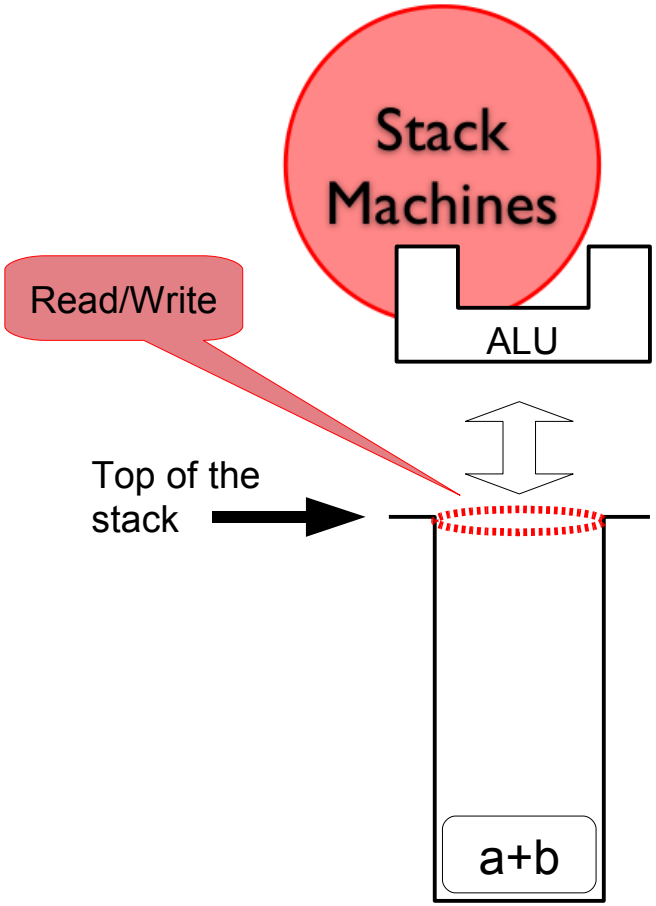
Stack machines

"x = (a+b) / (c-d)"

e.g. Forth, Java

Post-Order Traversal





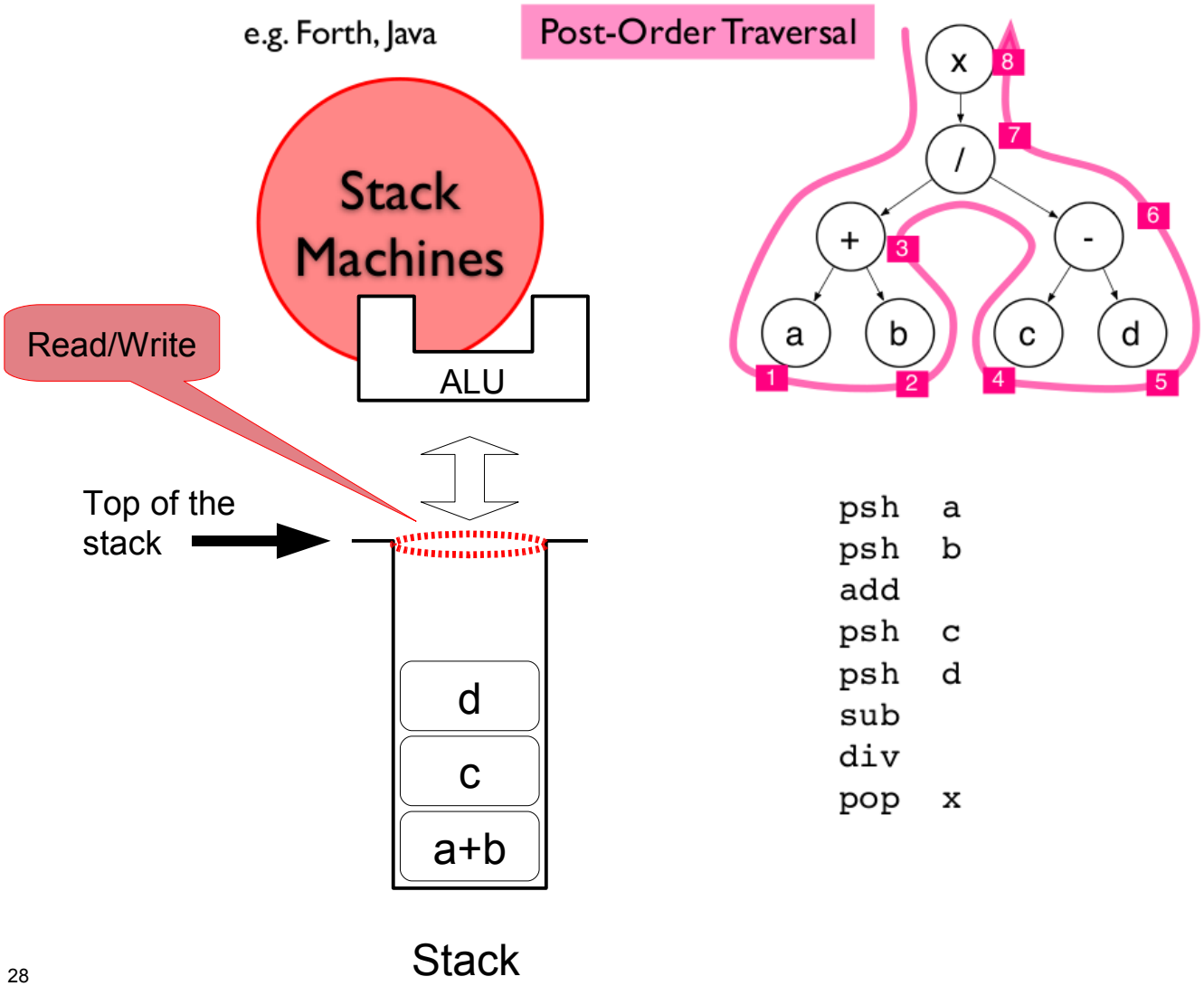

```
psh  a
psh  b
add
psh  c
psh  d
sub
div
pop  x
```

Stack machines

"x = (a+b) / (c-d)"

e.g. Forth, Java

Post-Order Traversal



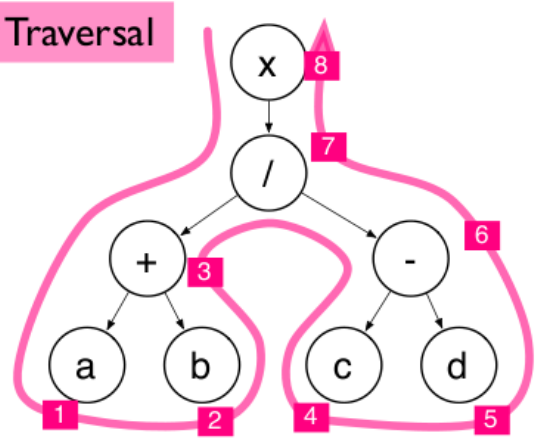

```
psh  a
psh  b
add
psh  c
psh  d
sub
div
pop  x
```

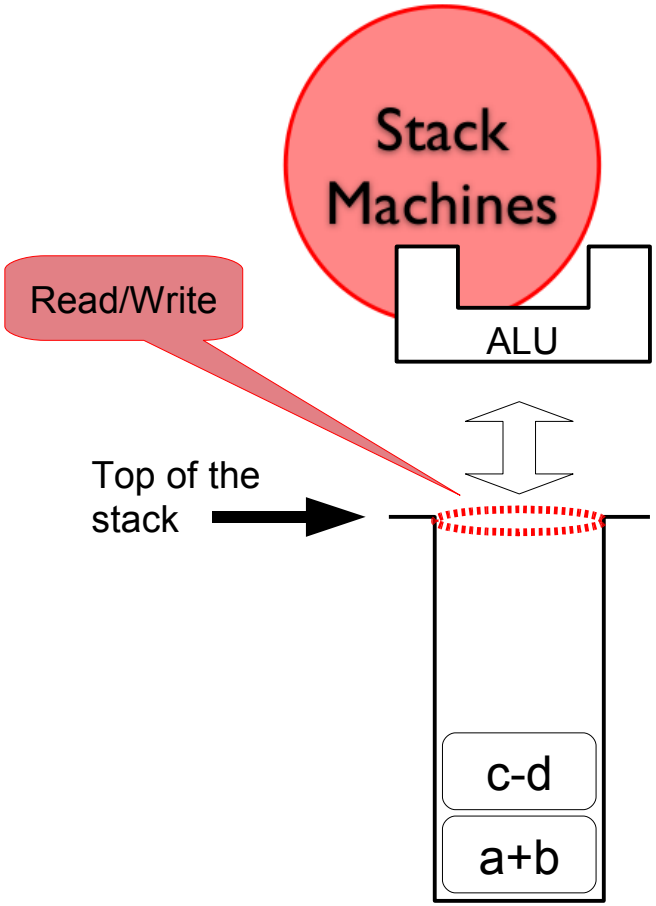
Stack machines

"x = (a+b) / (c-d)"

e.g. Forth, Java

Post-Order Traversal





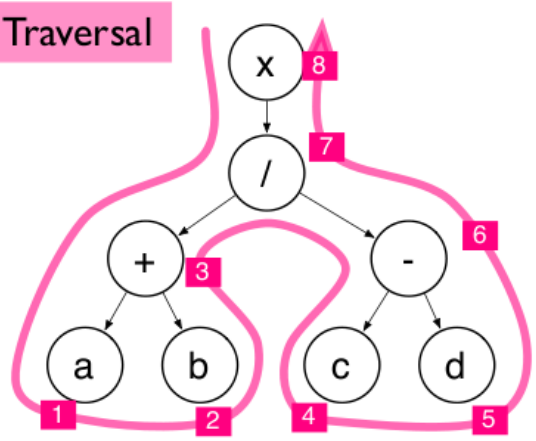

```
psh  a
psh  b
add
psh  c
psh  d
sub
div
pop  x
```

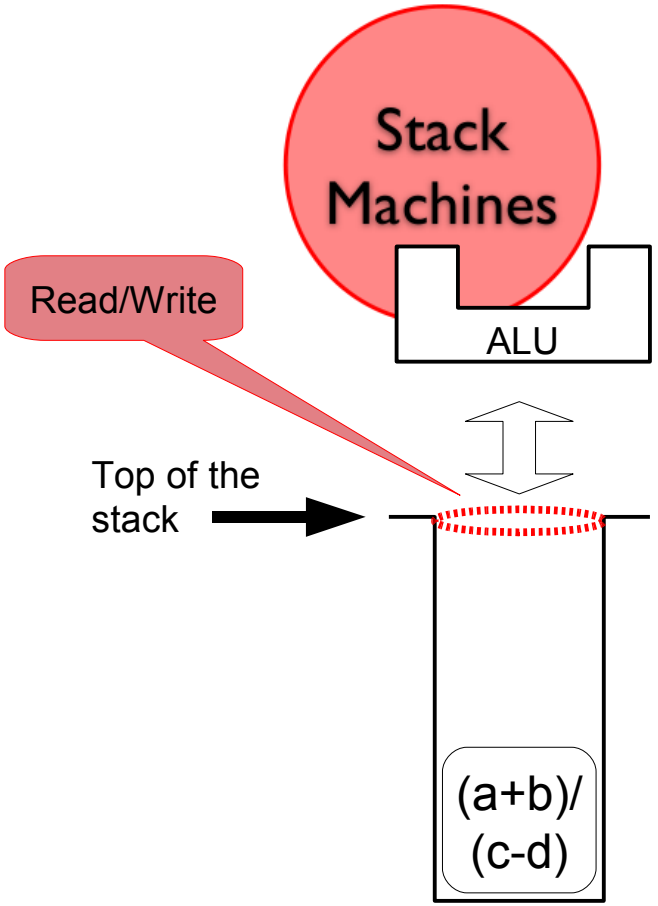
Stack machines

"x = (a+b) / (c-d)"

e.g. Forth, Java

Post-Order Traversal



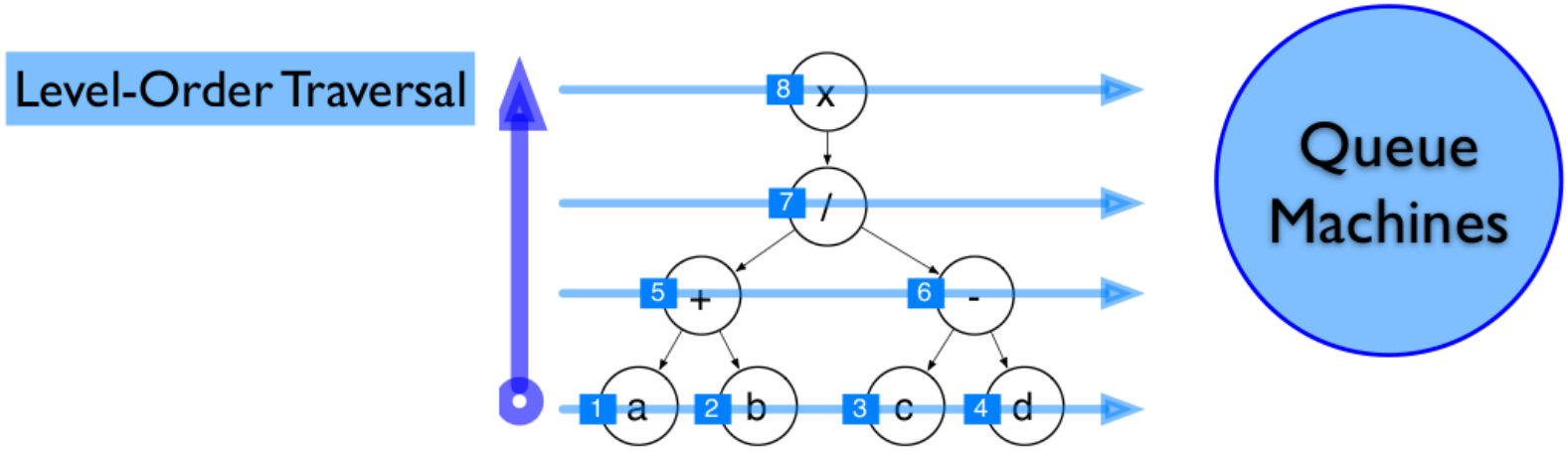


Minimal ISA

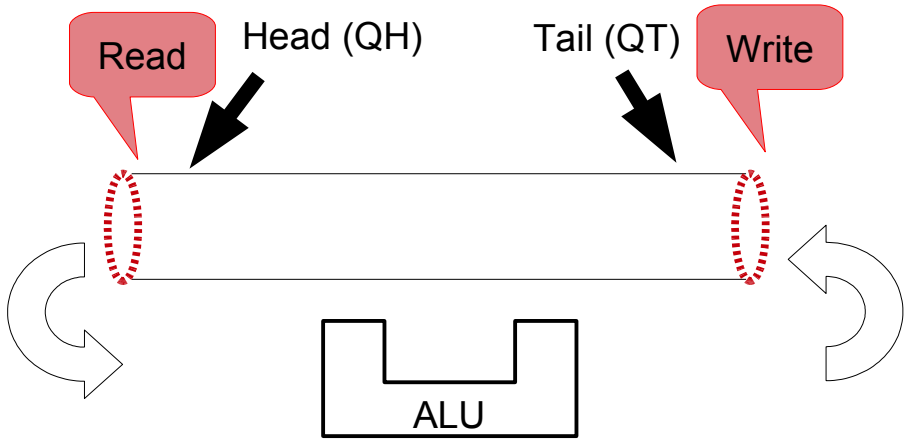
```
psh  a
psh  b
add
psh  c
psh  d
sub
div
pop  x
```

Queue machines

"x = (a+b) / (c-d)"

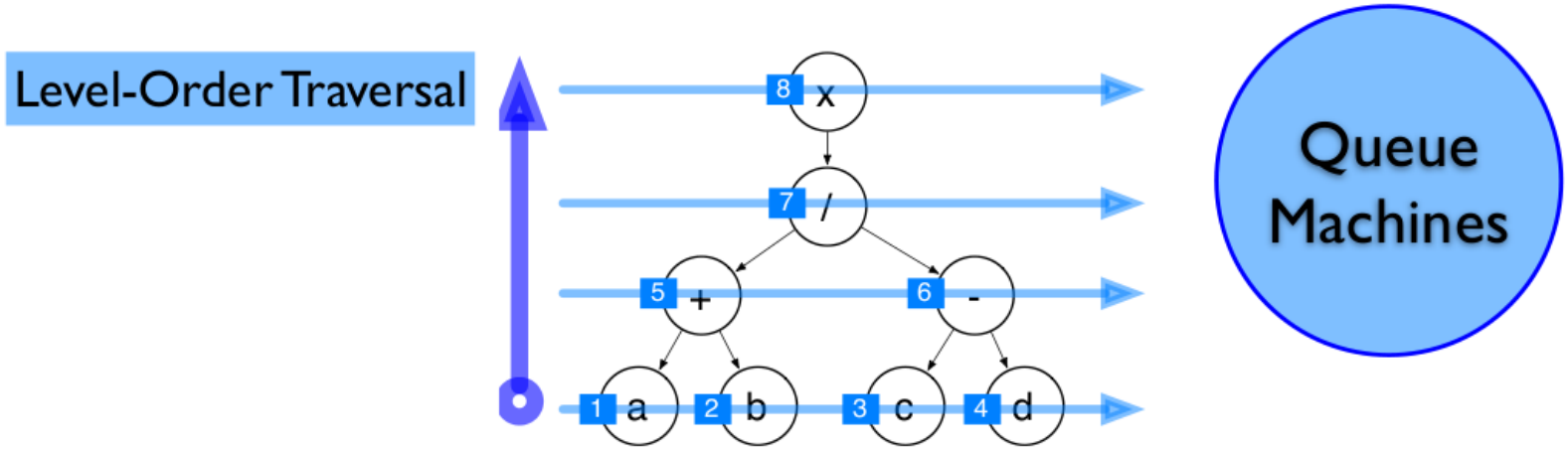



```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

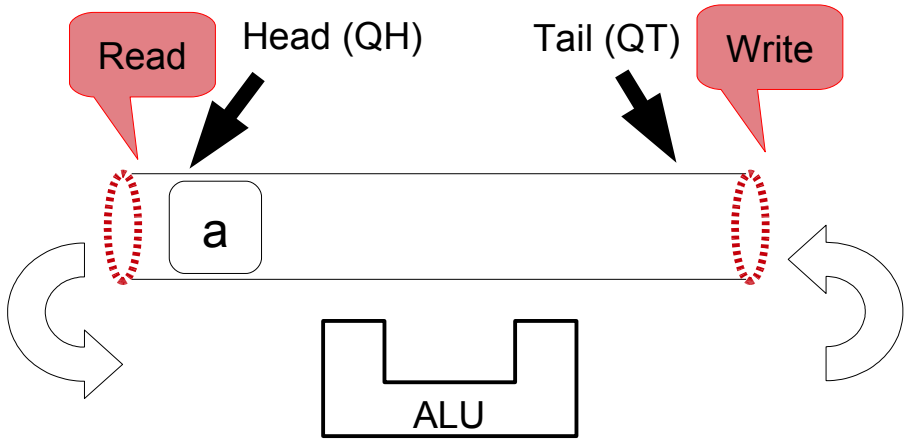


Queue machines

"x = (a+b) / (c-d)"

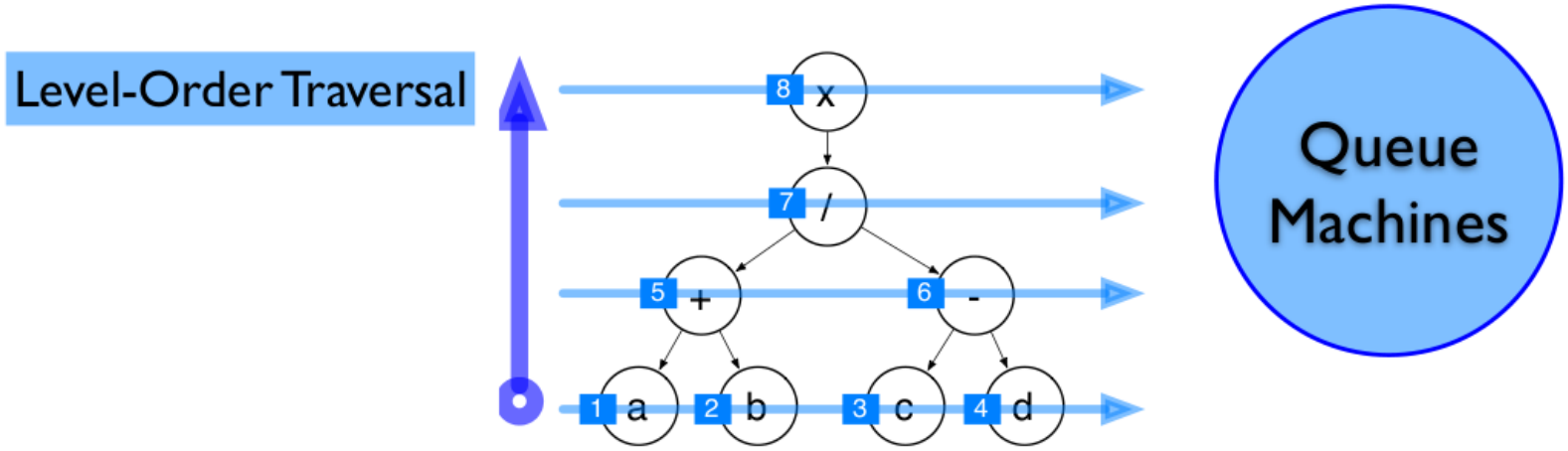



```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

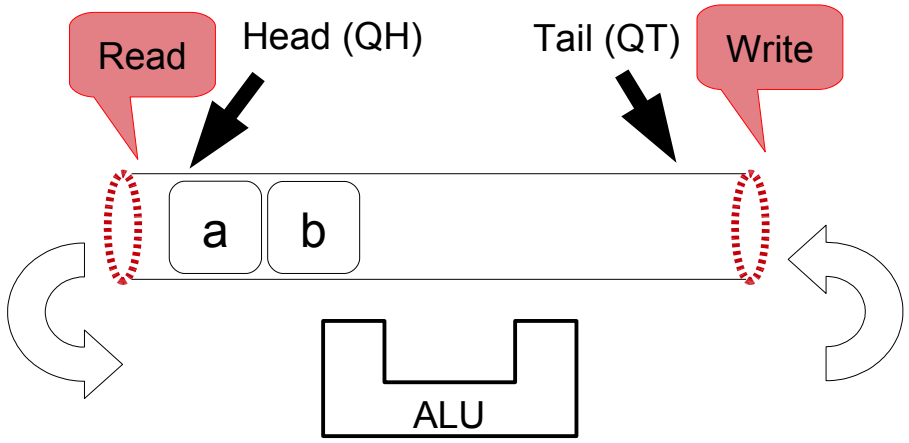


Queue machines

"x = (a+b) / (c-d)"

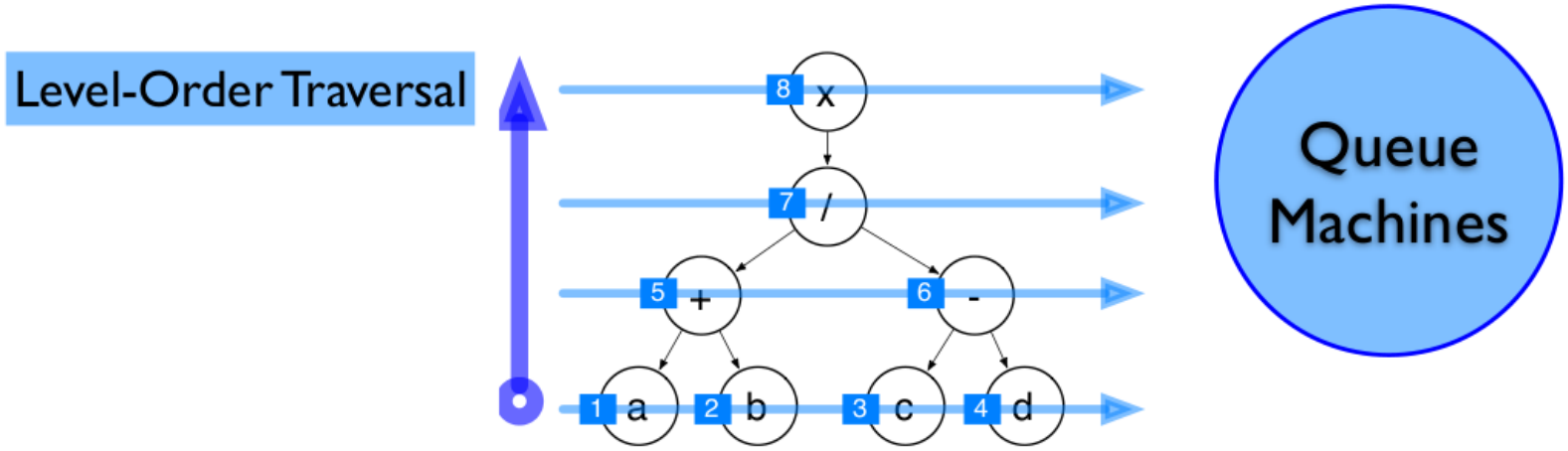



```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

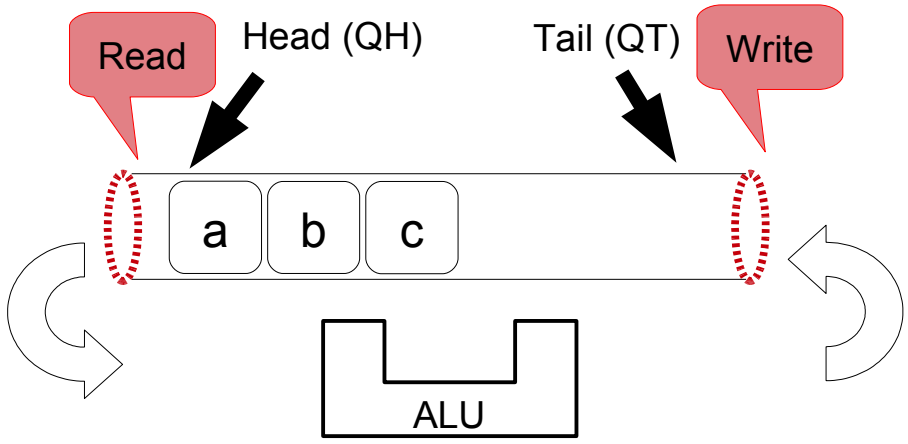


Queue machines

"x = (a+b) / (c-d)"

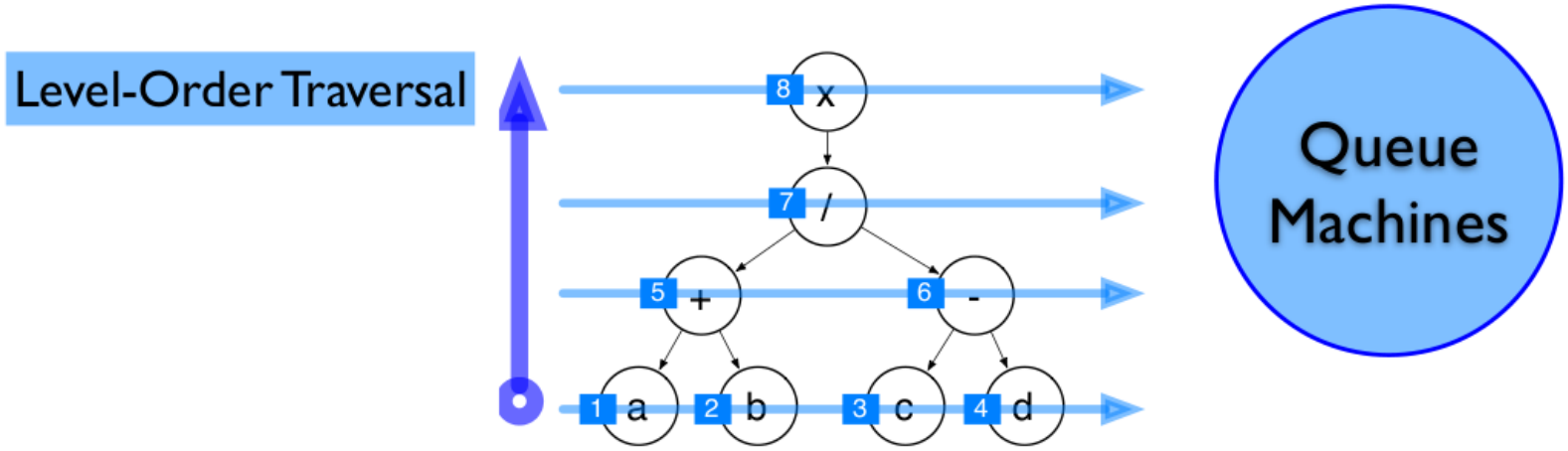



```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

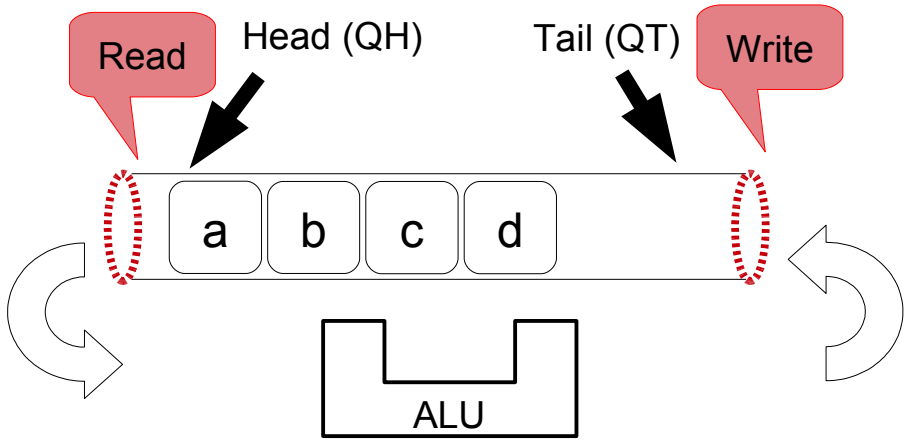


Queue machines

"x = (a+b) / (c-d)"

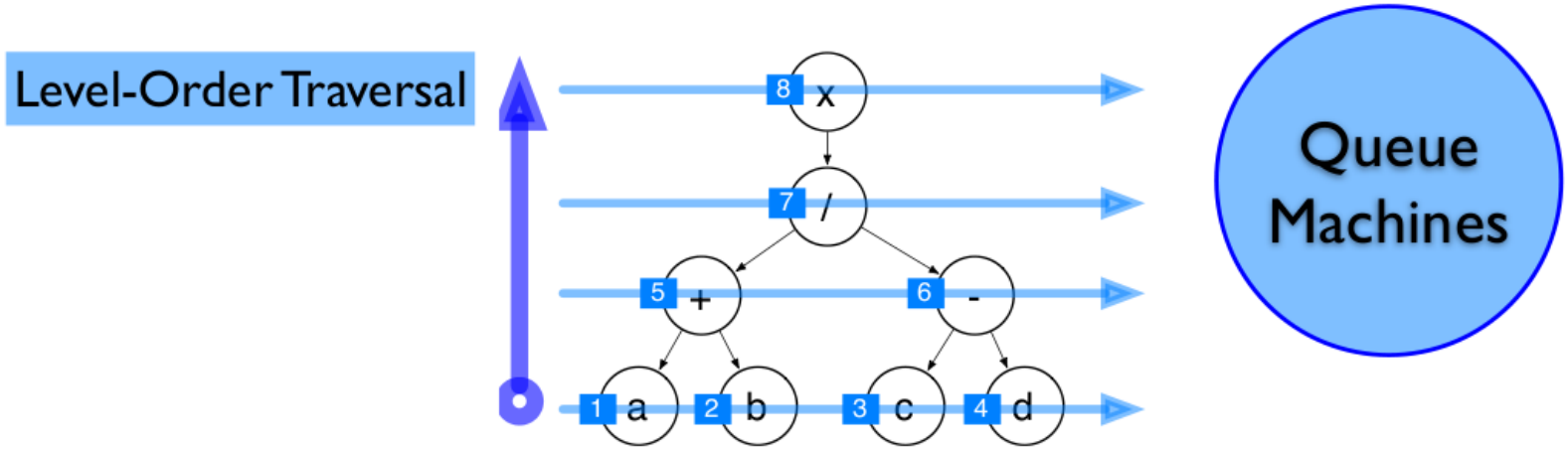



```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

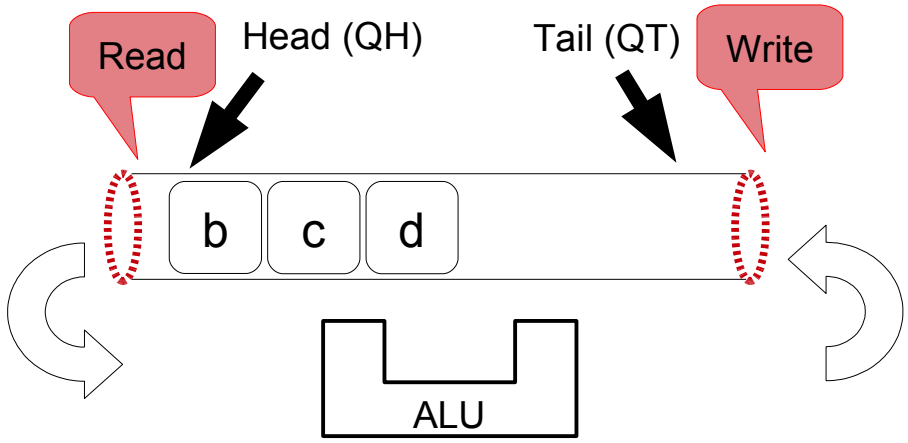


Queue machines

"x = (a+b) / (c-d)"

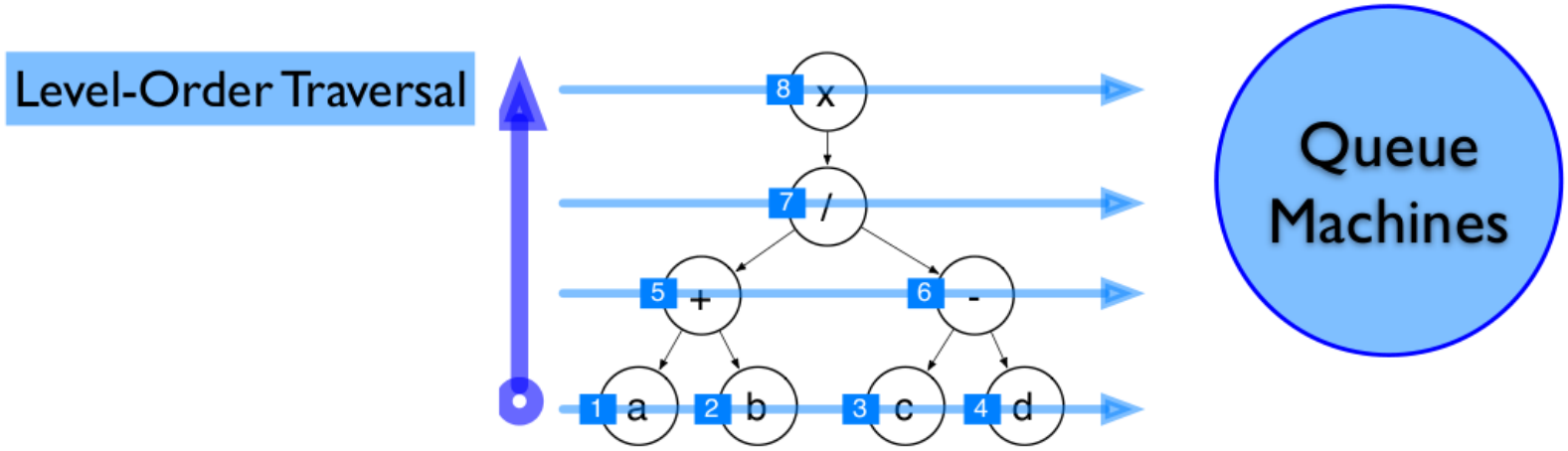



```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

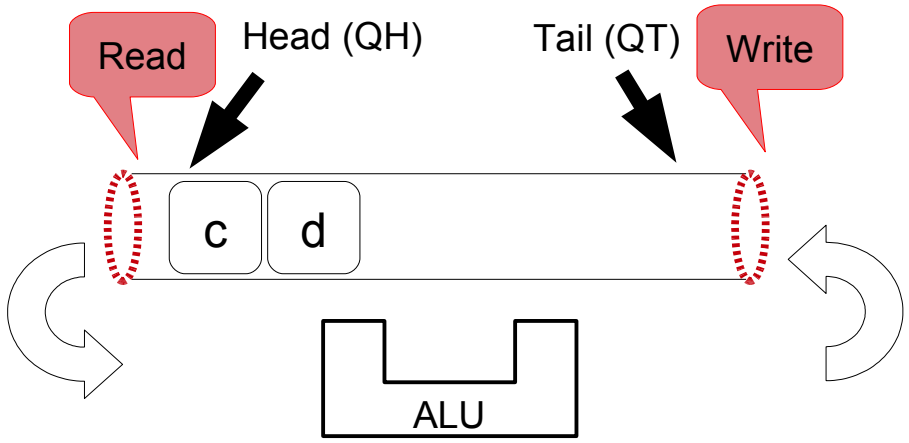


Queue machines

"x = (a+b) / (c-d)"

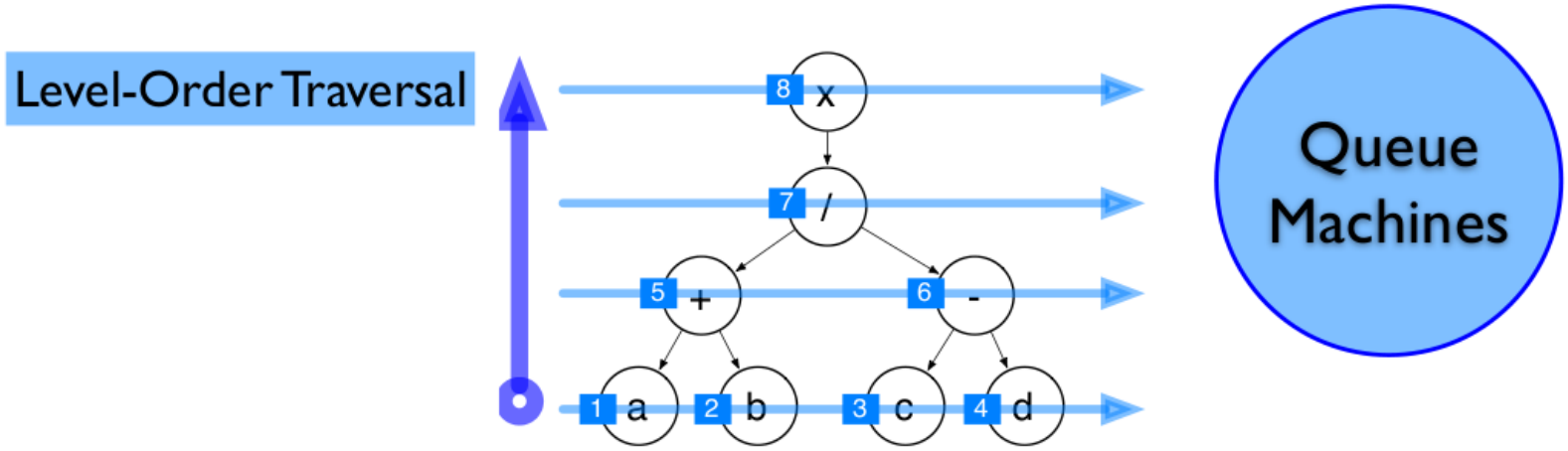



```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

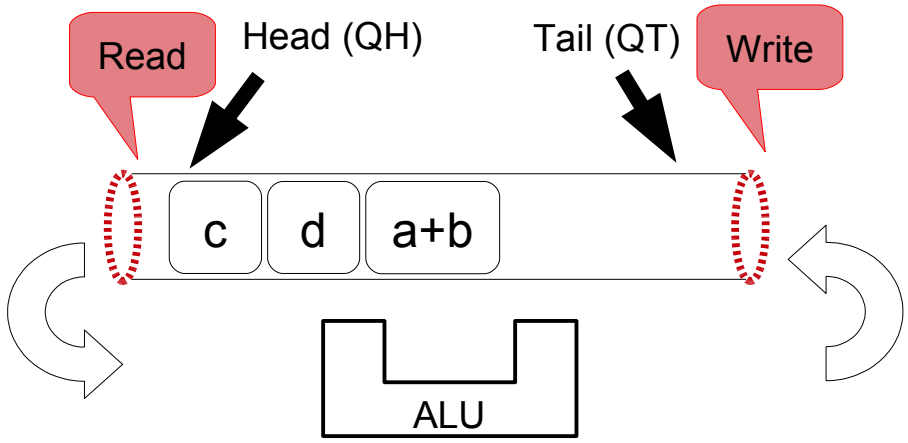


Queue machines

"x = (a+b) / (c-d)"

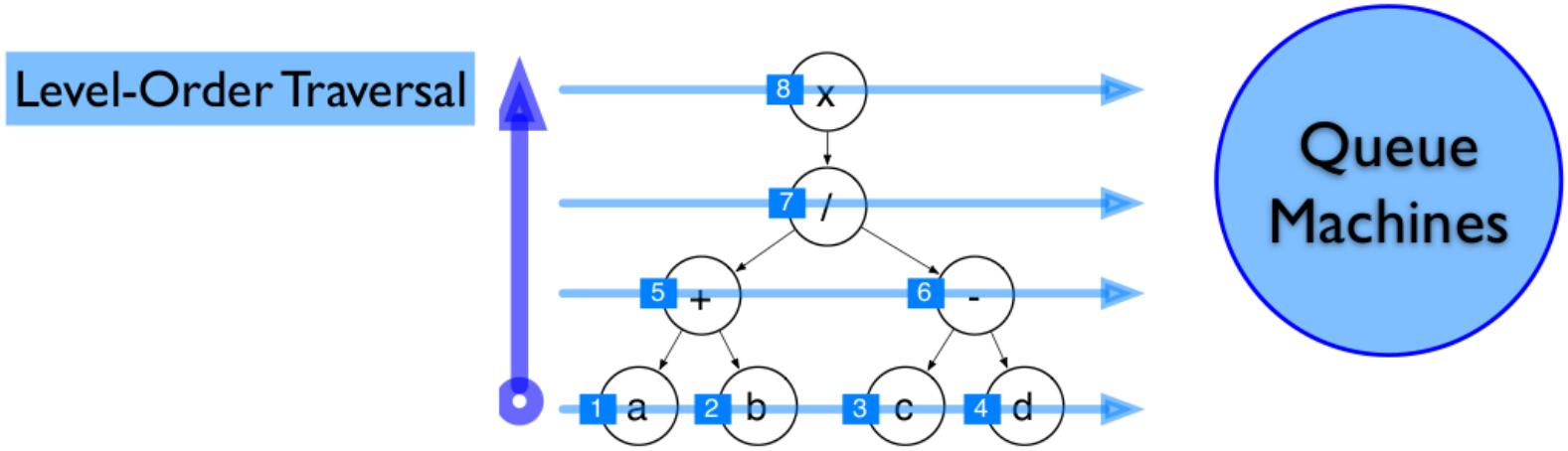



```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

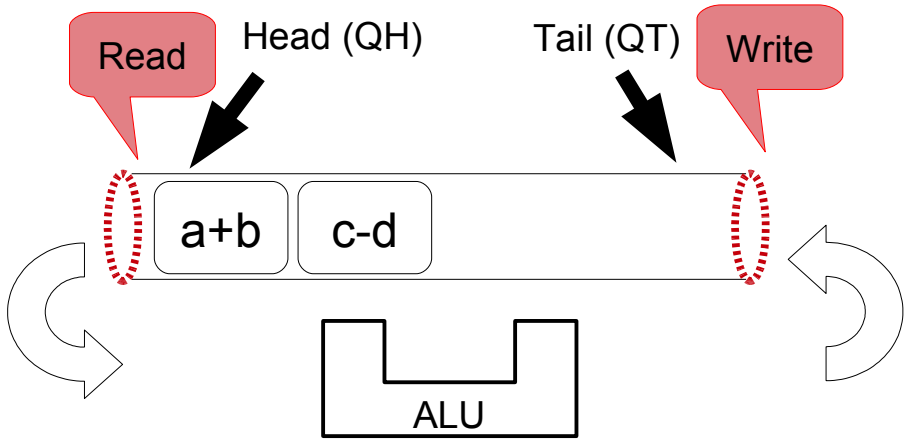


Queue machines

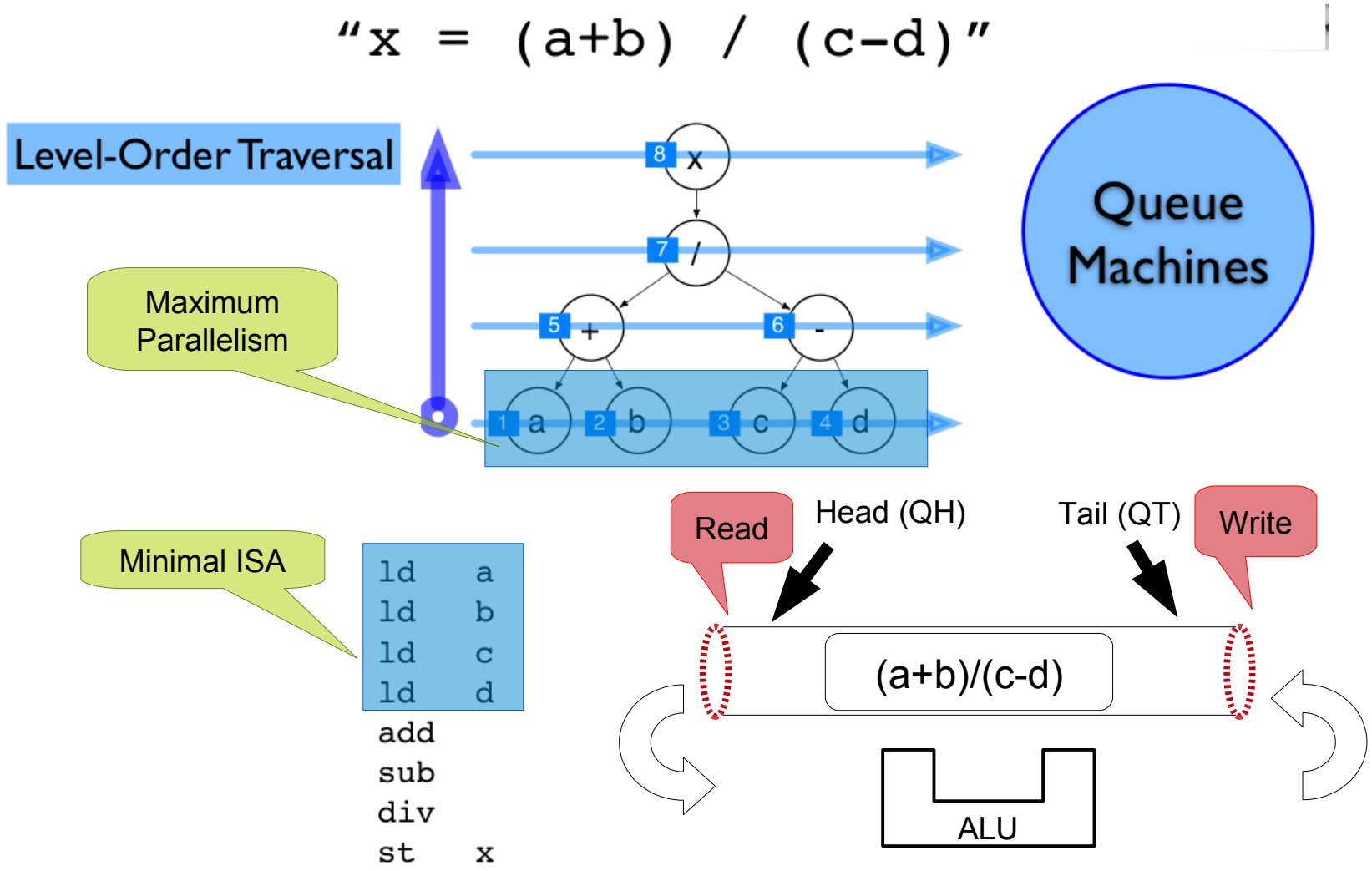
"x = (a+b) / (c-d)"




```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```



Queue machines



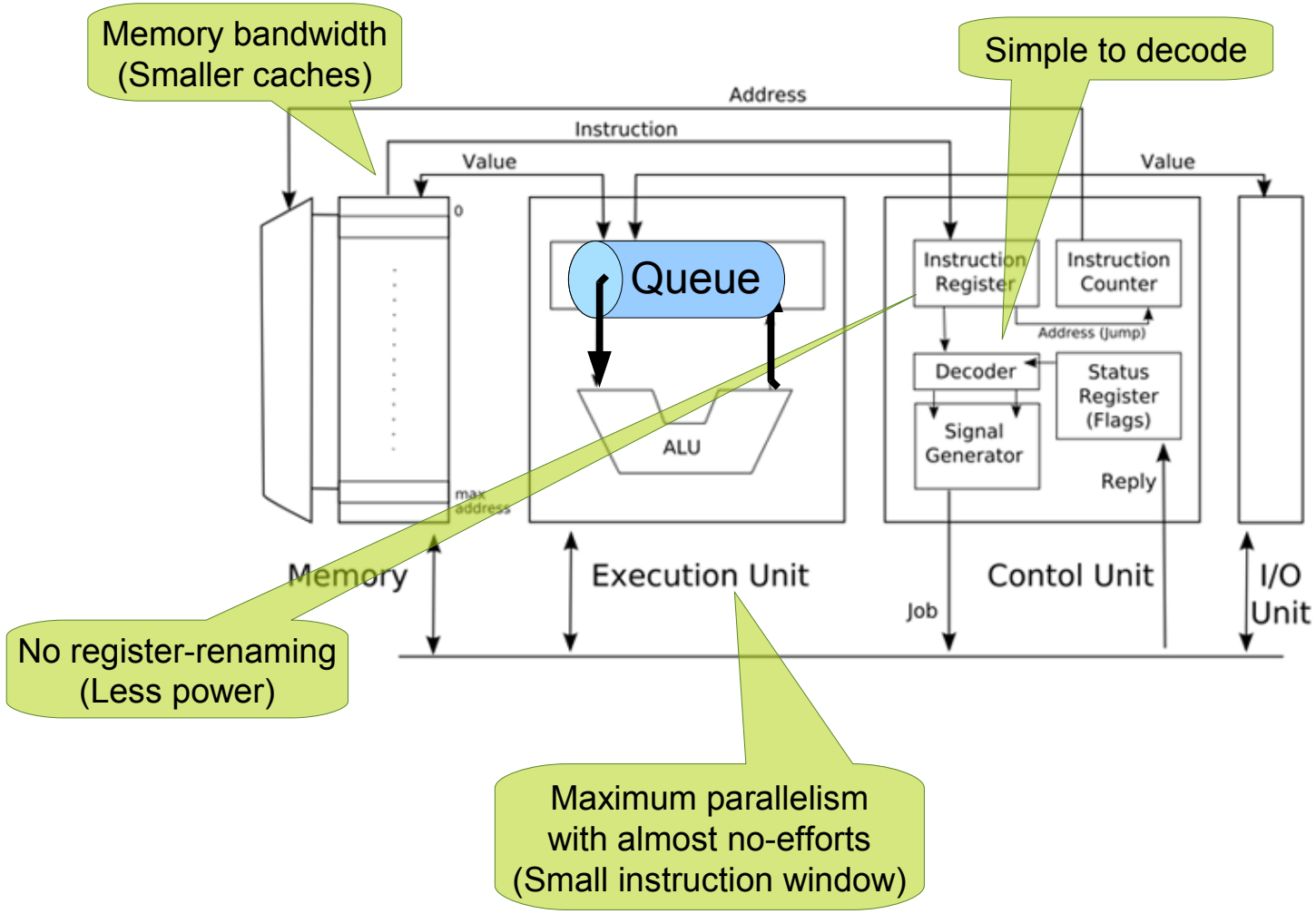
Instruction-Level Parallelism

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

- Dynamically scheduled
  - –Superscalar processors
- Statically scheduled
  - –VLIW processors

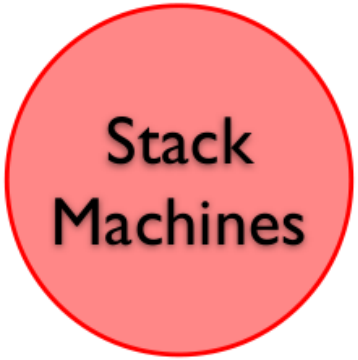
Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2	IF	ID	EX	MEM	WB		
3		IF	ID	EX	MEM	WB	
4		IF	ID	EX	MEM	WB	
5			IF	ID	EX	MEM	WB
6			IF	ID	EX	MEM	WB
7				IF	ID	EX	MEM
8				IF	ID	EX	MEM
9					IF	ID	EX
10					IF	ID	EX
	1	2	3	4	5	6	7

Why queue machines ?

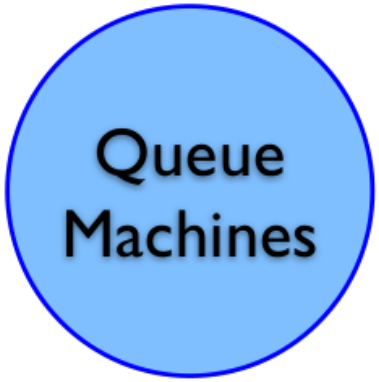


A closer look


```
ld   r1, a
ld   r2, b
add  r1, r1, r2
st   r1, tmp
ld   r2, c
ld   r1, d
sub  r2, r2, r1
ld   r1, tmp
div  r1, r1, r2
st   r1, x
```




```
psh  a
psh  b
add
psh  c
psh  d
sub
div
st   x
```




```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

A closer look

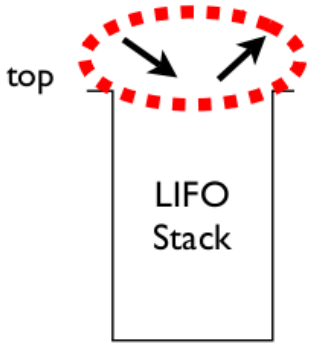
Shorter instructions
- Better I-cache performance
- Easier fetching and decoding
- Simpler hardware logic

No false dependencies
- No register renaming logic
- Less power consumption

```
ld  r1, a
ld  r2, b
add r1, r1, r2
st  r1, tmp
ld  r2, c
ld  r1, d
sub r2, r2, r1
ld  r1, tmp
div r1, r1, r2
st  r1, x
```

Stack Machines

```
psh  a
psh  b
add
psh  c
psh  d
sub
div
st   x
```



Queue Machines

```
ld   a
ld   b
ld   c
ld   d
add
sub
div
st   x
```

tail

A closer look

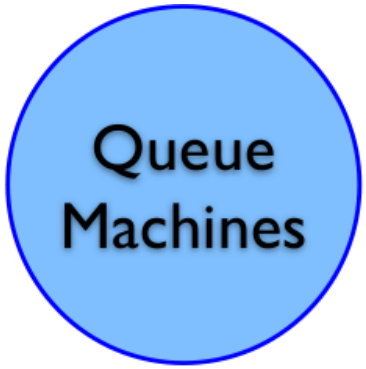
Shorter instructions
- Better I-cache performance
- Easier fetching and decoding
- Simpler hardware logic

No false dependencies
- No register renaming logic
- Less power consumption

```
ld  r1, a
ld  r2, b
add r1, r1, r2
st  r1, x
```

No Serialization at TOS
- No performance bottleneck
- No special techniques
- High parallelism

```
psh a
psh b
add
```



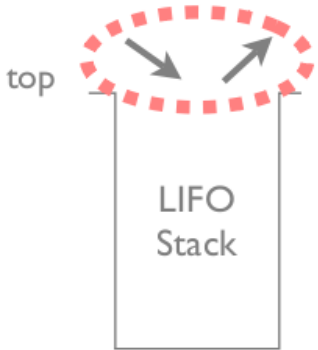

```
ld  a
ld  b
ld  c
```

Post-Order Traversal

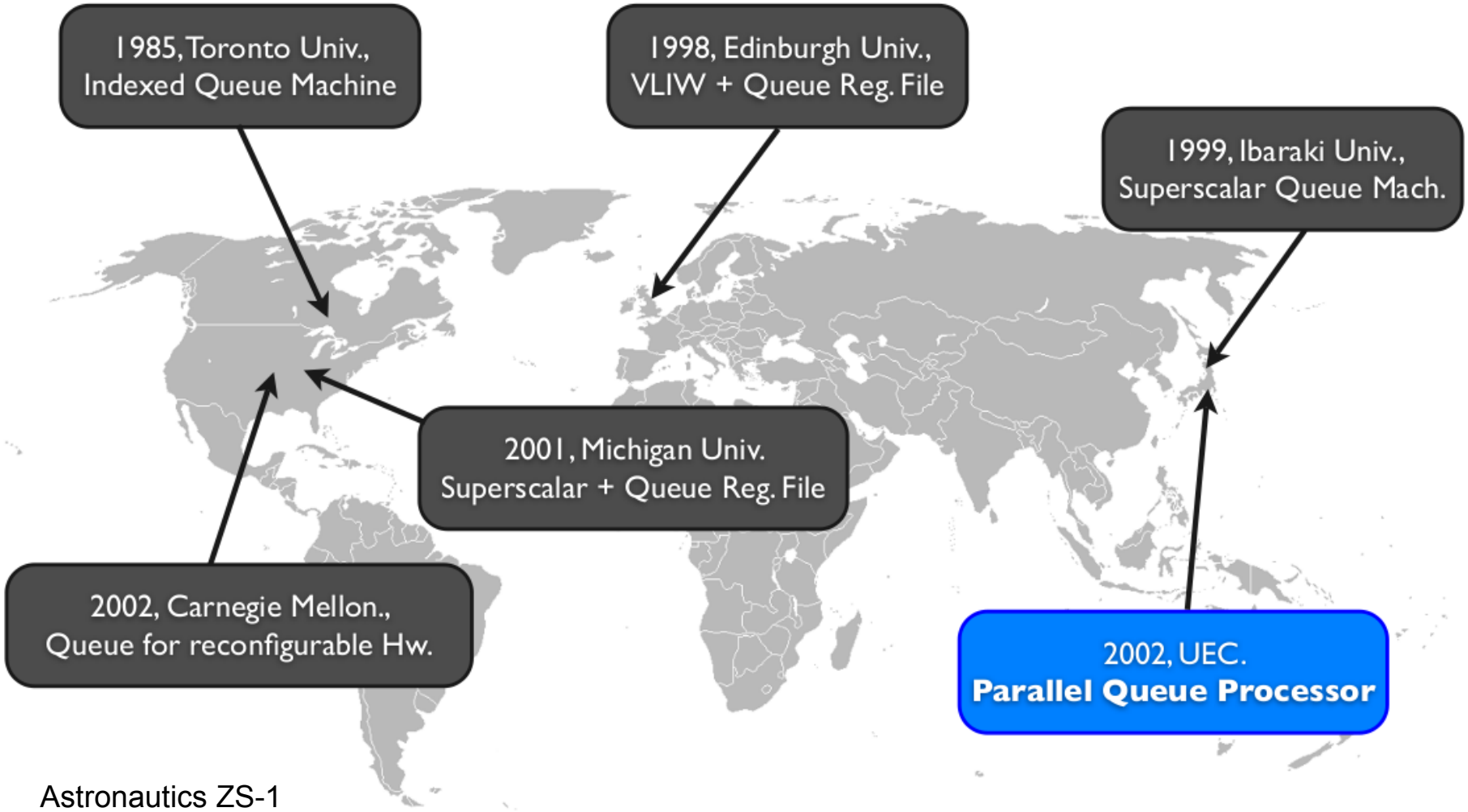
Level-Order Traversal

All parallelism available in the application
- Smaller instruction window
- Simpler hardware
- Less power consumption

Invisible queues



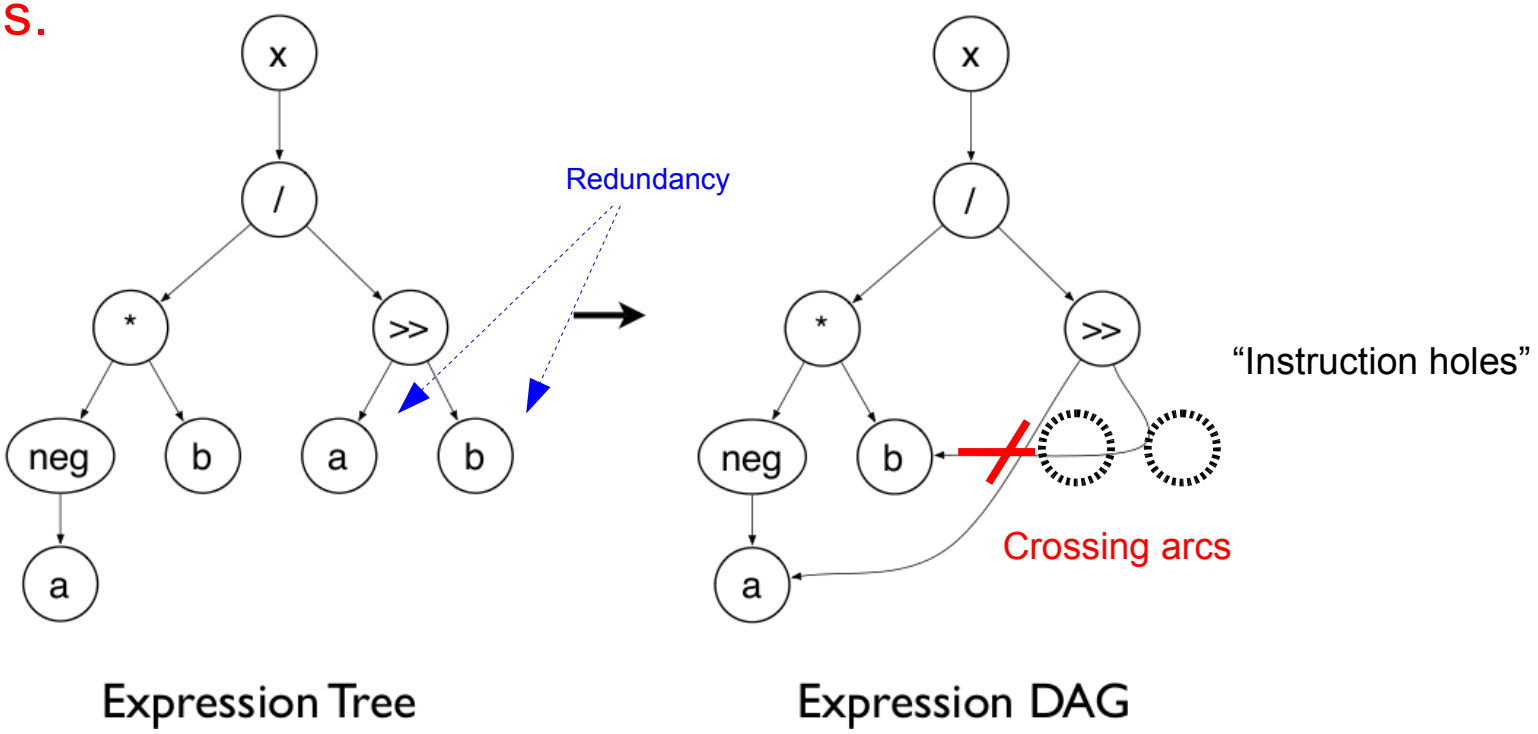
Related work on queue machines



Astronautics ZS-1
Decoupled Access-Execute architecture

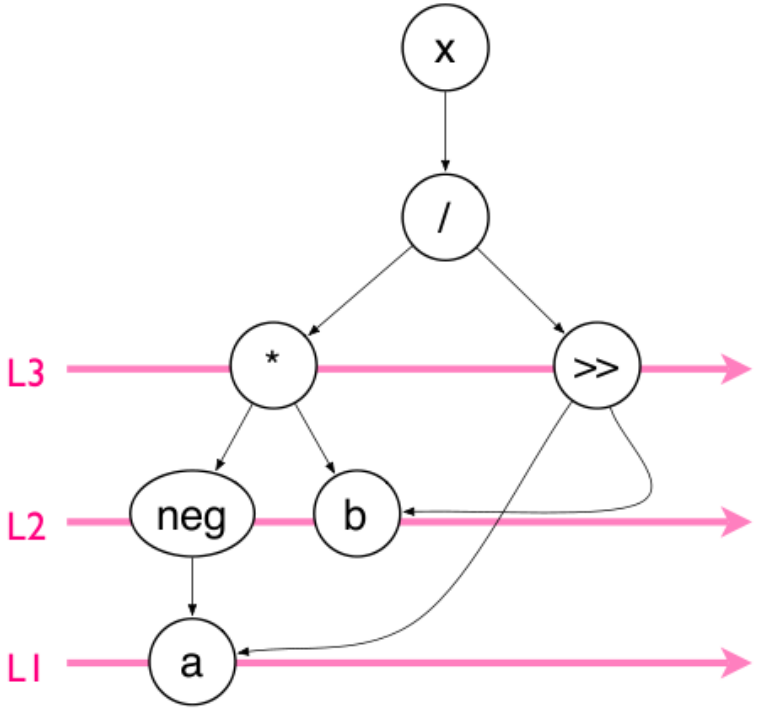
Main problem of queue machines

- Queue program:
  - Build an expression tree
  - Schedule the tree in level-order manner
- If the expression tree is transformed into a directed acyclic graph (DAG), level-order scheduling no longer produces correct programs.



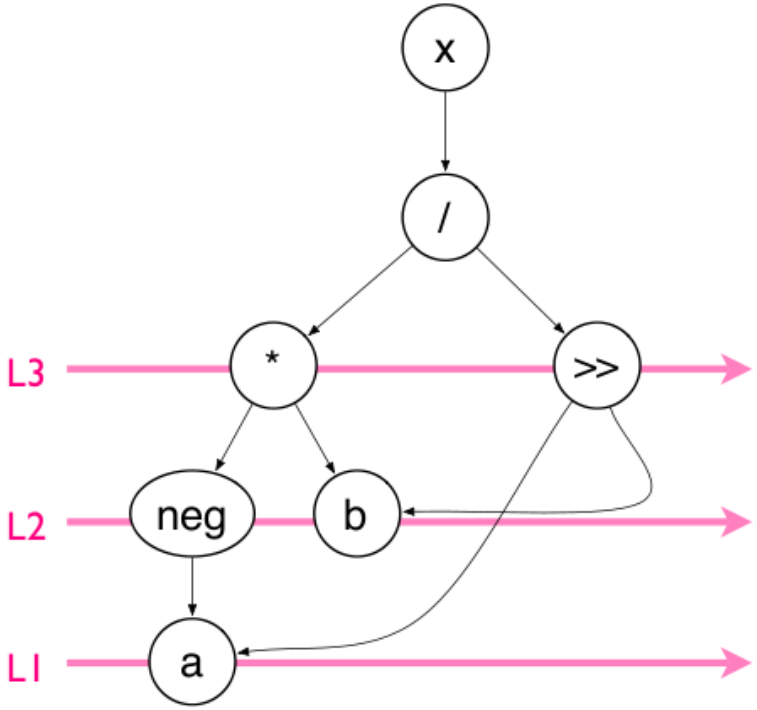
Main problem of queue machines

```
ld   a
neg
ld   b
mul
rsh
```



Main problem of queue machines

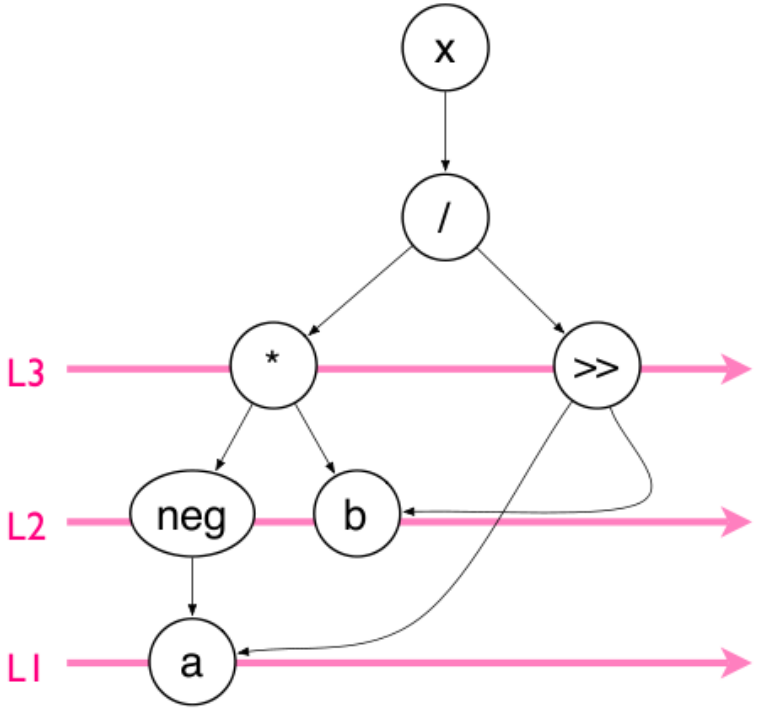
```
ld   a
neg
ld   b
mul
rsh
```



QH

Main problem of queue machines

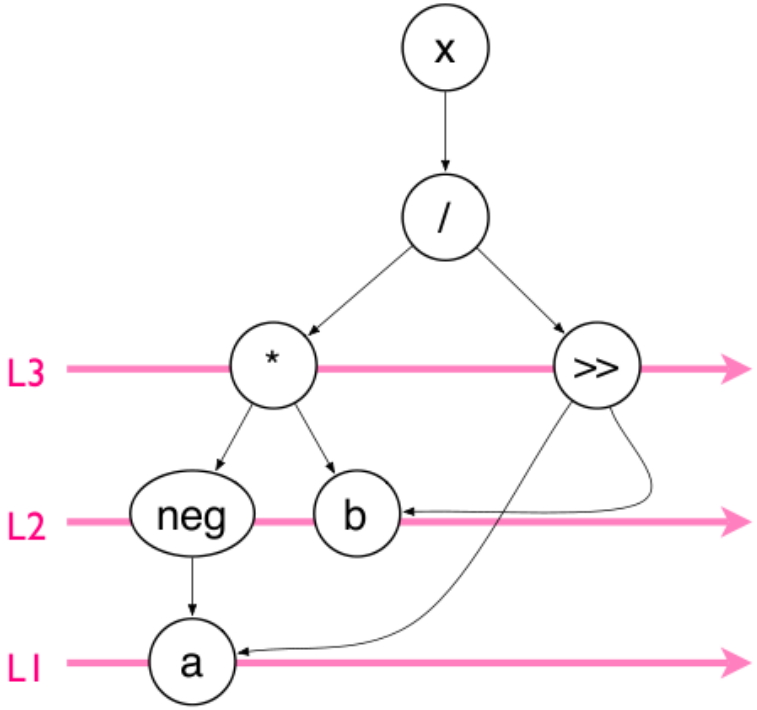
```
ld   a
neg
ld   b
mul
rsh
```



a

Main problem of queue machines

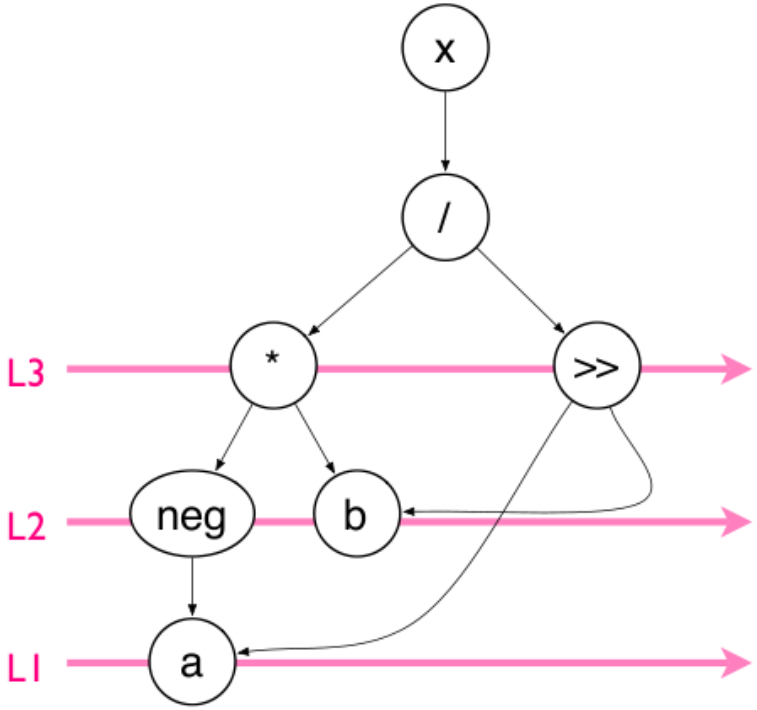
```
ld   a
neg
ld   b
mul
rsh
```



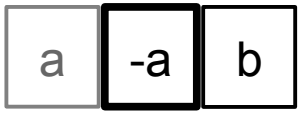
QH

Main problem of queue machines

```
ld   a
neg
ld   b
mul
rsh
```

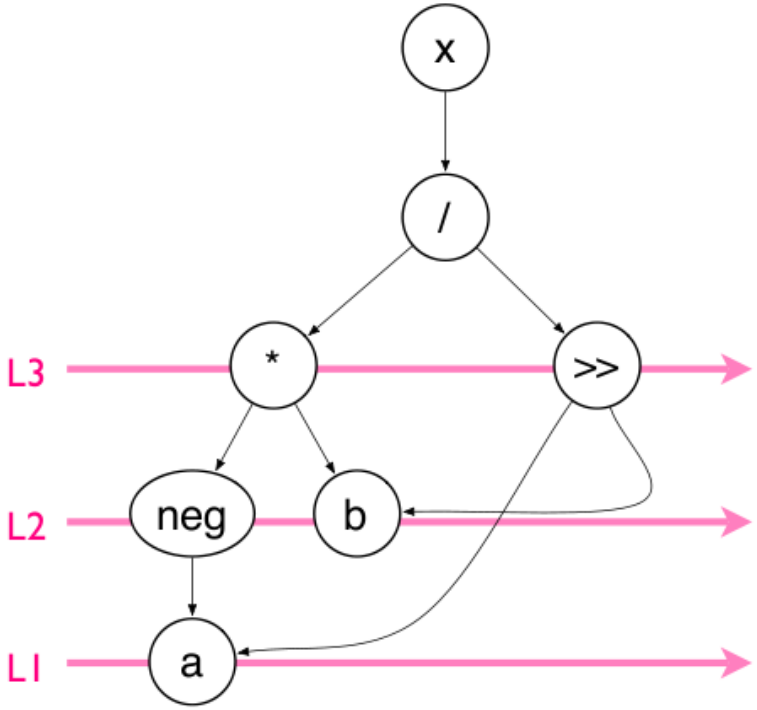


QH



Main problem of queue machines

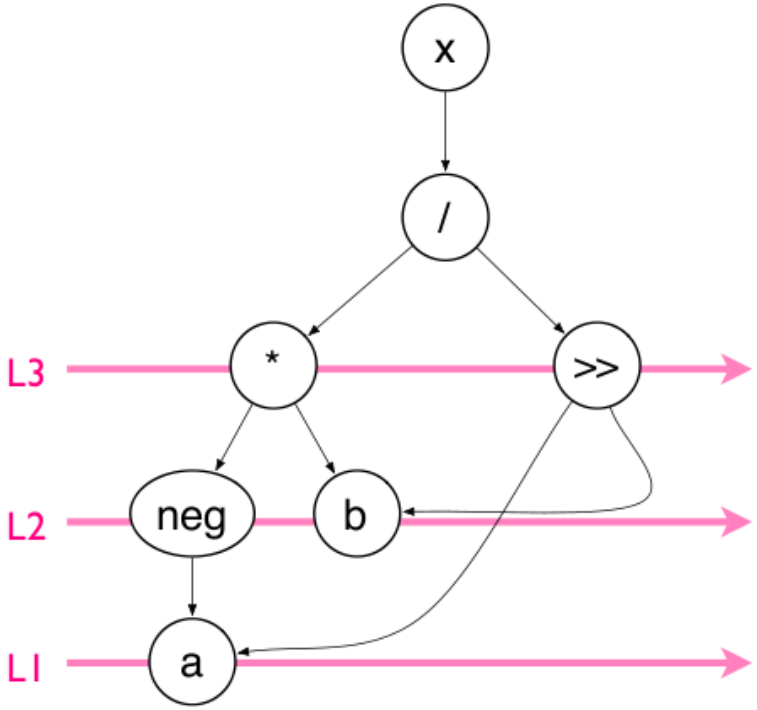
```
ld   a
neg
ld   b
mul
rsh
```



a	-a	b

Main problem of queue machines

```
ld   a
neg
ld   b
mul
rsh
```

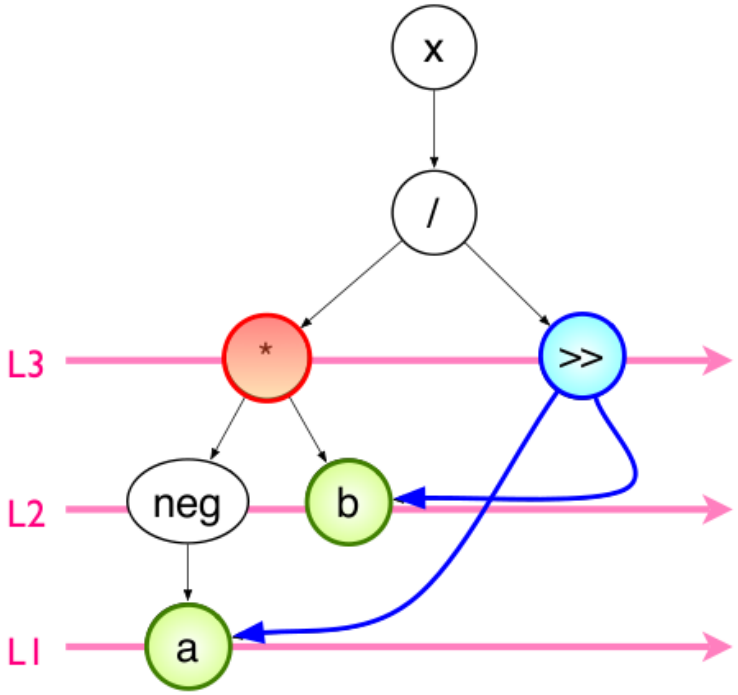


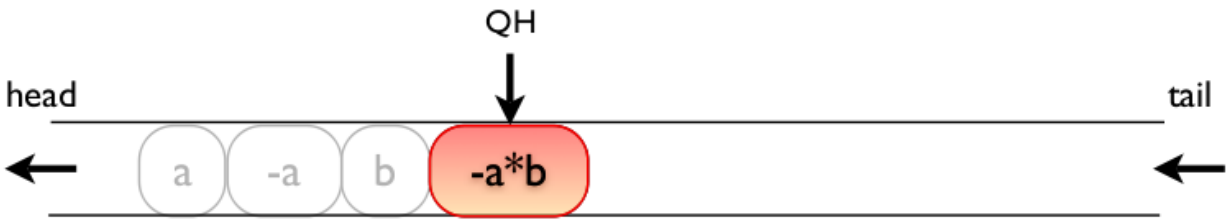
QH

a	-a	b	-a*b

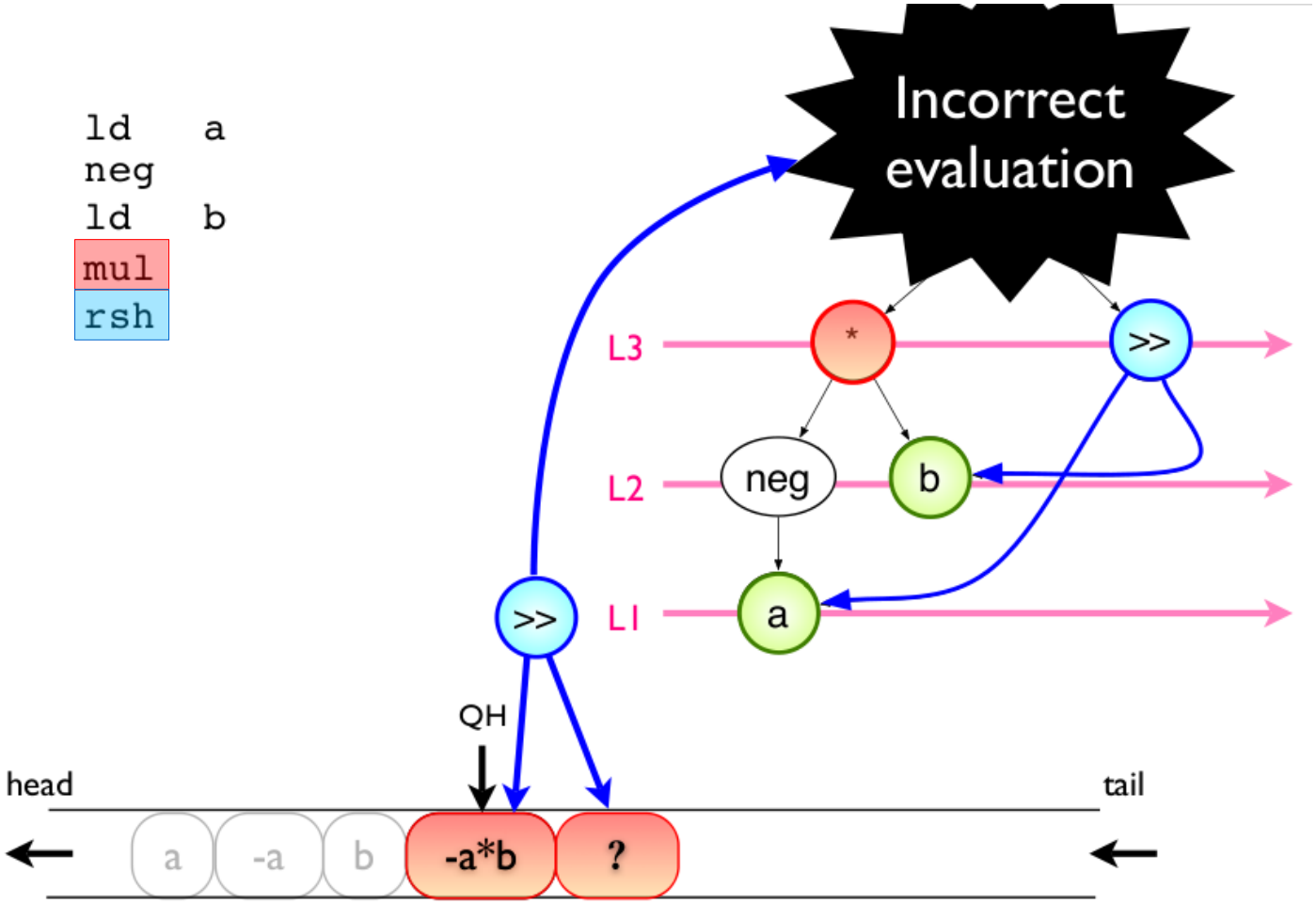
Main problem of queue machines

```
ld   a
neg
ld   b
mul
rsh
```





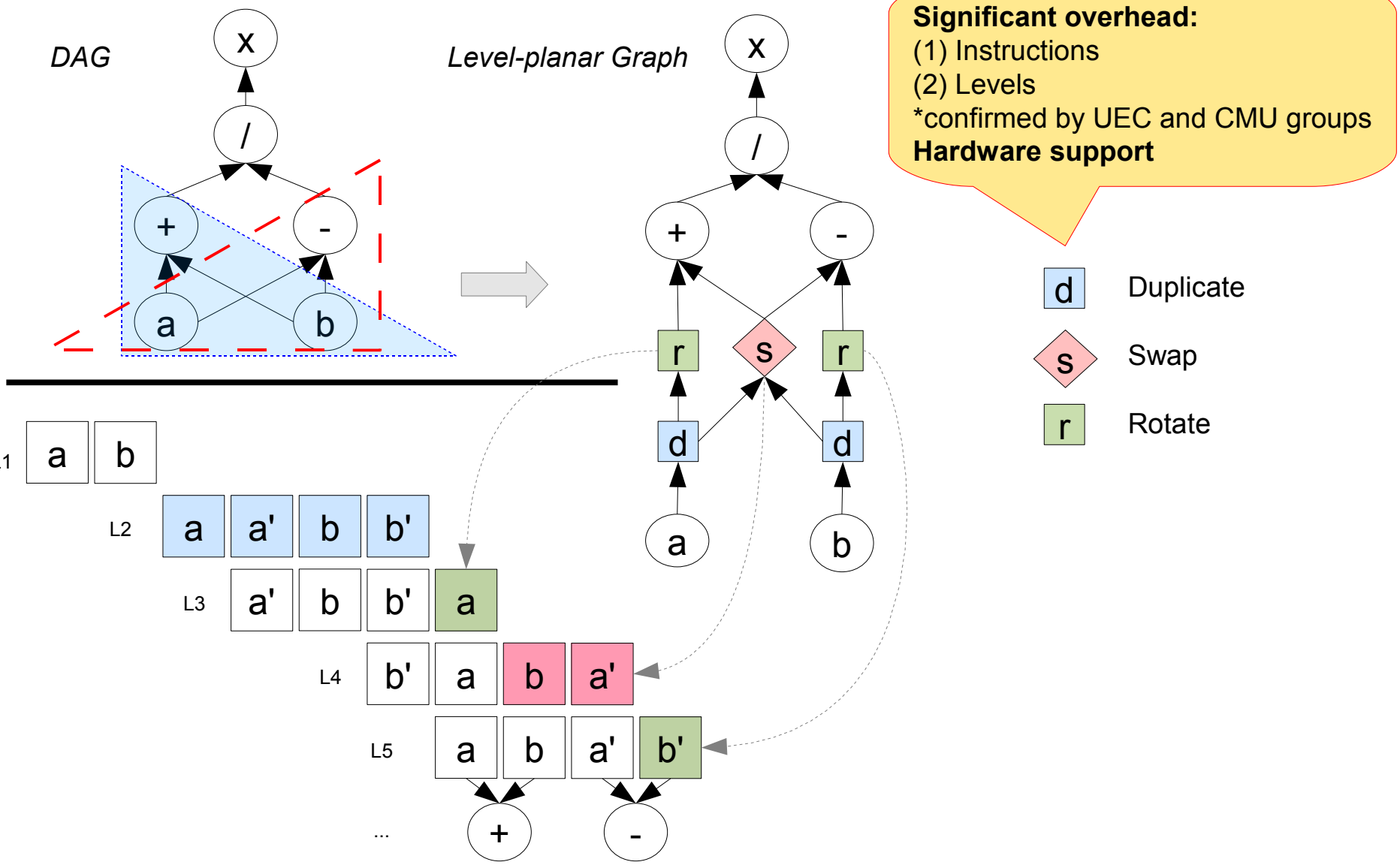
Main problem of queue machines



"Pure" queue computing (Producer-Consumer)

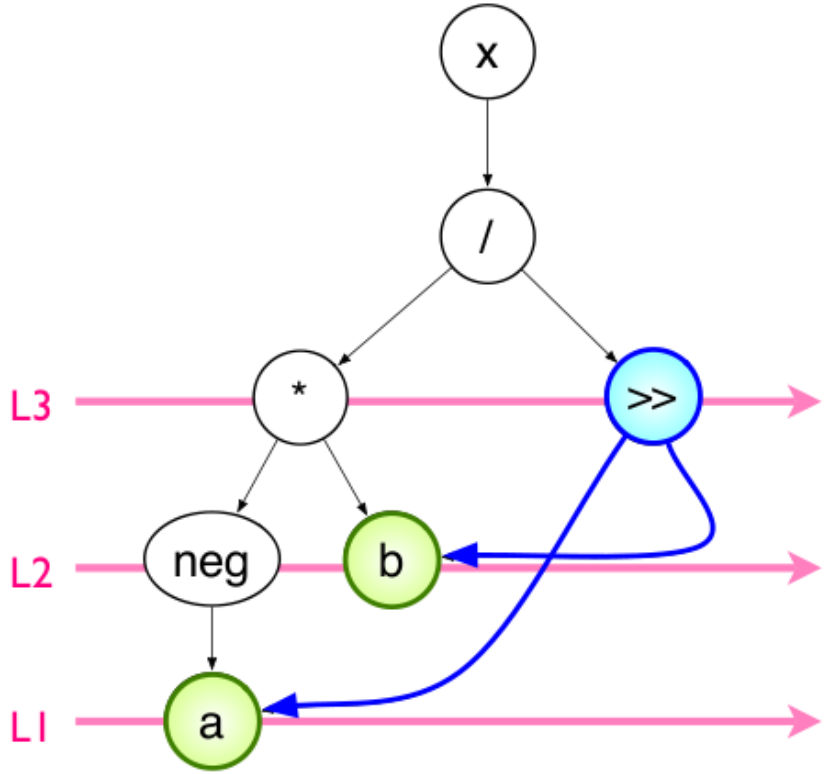
- No flexibility
  - Writes always at QT
  - Reads always at QH
- DAGs cannot be executed correctly.
- Solution:
  - Eliminate instruction holes by *duplicating* data.
  - Eliminate cross arcs by *swapping/rotating* operands in the queue.
- **Level-planar graph**
  - No cross arcs
  - All edges span one level
  - There are efficient algorithms to test level-planarity but heuristics must be used to create a level-planar graph from an arbitrary DAG.

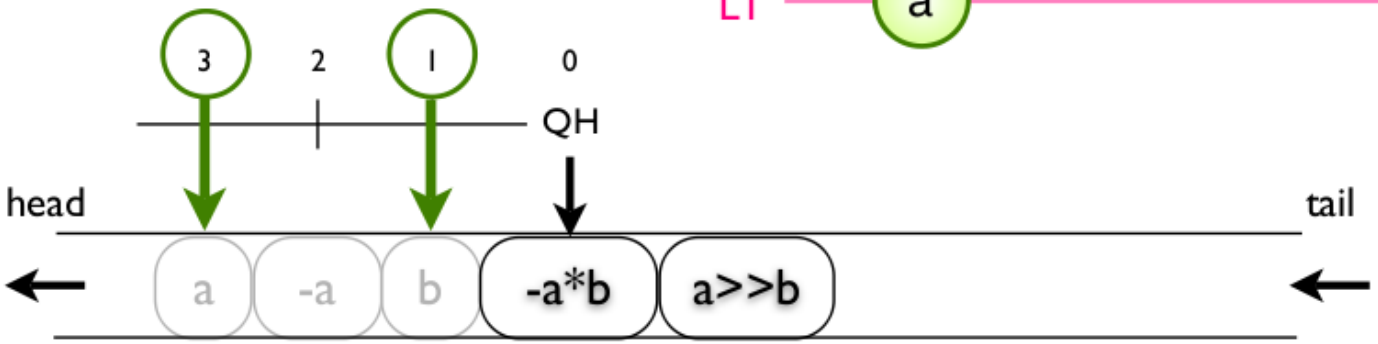
Making level-planar graphs



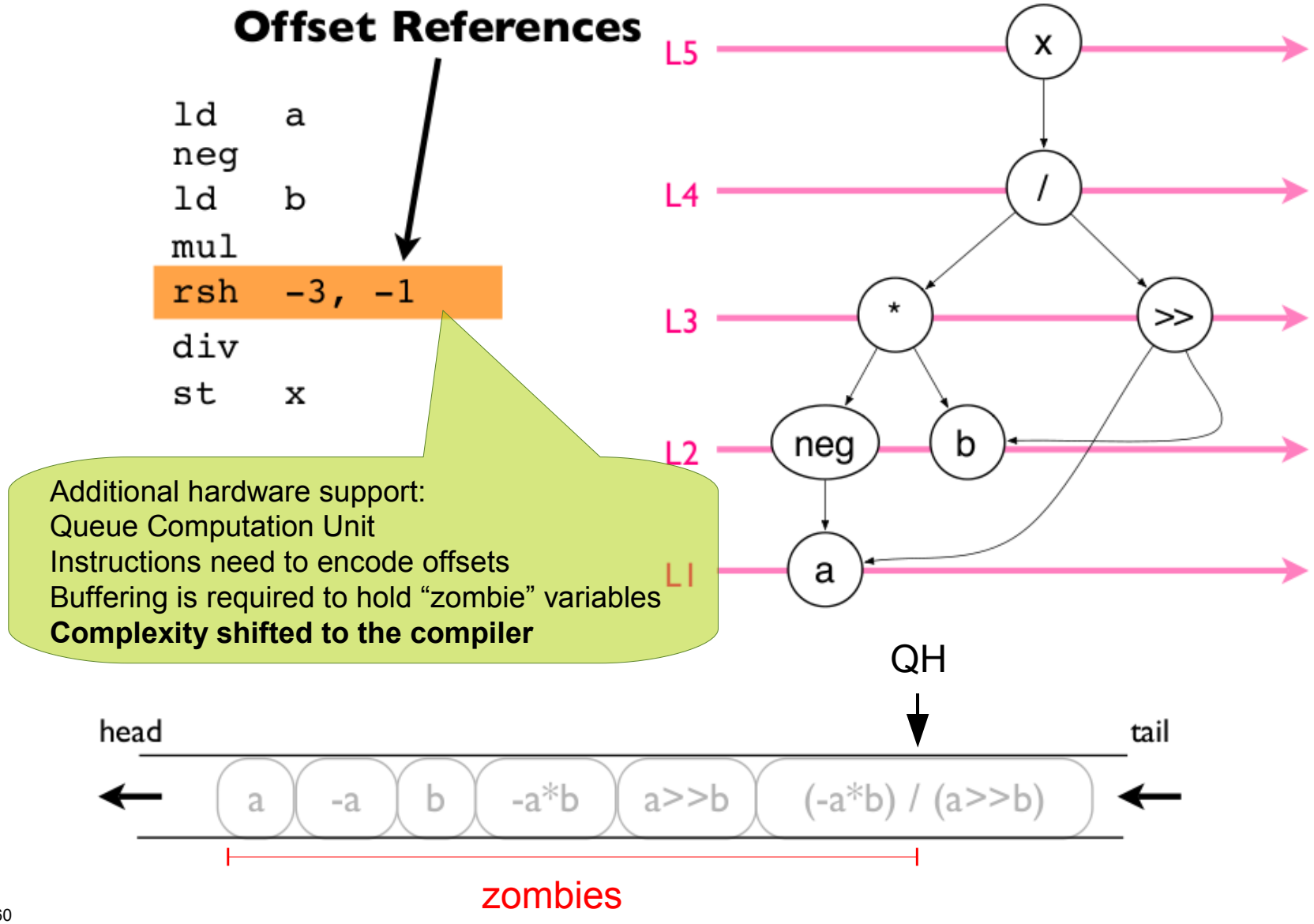
Our solution: Producer-Order Queue Computation Model (PQCM)

```
ld    a
neg
ld    b
mul
rsh   -3, -1
```





Producer-Order Queue Computation Model (PQCM)



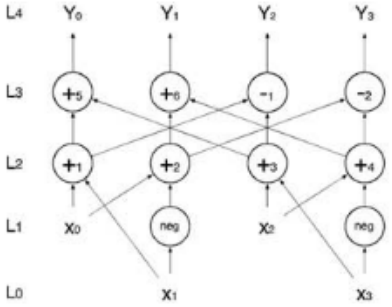
Producer-Order graphs vs Level-planar graphs

Table 6
Code size and depth of application graphs comparison.

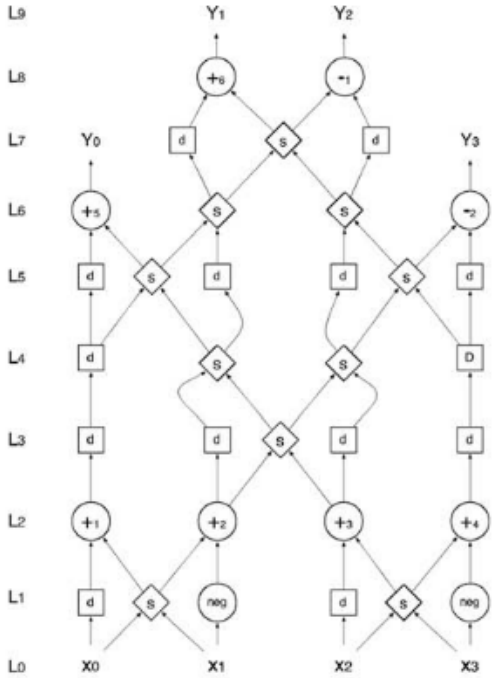
Application	Level-planar graph		Producer order graph	
	Code size	Depth	Code size	Depth
dct1	537	49	244	18
fft8_iterative	909	41	176	7
Haar16	918	17	248	6
rc6	330	42	148	23
Idea	1462	235	606	160
Popcount	229	24	62	5

```
for(i=0; i<x; i+=4) {
    x0 = A[i]; x1 = A[i+1];
    x2 = A[i+2]; x3 = A[i+3];
    tmp1 = x0+x1; tmp2 = x0+(-x1);
    tmp3 = x2+x3; tmp4 = x2+(-x3);
    tmp5 = tmp1+tmp3;
    tmp6 = tmp2+tmp4;
    tmp7 = tmp1-tmp3;
    tmp8 = tmp2-tmp4;
    Y[i] = tmp5; Y[i+1] = tmp6;
    Y[i+2] = tmp7; Y[i+3] = tmp8;
}
```

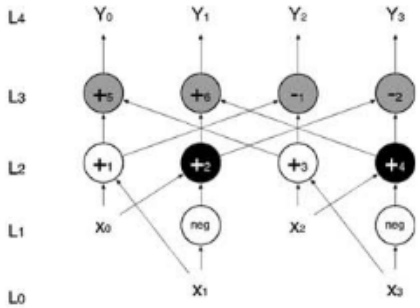
(a) Sample C program



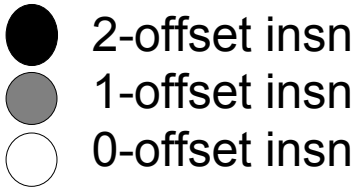
(b) Original Graph



(c) Level Planar Graph



(d) Producer Order Graph



PQCM Instructions with offsets

Production of data (writes) is fixed at QT.

Binary instructions

Unary instructions

0-offset

add

add 0, 1

1-offset

neg -2

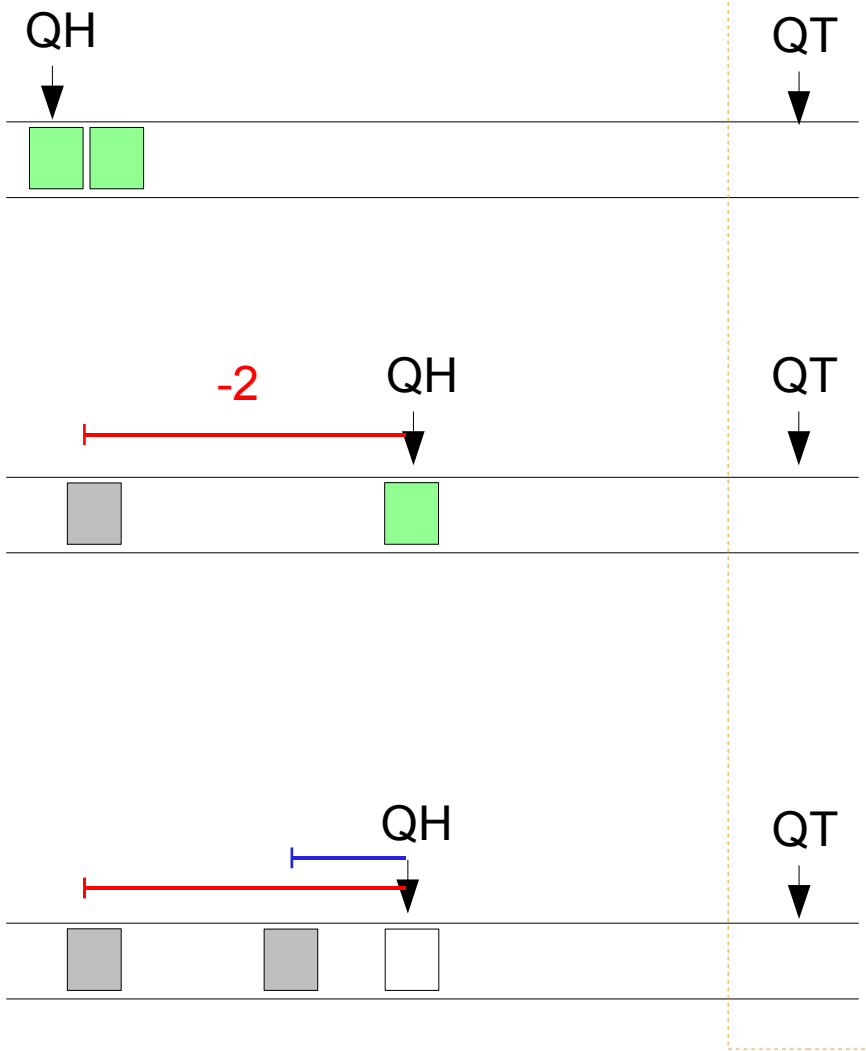
mul 0, -2

div 0, 0

2-offset

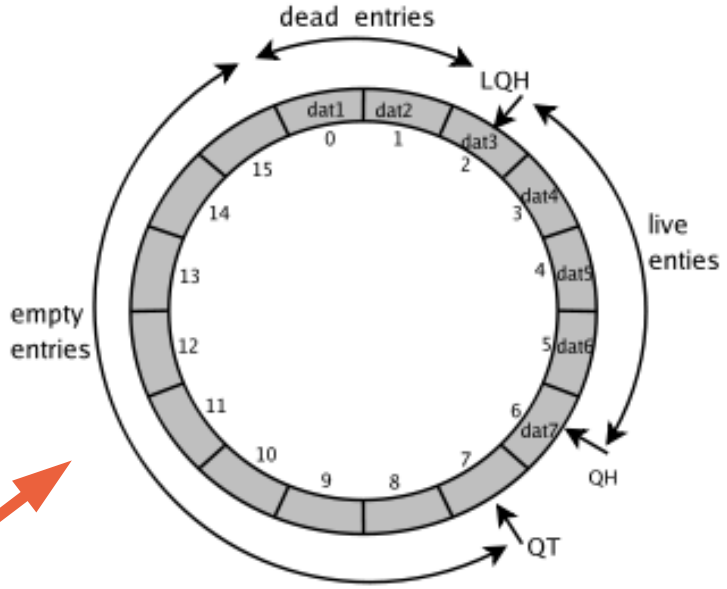
add -2, -3

sub -5, -1



Parallel Queue Processor (PQP)

- 16-bit fixed-length instructions
- 32-bit processor (datapath)
- Addressable offset references
- 8 addressable special purpose registers
  - –Frame pointer, stack pointer
  - –Data/Address registers
- 4-way superscalar execution
- Register file (circular queue)
  - –256 queue registers
- VHDL Simulators
- Virtual machine
- **Queue compiler**

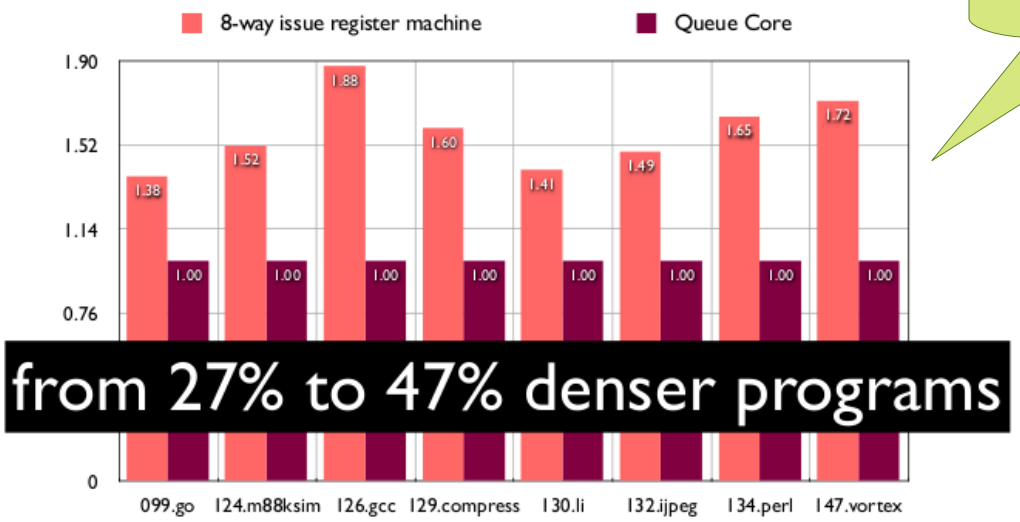


Cores	Speed (SPD)	Speed (ARA)	Average Power(mW)
PQP	22.5	21.5	120
SH-2	15.3	14.1	187.5
ARM7	25.2	24.5	22
LEON2	27.5	26.7	458
MicroBlaze	26.7	26.7	135
QC-2	25.5	24.2	90

Characteristics of PQP programs

Dense programs

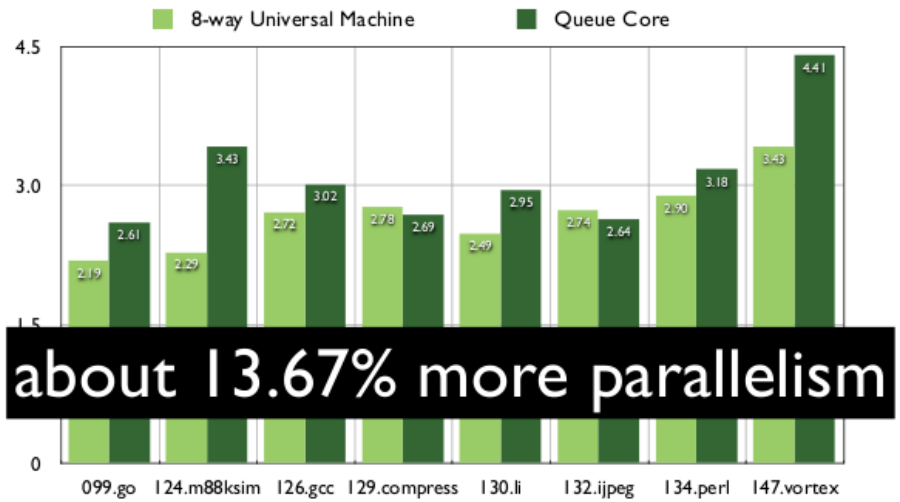
Code Size



from 27% to 47% denser programs

High parallelism

Instruction Level Parallelism



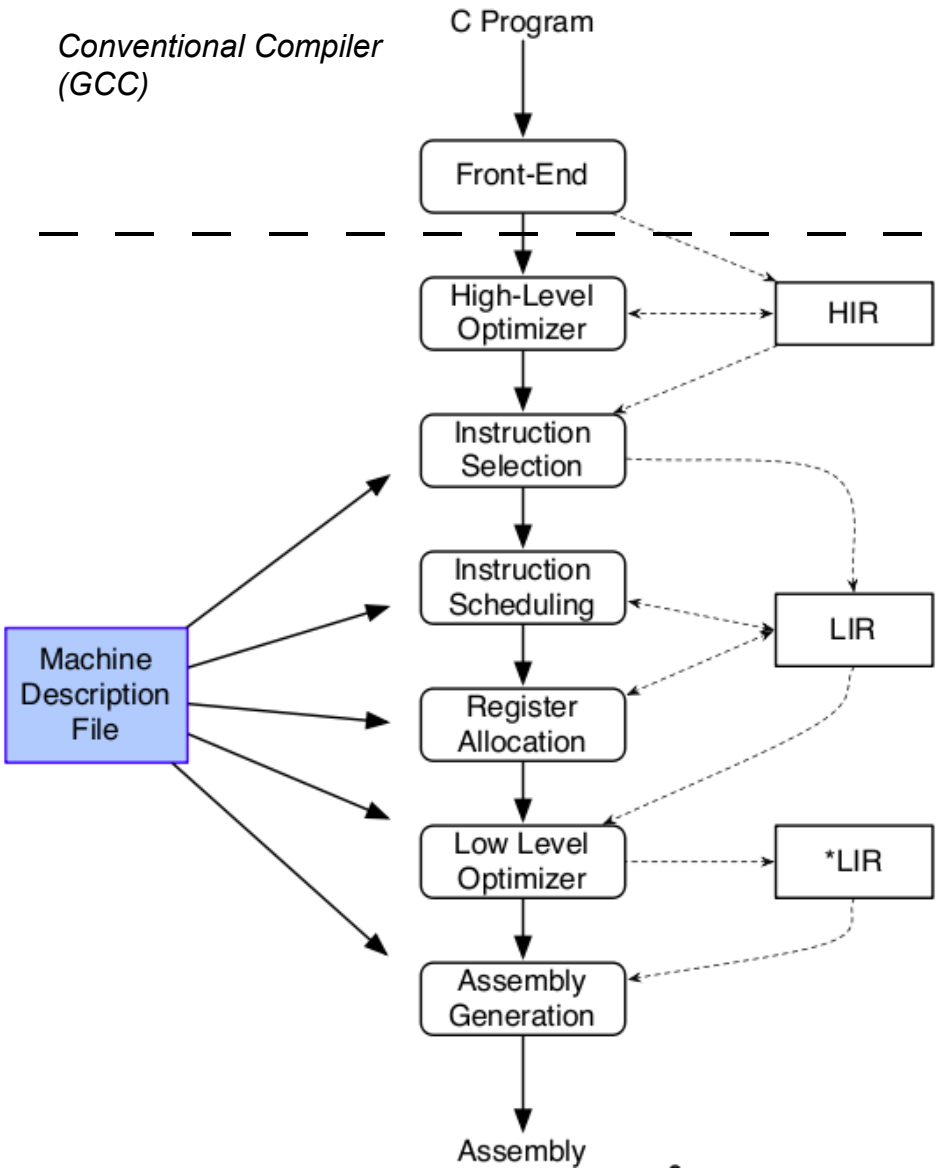
about 13.67% more parallelism

Queue Compiler Framework

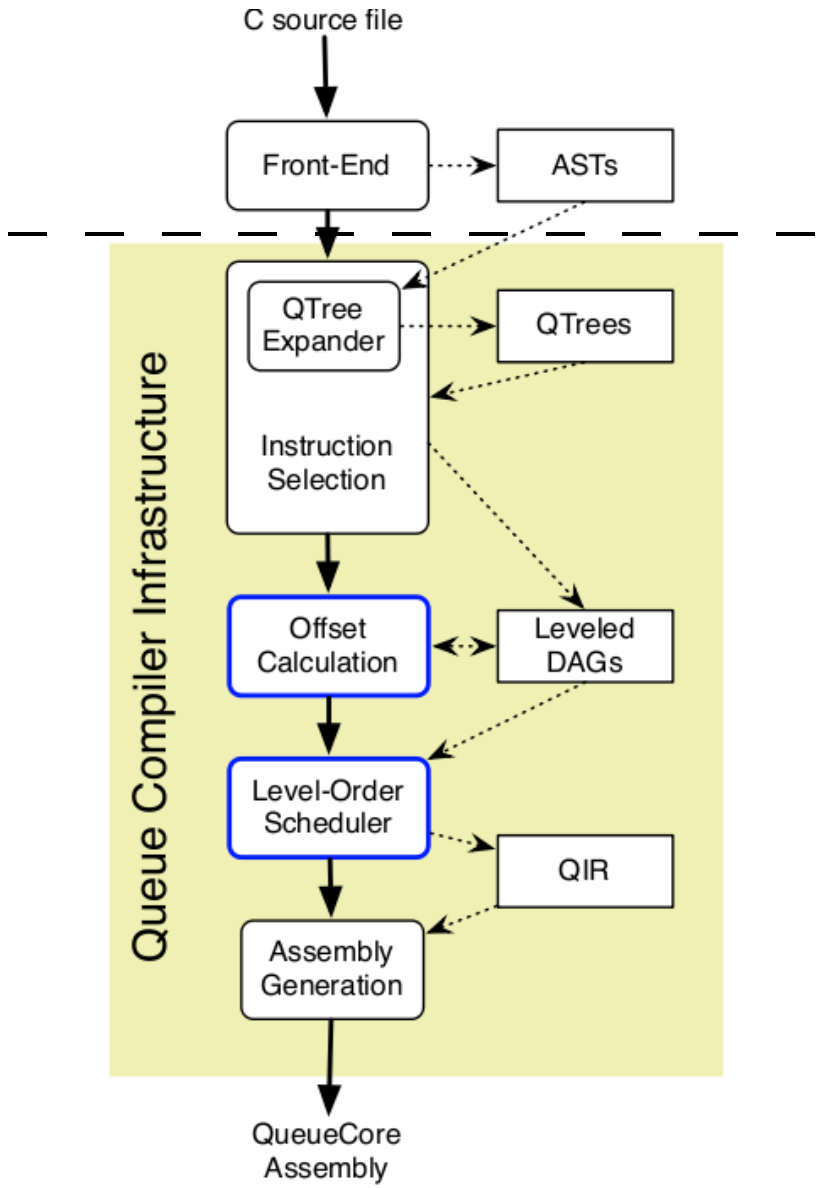
Part II

Queue Compiler

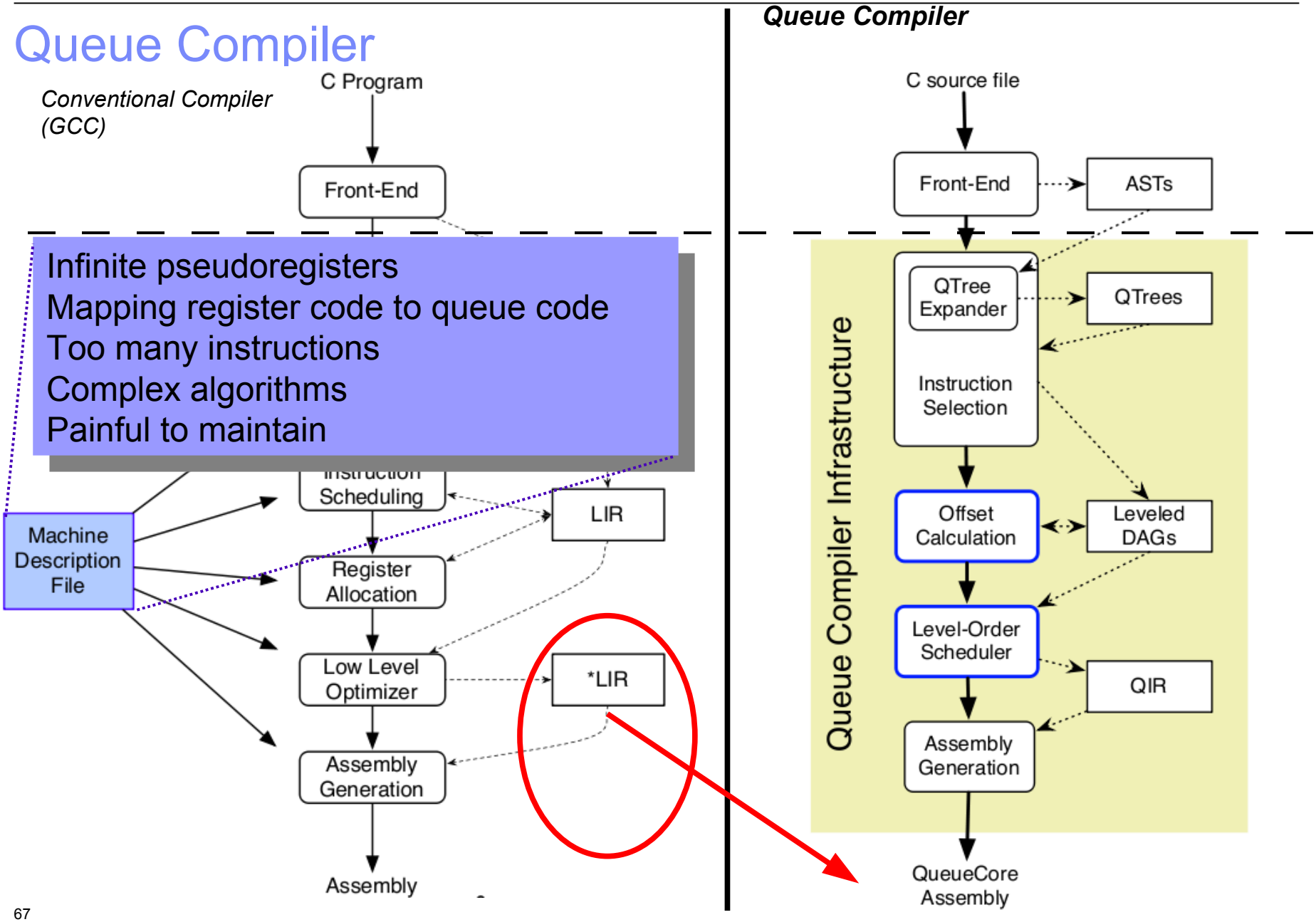
Conventional Compiler (GCC)



Queue Compiler



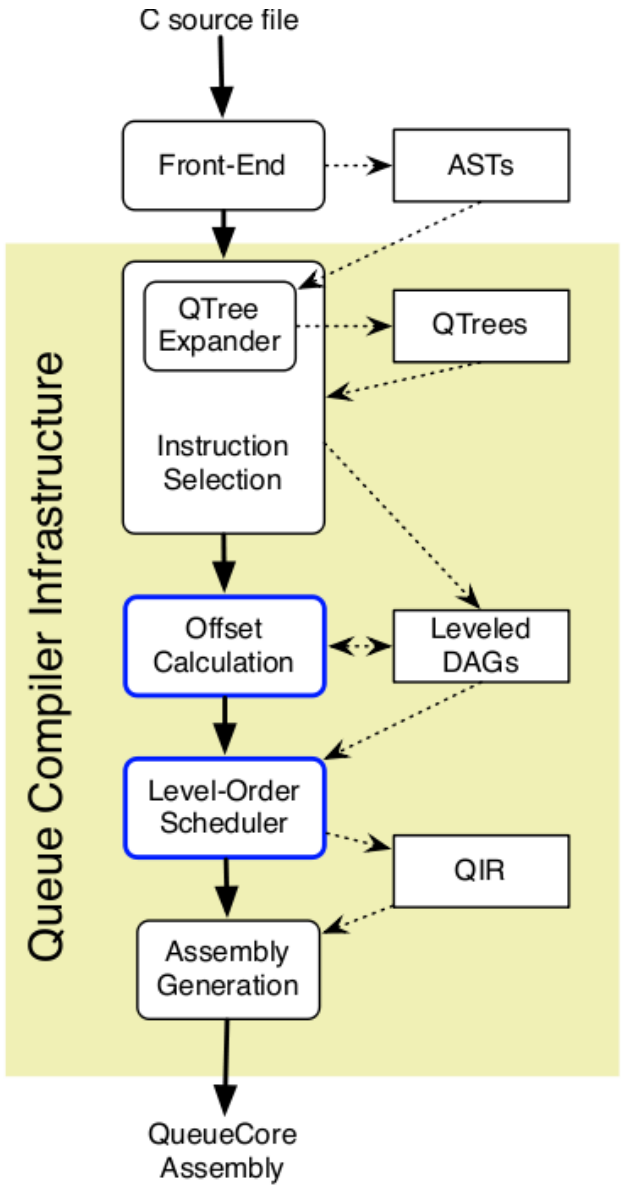
Queue Compiler



Queue Compiler

- Eliminates the concept of registers
- Built for the queue computation model
  - Suitable data structures
  - Queue compilation algorithms
    - **Offset calculation**
    - **1-offset constraint**
    - **Queue utilization**
    - **Queue register allocation**
- Produces good code
- Easy to maintain
- Clean infrastructure
- Able to compile complete programs

Queue Compiler

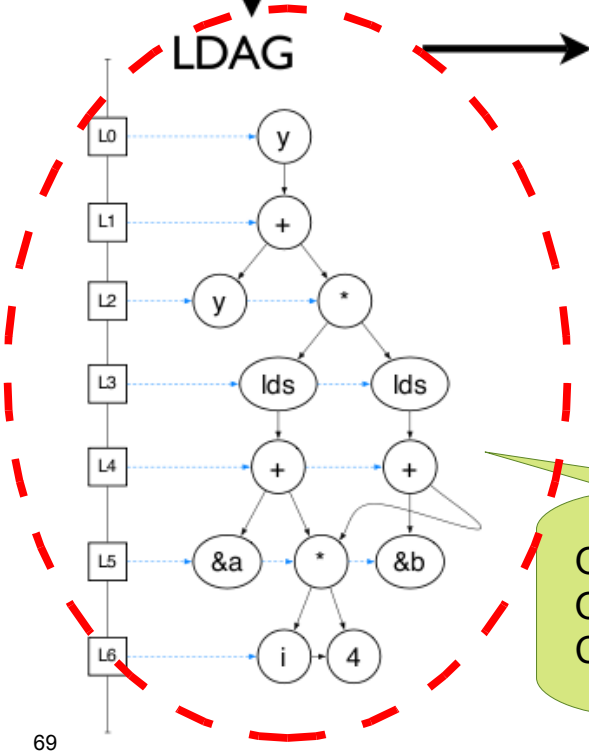


Compilation example

C program

```
int main(void)
{
     int A[10], B[10];
     int y, i;

     for(i=0; i<10; i++) {
          y = y + A[i] * B[i];
     }
}
```

LDAG



QIR

```
(QMARK_LEVEL [id=2])
(PUSH_Q [iws] (LOCAL_VAR_Q: 0($fp) )
(LOADI_Q [iwu] (INTEGER_Q: 4 )
(QMARK_LEVEL [id=3])
(LOAD_ADDR_Q [iws] (LOCAL_VAR_Q: 48($fp) )
(MUL_Q [iws]  ( QT, QH+0, QH+1 ))
(LOAD_ADDR_Q [iws] (LOCAL_VAR_Q: 8($fp) )
(QMARK_LEVEL [id=4])
(ADD_Q [iws]  ( QT, QH+0, QH+1 )
(ADD_Q [iws]  ( QT, QH+0, QH-1 )
(QMARK_LEVEL [id=5])
(SLOAD_Q [iws]  ( QT, QH+0, QH+0 ))
(SLOAD_Q [iws]  ( QT, QH+0, QH+0 ))
(QMARK_LEVEL [id=6])
(MUL_Q [iws]  ( QT, QH+0, QH+1 ))
(PUSH_Q [iws] (LOCAL_VAR_Q: 4($fp) )
(QMARK_LEVEL [id=7])
(ADD_Q [iws]  ( QT, QH+0, QH+1 ))
(QMARK_LEVEL [id=8])
(POPQ_Q [iws] (LOCAL_VAR_Q: 4($fp) )
```

PQP Assembly

```
L0:
    ld      iws, qt, ($fp)0
    ldi     iwu, qt, 4
    lea     iwu, qt, $fp, 48
    mul     iws, qt, qh, qh+1
    lea     iwu, qt, $fp, 8
    add     iws, qt, qh, qh+1
    add     iws, qt, qh, qh-1
    lds     iws, qt, qh
    lds     iws, qt, qh
    mul     iws, qt, qh, qh+1
    ld      iws, qt, ($fp)4
    add     iws, qt, qh, qh+1
    st      iws, qh, ($fp)4
    ld      iws, qt, ($fp)0
    ldi     iws, qt, 1
    add     iws, qt, qh, qh+1
    st      iws, qh, ($fp)0
```

Offset calculation
Queue register allocation
Constrained compilation

Leveled DAG (LDAG)

$$\vec{G} = (V, \vec{E})$$

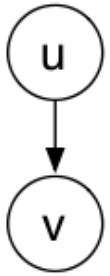
Nodes

Edges

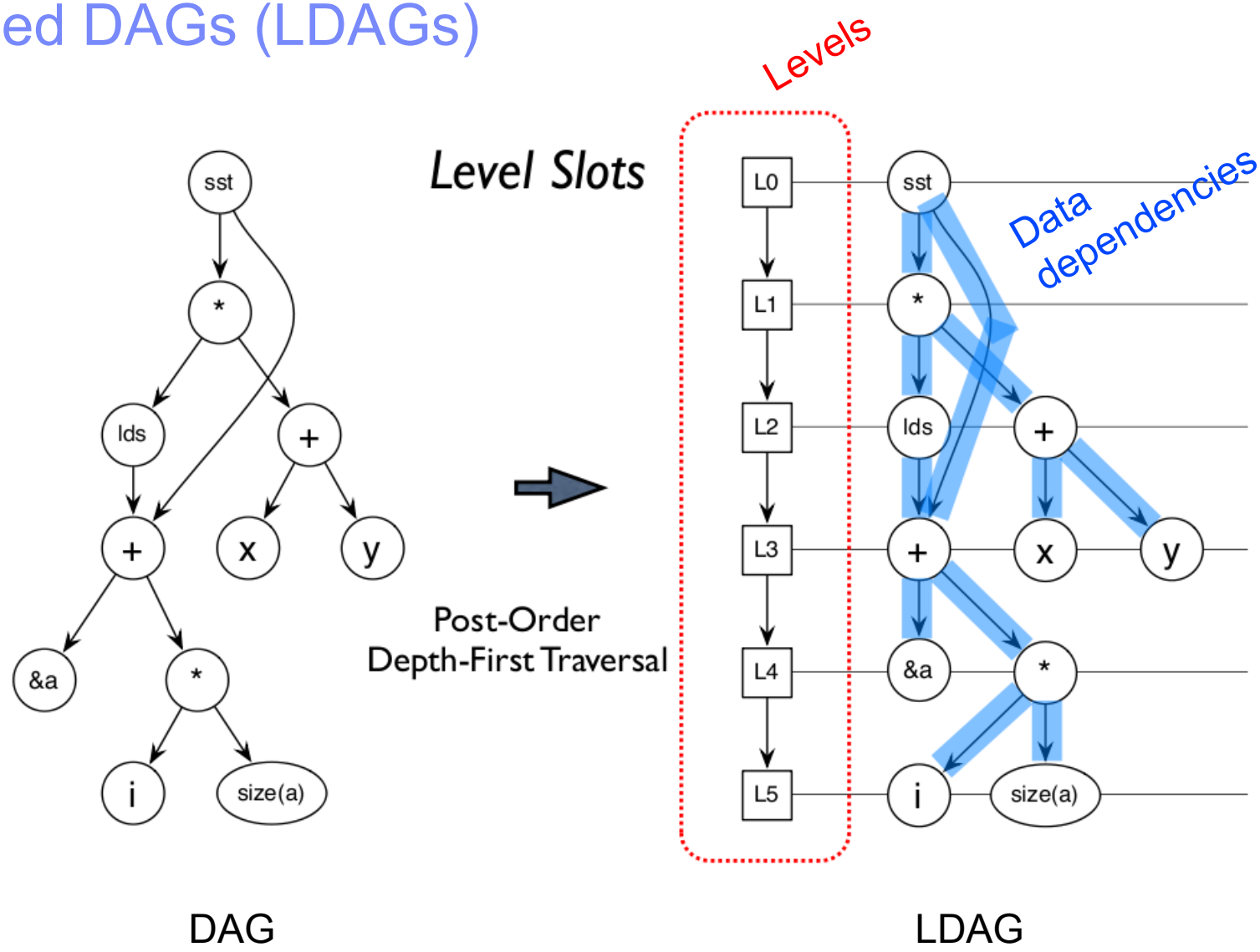
- An LDAG is the mapping of nodes to integers such that if there is an edge from *u* to *v*, then

  lev(v) = lev(u) + N

  for all *E*



Leveled DAGs (LDAGs)

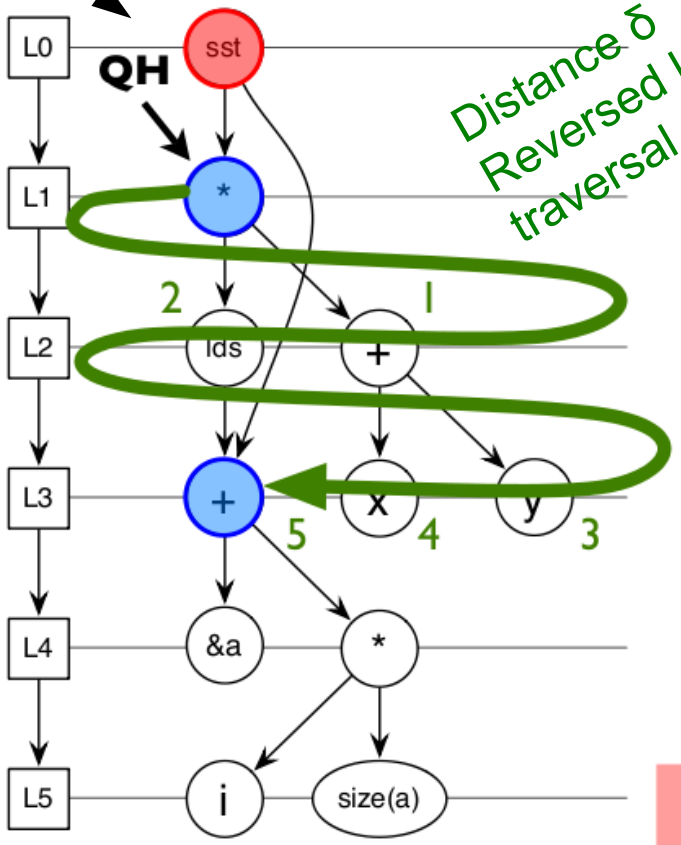


Offset calculation algorithm

$$offset(m) = \delta(qh\_pos(u), m)$$

Algorithm 2 qh_pos (LDAG w, node u)

```
1:  I ⇐ getLevel (u)
2:  for i ⇐ u.prev to I.α-node do
3:     if isOperation (i) then
4:        if isHardEdge (i.right) then
5:           v ⇐ BFS_nextnode (i.right)
6:           return v
7:        end if
8:        if isHardEdge (i.left) then
9:           v ⇐ BFS_nextnode (i.left)
10:          return v
11:       end if
12:    end if
13: end for
14: L ⇐ getNextLevel (u)
15: v ⇐ L.α-node
16: return v
```



Distance δ
Reversed level-order traversal

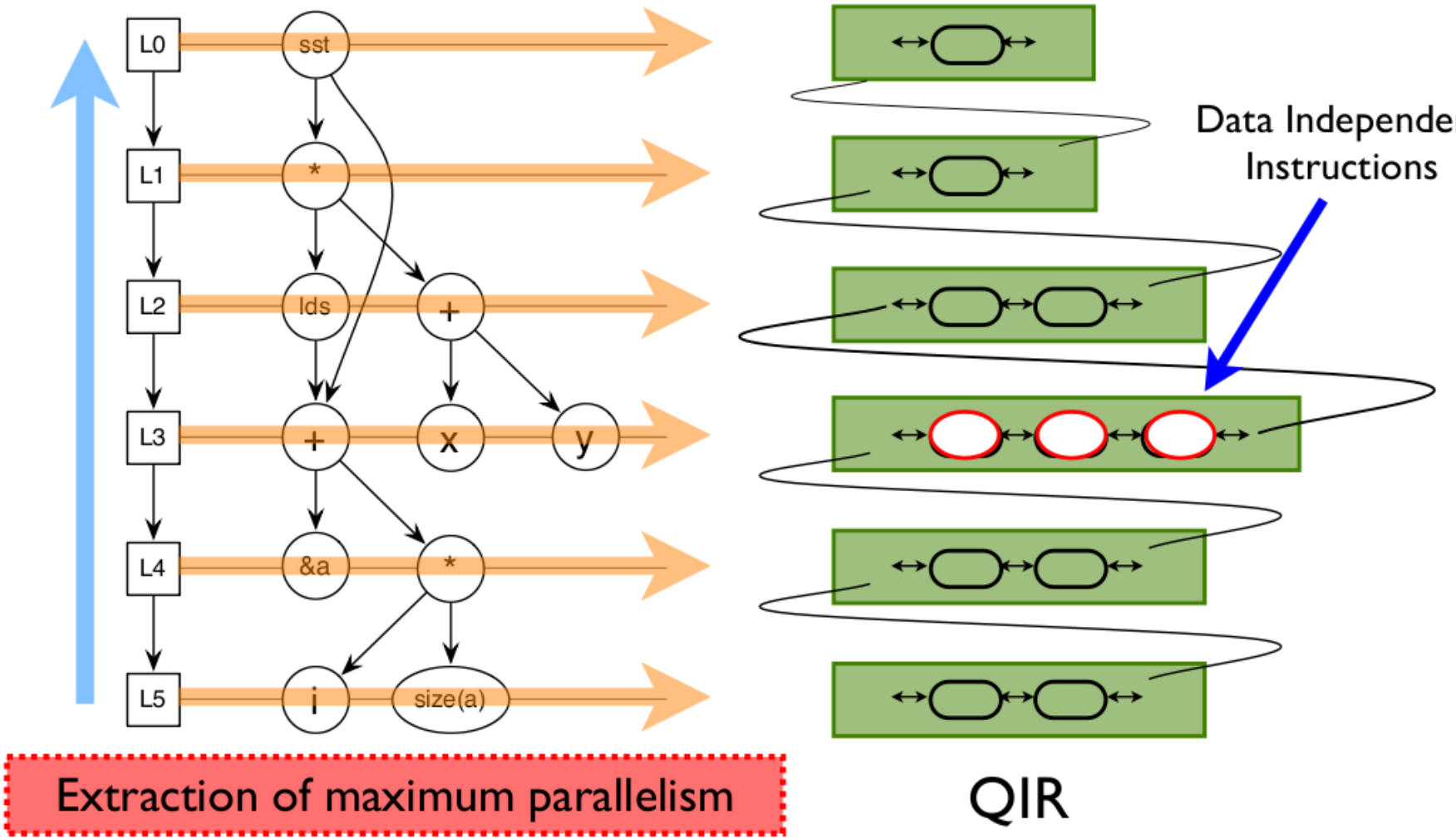
offset(m1) = 0

offset(m2) = -5

sst 0, -5

Important: any change to the DFG requires offset recomputation

LDAG to QIR (Level-Order Scheduling)



QIR to Assembly

Single operand

Annotations

```
(QMARK_LEVEL [id=0])
(PUSH_Q [iws] (LOCAL_VAR_Q: 8($fp) )
(LOADI_Q [iws] (INTEGER_Q: 1 )
(QMARK_LEVEL [id=1])
(NE_Q [iws] )
(GOTO_Q [iws] (LABEL_Q: L1 )
(TLABEL_Q [iws] (LABEL_Q: L0 )
(QMARK_LEVEL [id=2])
(PUSH_Q [iws] (LOCAL_VAR_Q: 12($fp) )
(LOADI_Q [iwu] (INTEGER_Q: 4 )
(QMARK_LEVEL [id=3])
(LOAD_ADDR_Q [iws] (LOCAL_VAR_Q: 16($fp) )
(MUL_Q [iws] ( QT, QH+0, QH+1 ))
(QMARK_LEVEL [id=4])
(ADD_Q [iws] ( QT, QH+0, QH+1 ))
(PUSH_Q [iws] (LOCAL_VAR_Q: 4($fp) )
(PUSH_Q [iws] (LOCAL_VAR_Q: 0($fp) )
(QMARK_LEVEL [id=5])
(SLOAD_Q [iws] ( QT, QH+0, QH+0 ))
(ADD_Q [iws] ( QT, QH+0, QH+1 ))
(QMARK_LEVEL [id=6])
(MUL_Q [iws] ( QT, QH+0, QH+1 ))
(QMARK_LEVEL [id=7])
(STORE_Q [iws] ( QT, QH-5, QH+0 ))
(GOTO_Q [iws] (LABEL_Q: L2 )
```

offsets

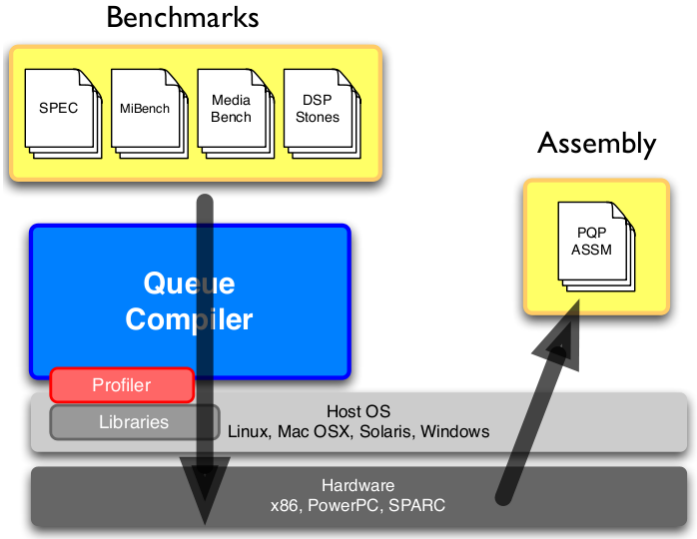
QIR (debugging mode)

PQP Assembly

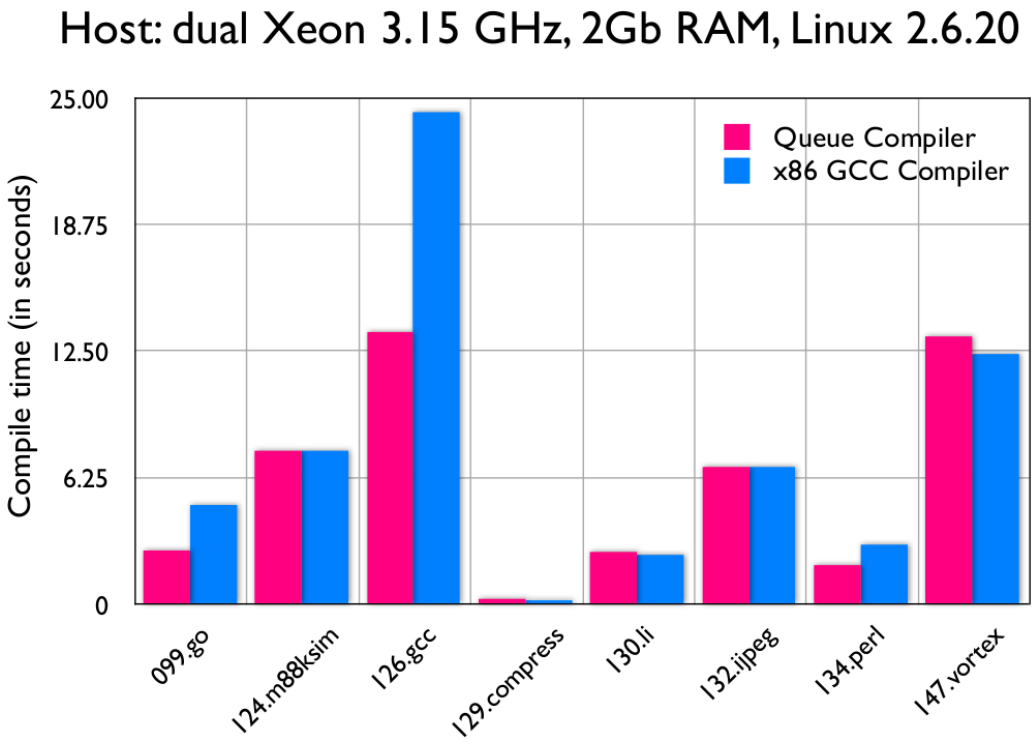
```
                Opcode   Data type   DST   SRC
L0:
                ld       iws, qt,          ($fp)8
                ldi      iws, qt,          1
                cne      iws, iws,         $cc, qh, qh+1
                bt       iws, L1,          $cc

                ld       iws, qt,          ($fp)12
                ldi      iws, qt,          4
                lea      iws, qt,          $fp, 16
                mul      iws, qt,          qh, qh+1
                add      iws, qt,          qh, qh+1
                ld       iws, qt,          ($fp)4
                ld       iws, qt,          ($fp)0
                lds      iws, qt,          qh
                add      iws, qt,          qh, qh+1
                mul      iws, qt,          qh, qh+1
                sst      iws, qt,          qh-5, qh
                j        iws,              L2
```

Queue compiler

- 4 years of research and development
- 15,000 lines of code (back-end)
- Supports all flavors of queue computing
- Compiles any program (including itself)



Cross compiler configuraiton

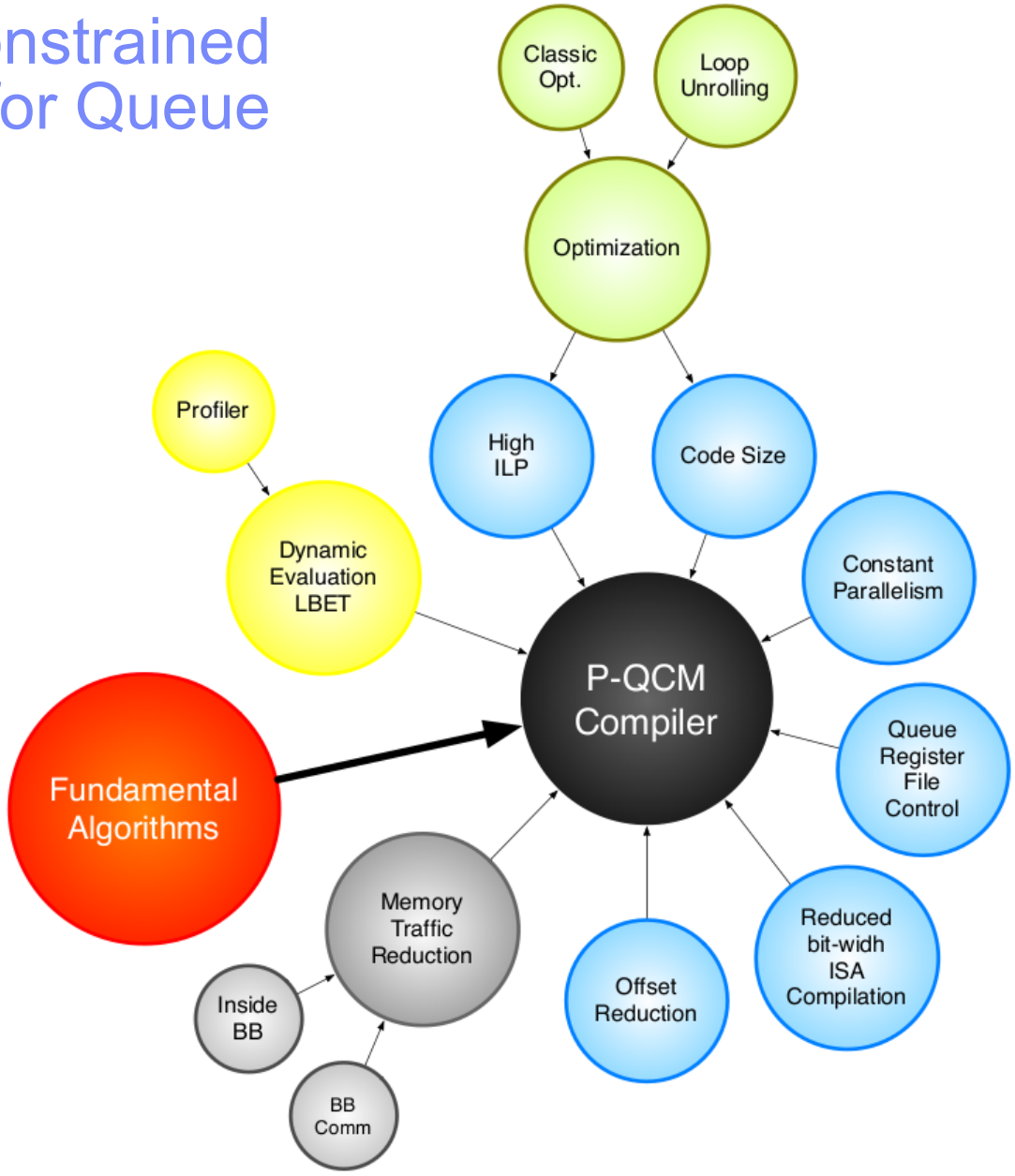


Constrained-Compilation

Targeting actual hardware

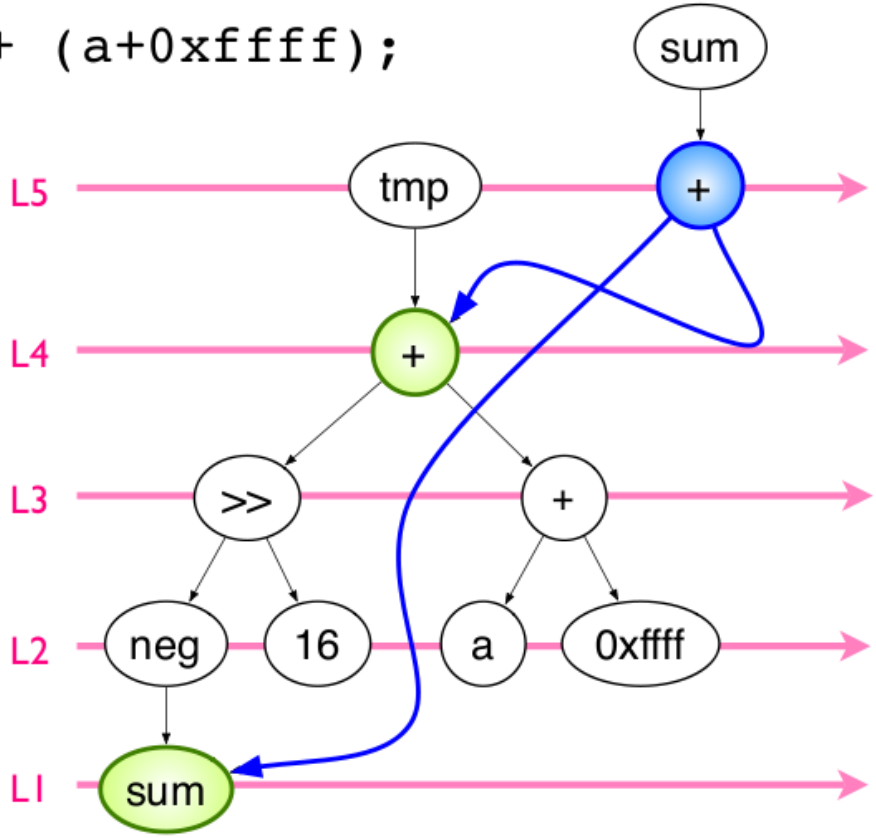
Part III

Topics on Constrained Compilation for Queue Machines



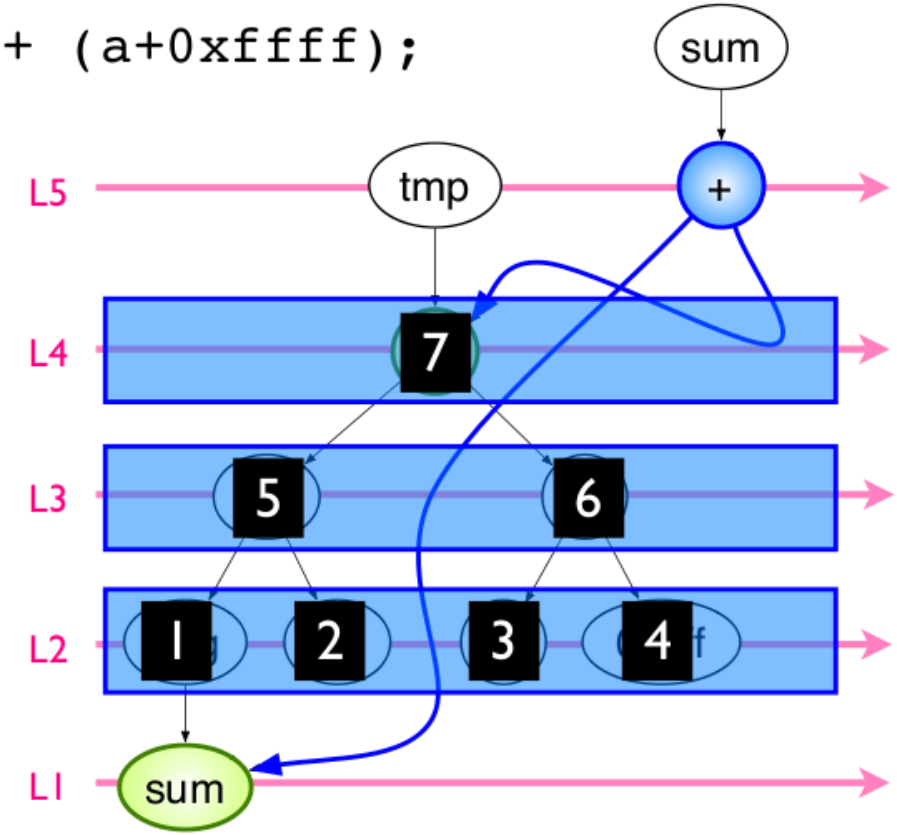
Queue Utilization

```
tmp = (-sum >> 16) + (a+0xffff);
sum = sum + tmp;
```

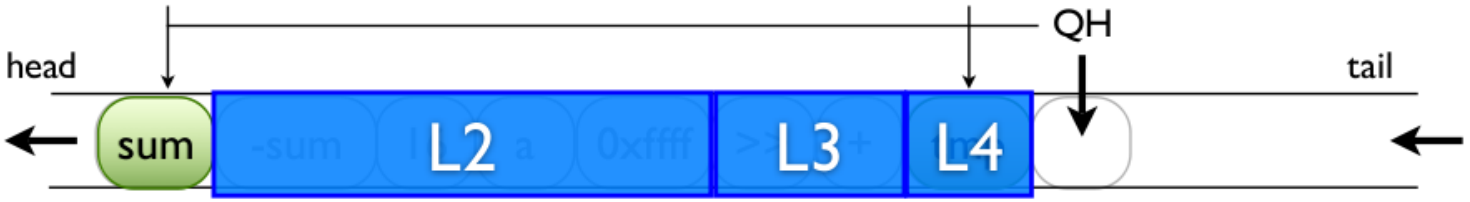


Queue Utilization

```
tmp = (-sum >> 16) + (a+0xffff);
sum = sum + tmp;
```

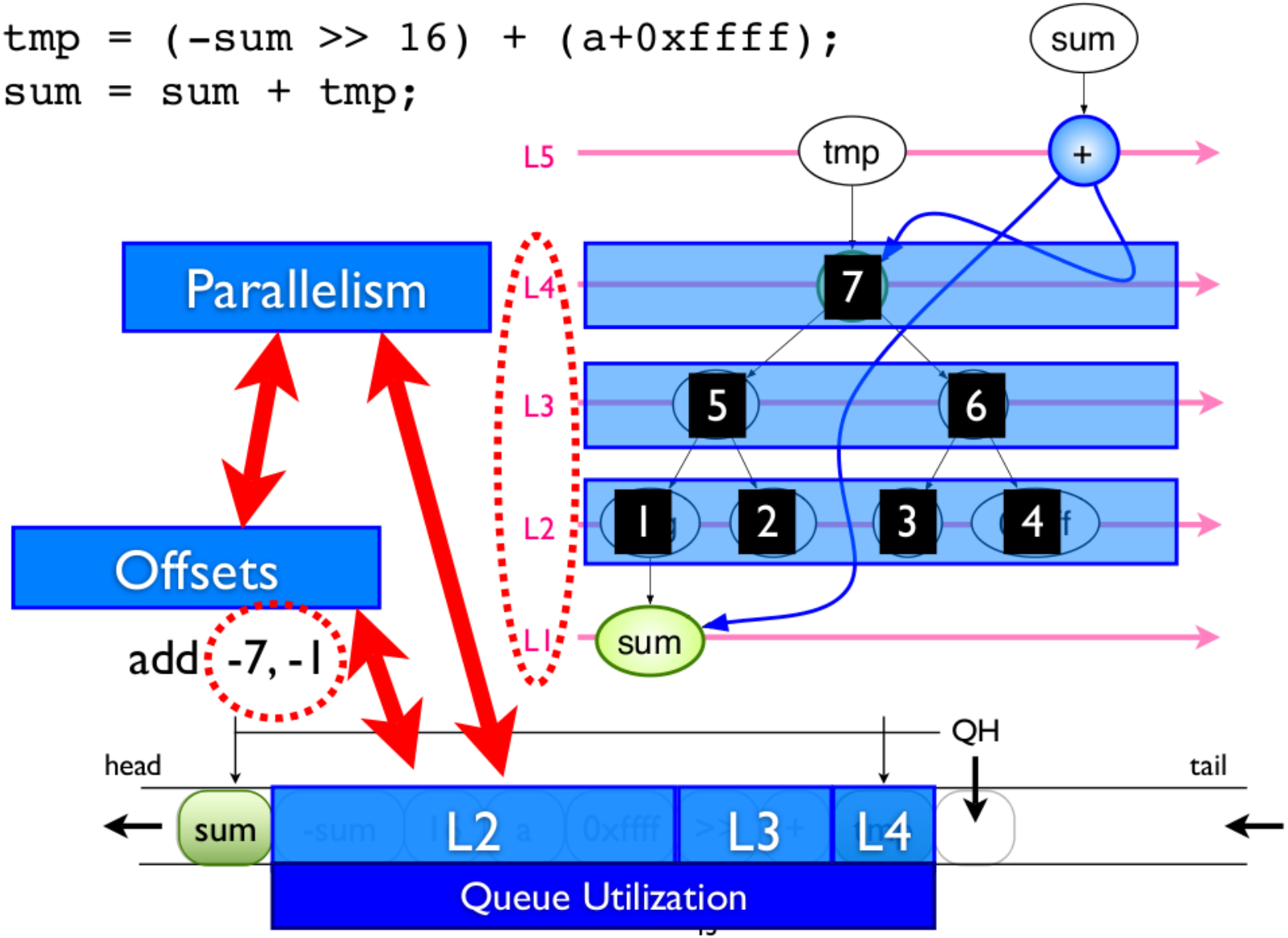


add -7, -1

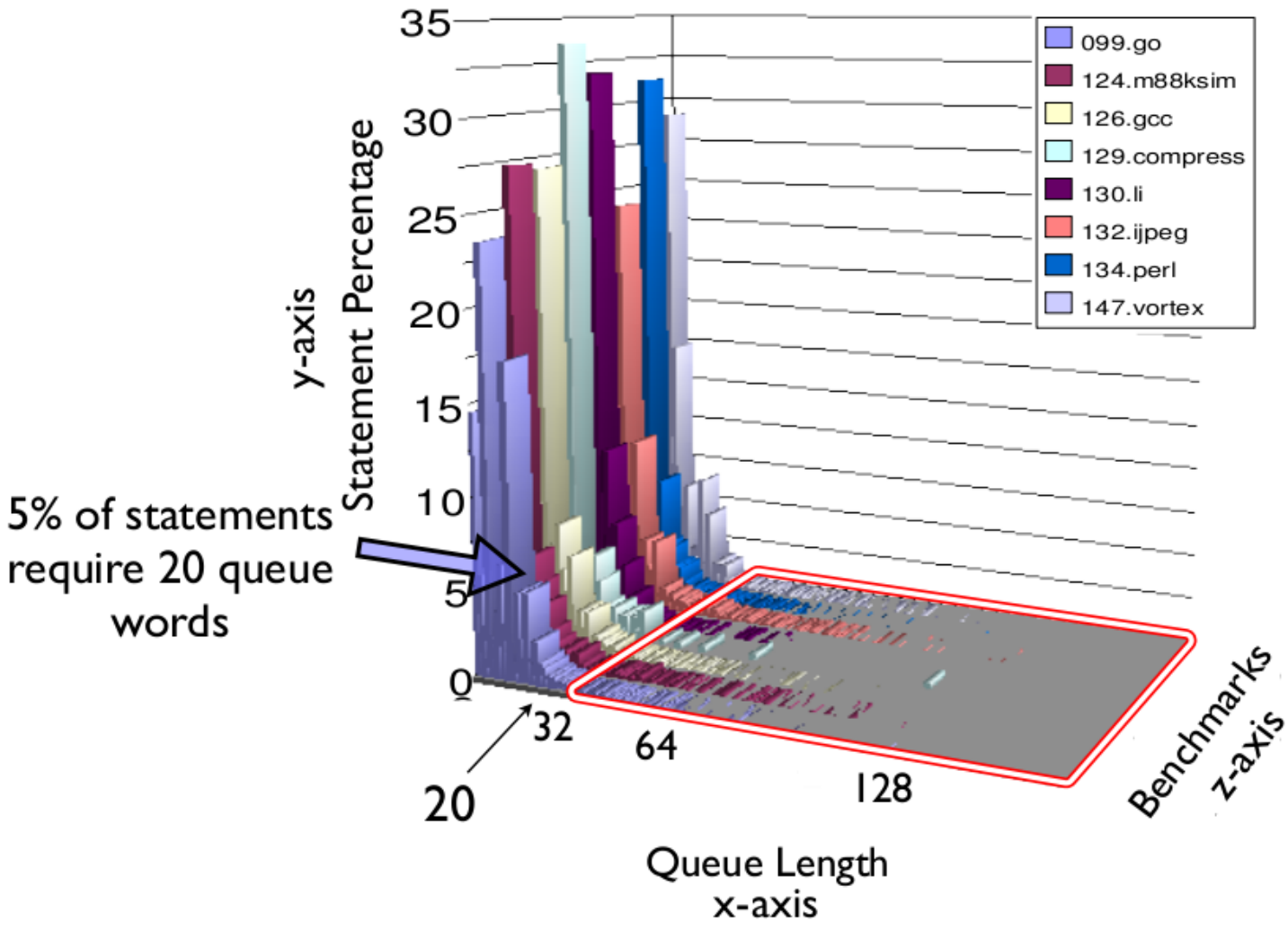


Queue Utilization

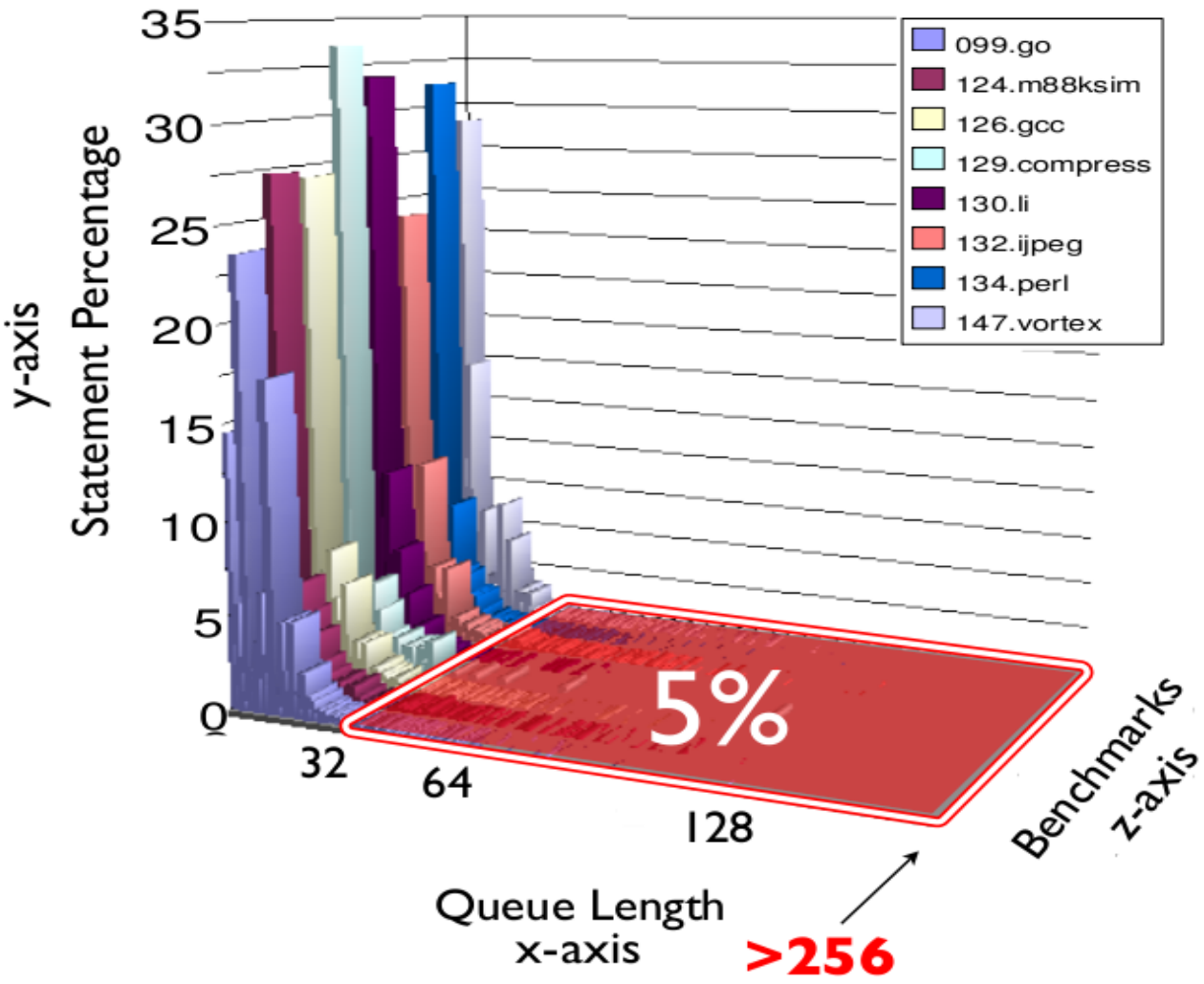
```
tmp = (-sum >> 16) + (a+0xffff);
sum = sum + tmp;
```



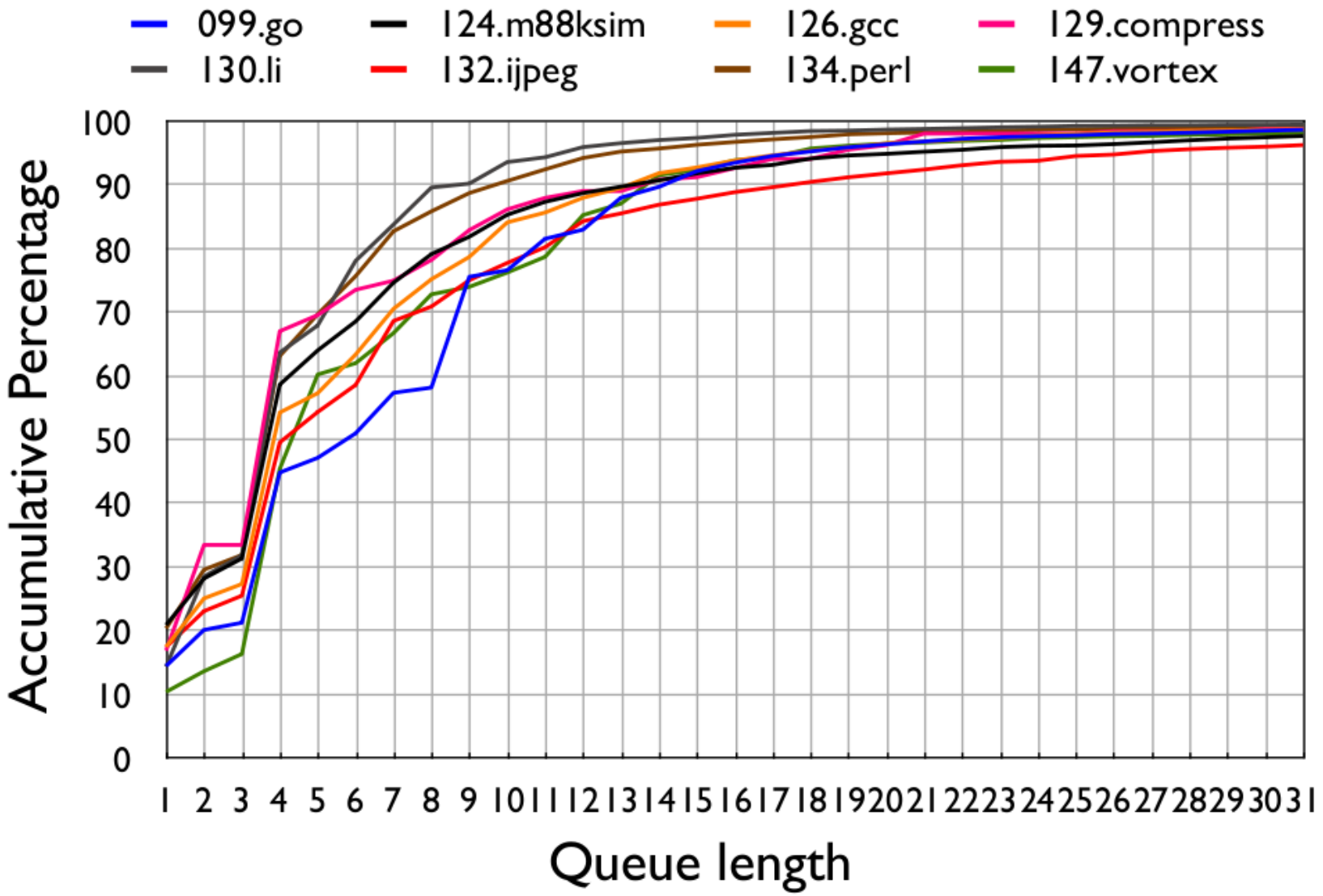
Queue Utilization of SPEC95



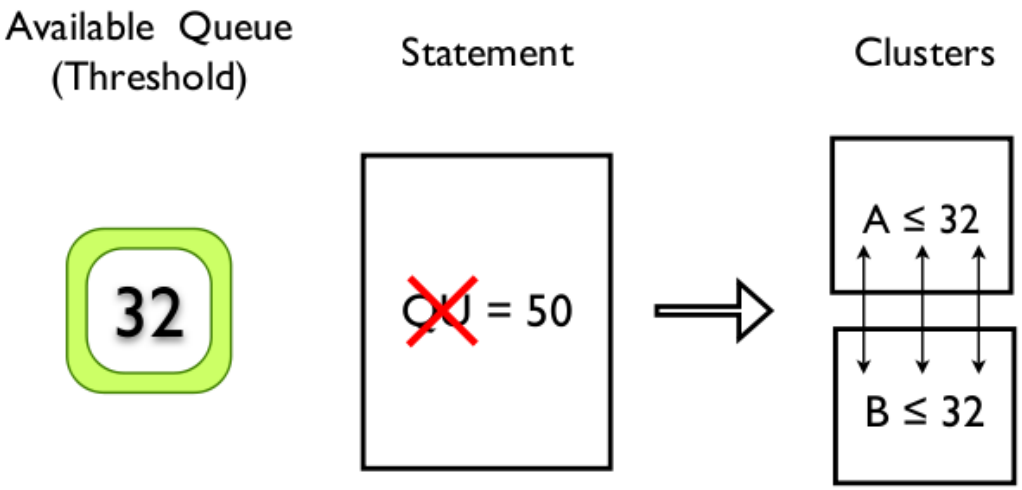
Queue Utilization of SPEC95



In other words



Queue-length optimization algorithm



- Break the queue hungry statements into fragments (clusters) that require equal or less amount of queue than available in the hardware.
  - Clusters must comply with the semantics of queue computation model.
  - Generate the fewest clusters.
  - Keep communication code minimal.

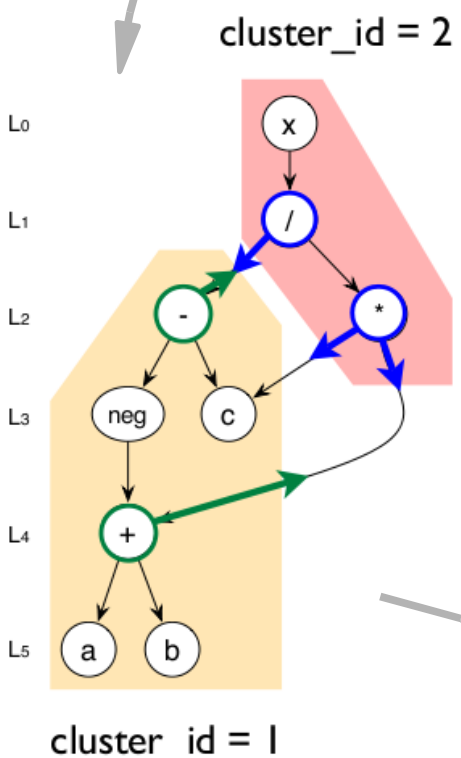
Algorithm on LDAGs

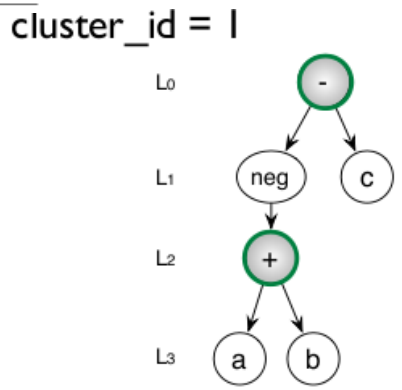
Algorithm 1 labelize (LDAG W)

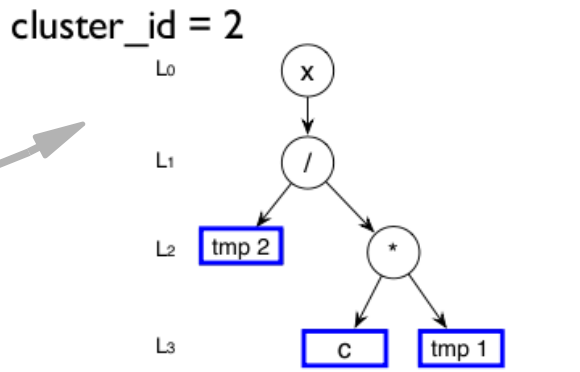
```
Require: global Threshold
Require: global cluster_id = 0
 1: root ← W's root node
 2: if SubTree_Width (root) > Threshold then
 3:    lhs ← labelize (root.lhs)
 4:    if isBinary (root) then
 5:       if SubTree_Width (root.rhs) > Threshold then
 6:          rhs ← labelize (root.rhs)
 7:          root ← Assign_ID_to_node (rhs.id)
 8:          return root
 9:       else
10:          root.rhs ← Assign_ID_to_subtree
             (cluster_id++);
11:          root ← Assign_ID_to_node (root.rhs.id);
12:          return root
13:       end if
14:    end if
15: else
16:    root ← Assign_ID_to_subtree (cluster_id++);
17:    return root
18: end if
```

Algorithm 2 clusterize (node u, LDAG W)

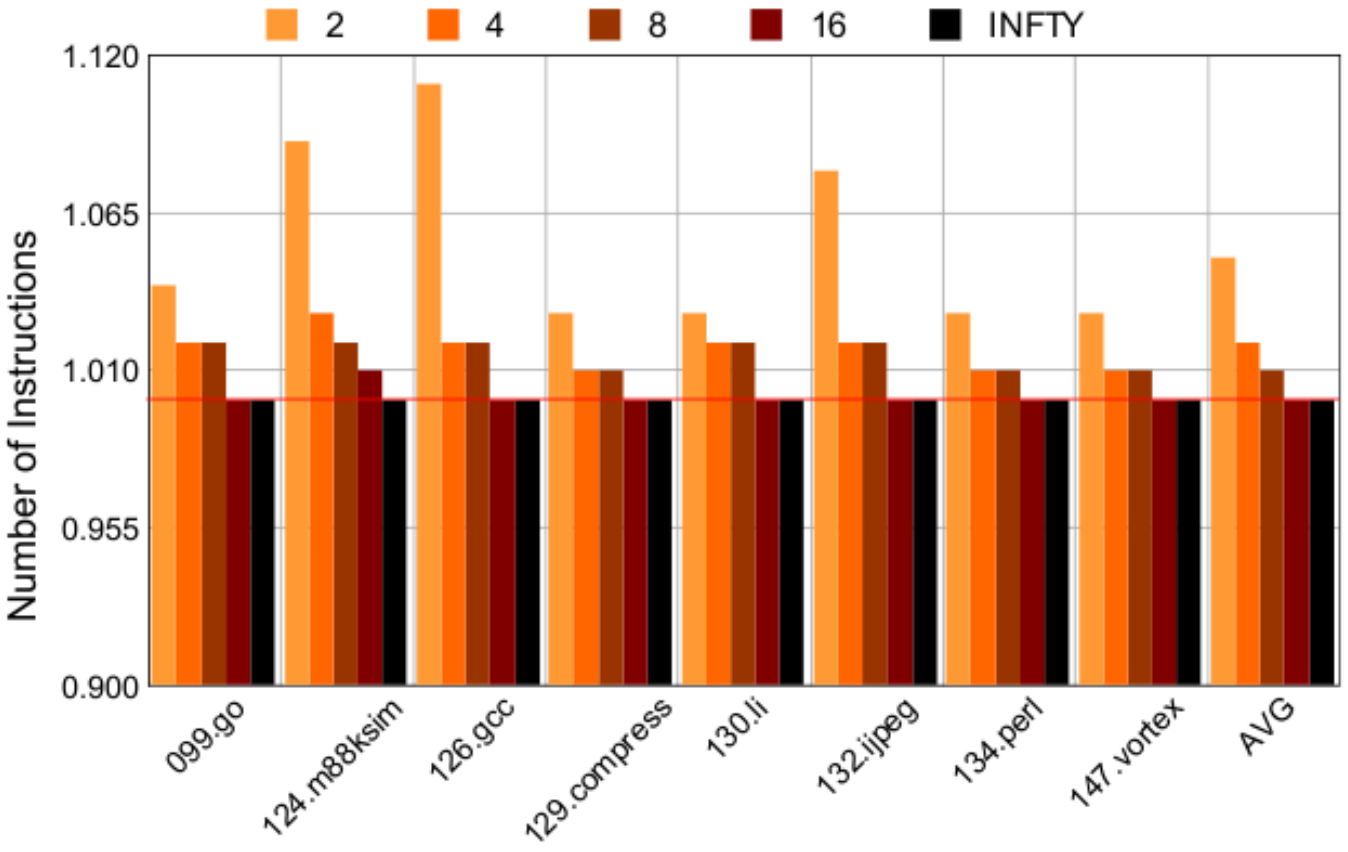
```
Require: An empty cluster set C of N elements
 1: /* Traverse as a DAG */
 2: if AlreadyVisited (u) then
 3:    return NIL
 4: end if
 5: /* Action 1: add to corresponding cluster */
 6: ClusterSet_Add (C, ID(u), u)
 7: /* Action 2: generate reloads */
 8: for all children v of u do
 9:    if ID(v) ≠ ID(u) then
10:       GenReload (C, ID(u), v)
11:    else if isViolatingSoftEdge (u, v) then
12:       GenReload (C, ID(u), v)
13:    else
14:       /* Post-Order traversal */
15:       clusterize (v, W)
16:    end if
17: end for
18: /* Action 3: generate spills */
19: if Parents (u) > 1 AND isSubexpression (u) then
20:    GenSpill (C, ID(u), u)
21: end if
22: /* Mark visited and return */
23: MarkVisited (u)
24: return u
```







Controlling Queue utilization



1-offset PQCM Instruction set

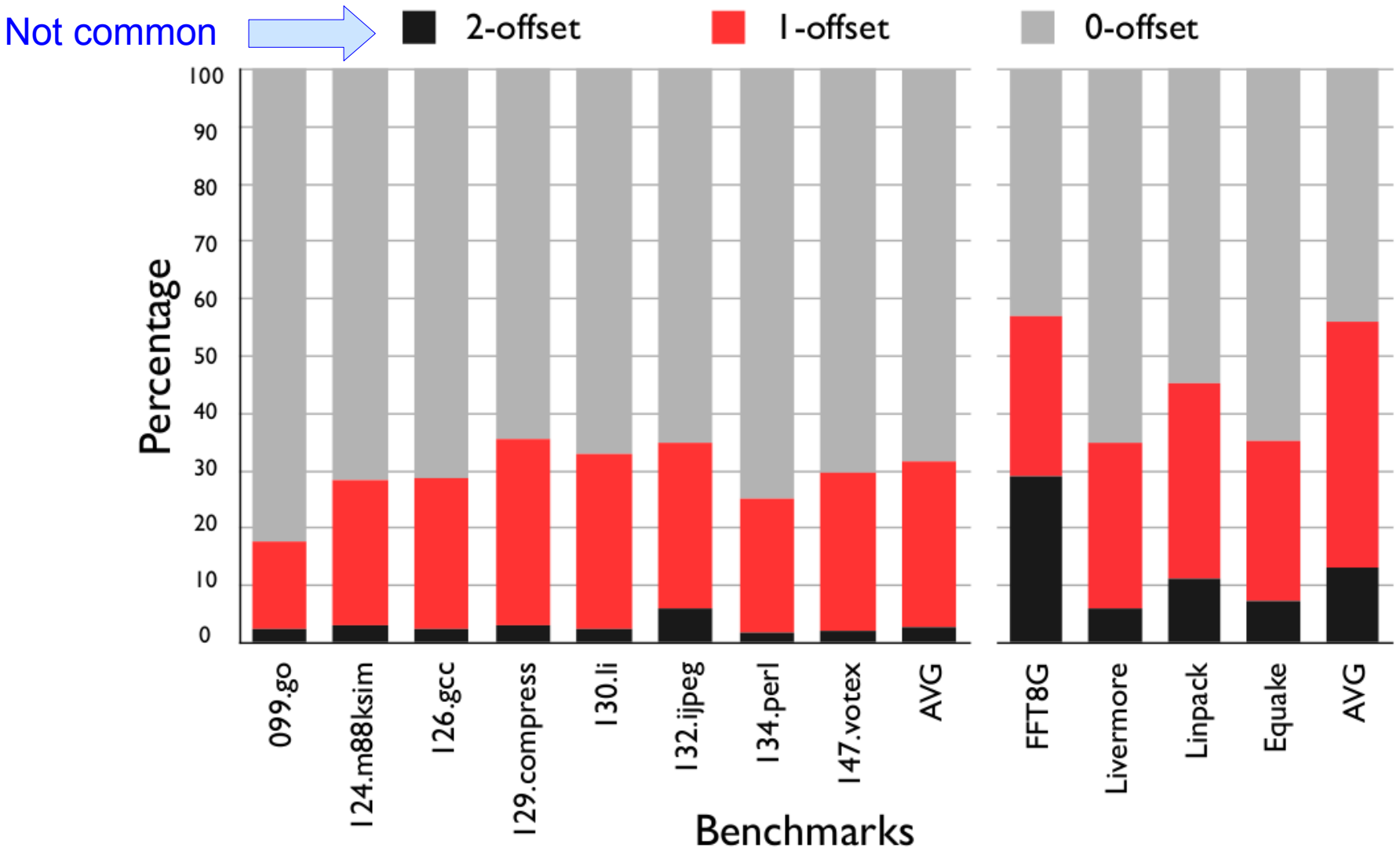
- Forget about 2-offsets !
  - 2/100 instructions.
- We can optimize the rare occurrences of 2-offset instructions.
- Gives more reach to offsets.
- Potentially reduces the bit requirements of the instruction set

```
add -1, -35
```

Rare instruction < 2%

Not reachable by a 16-bit instructions with 2^5 bits per offset

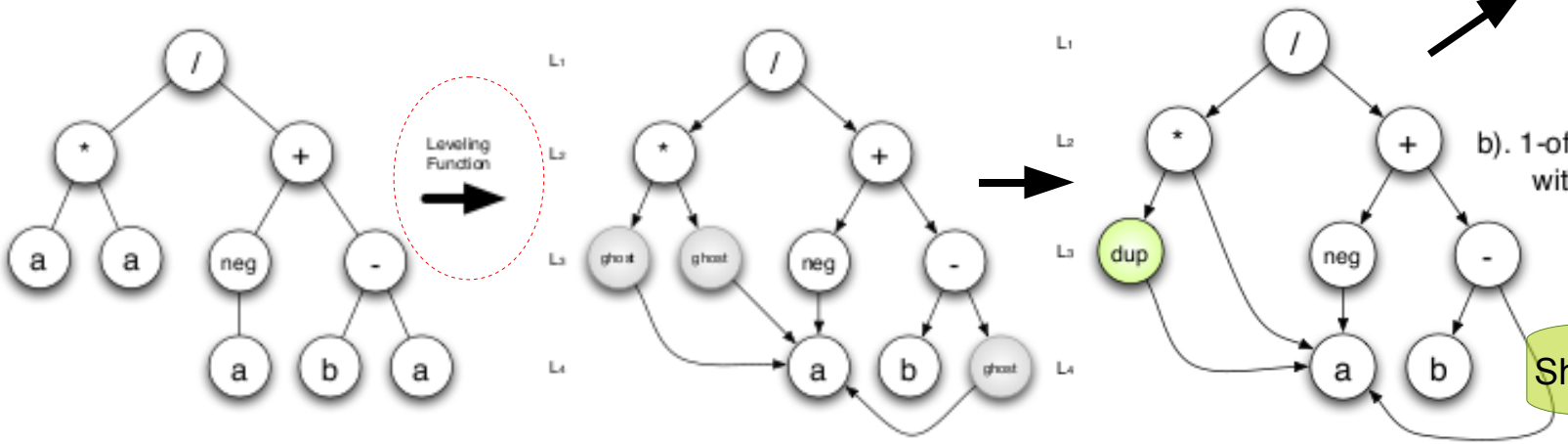
Offset references: the other problem

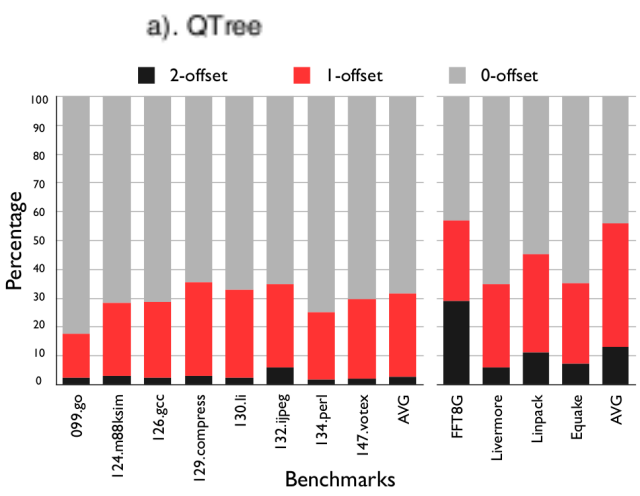


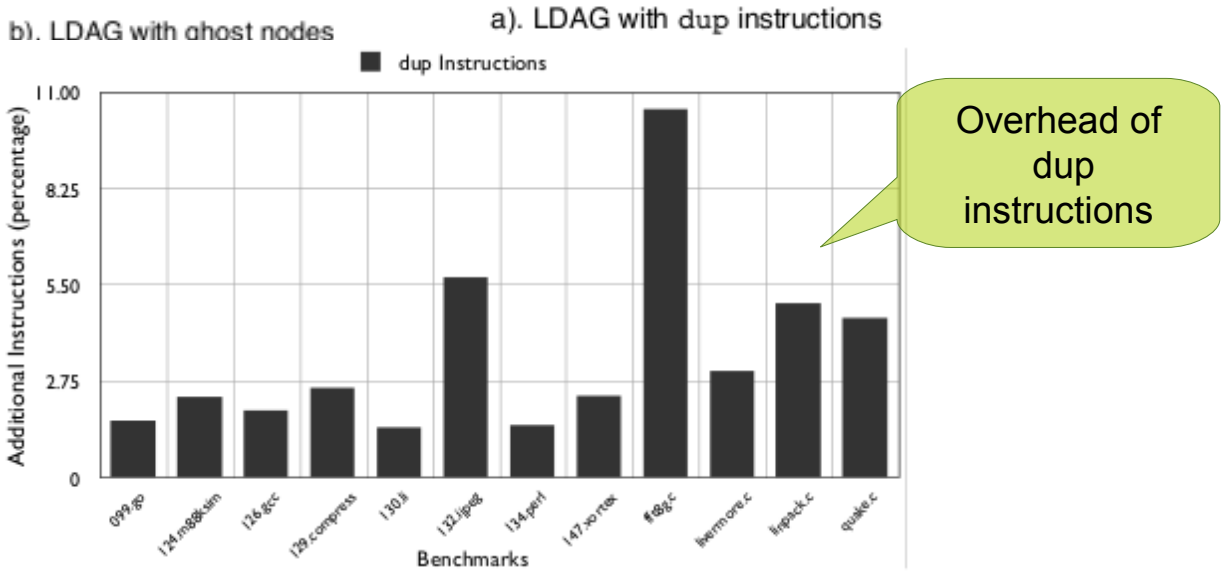
1-offset code generation

- Insert dummy nodes and fill with duplicate instructions.

```
ld      a
ld      b
dup     0
neg     0
sub    -1
mul    -2
add     0
div     0
st      x
```

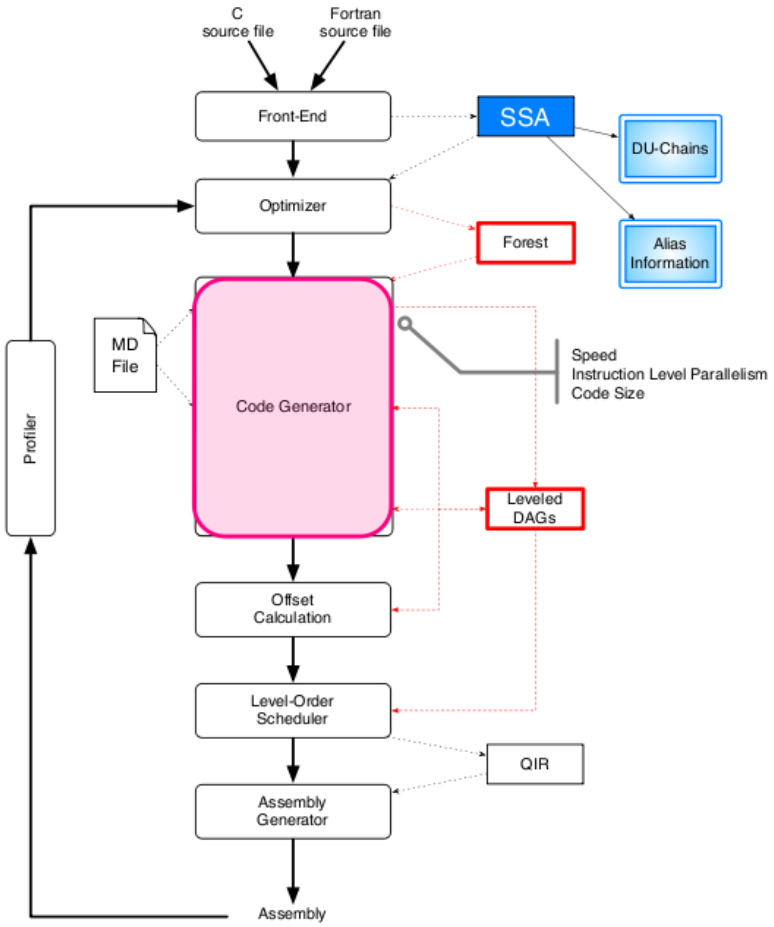






Global optimizer

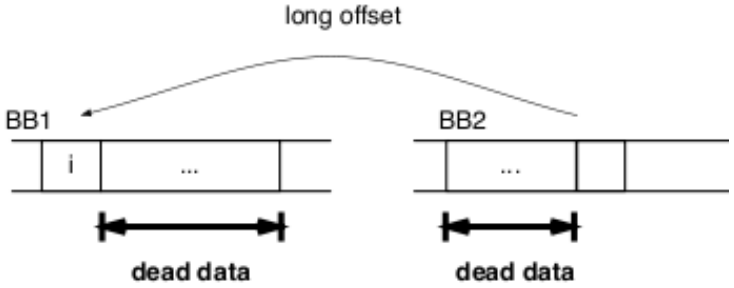
- Get parallelism across basic blocks
- **Solve the fundamental problems**
- Evaluate its effects on programs
  - Offsets
  - Queue utilization
  - Reduce memory operations
  - etc.



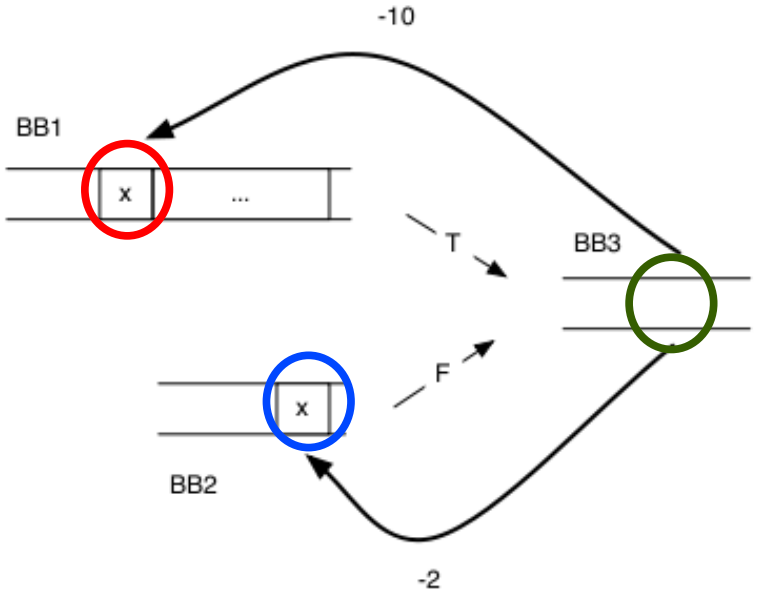
Offset inconsistency problem

- Occurs when the execution unit is across basic blocks.

```
if (true) {
   ...
   x = 1;
   produce data;
}
else {
   ...
   x = (a<<b)*c/10;
}
...
return x+1;
```



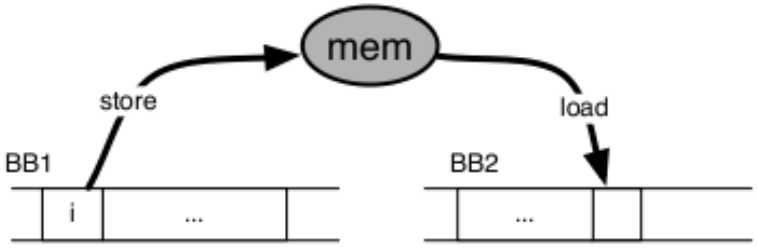
(a). Dead queue elements across basic-blocks



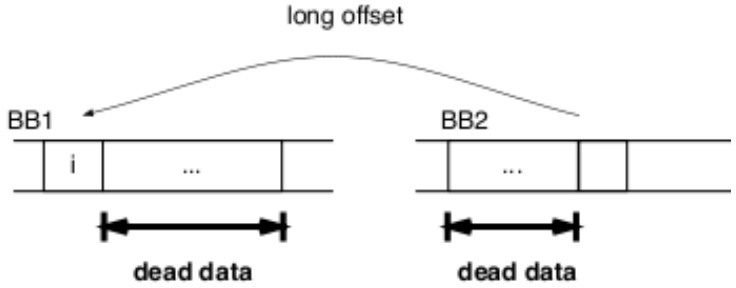
(b). Offset inconsistency problem

Offset inconsistency problem

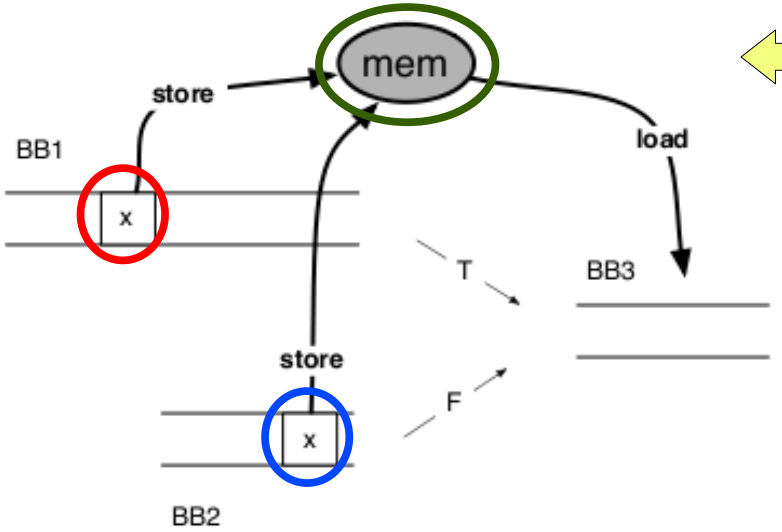
- Occurs when the execution unit is across basic blocks.



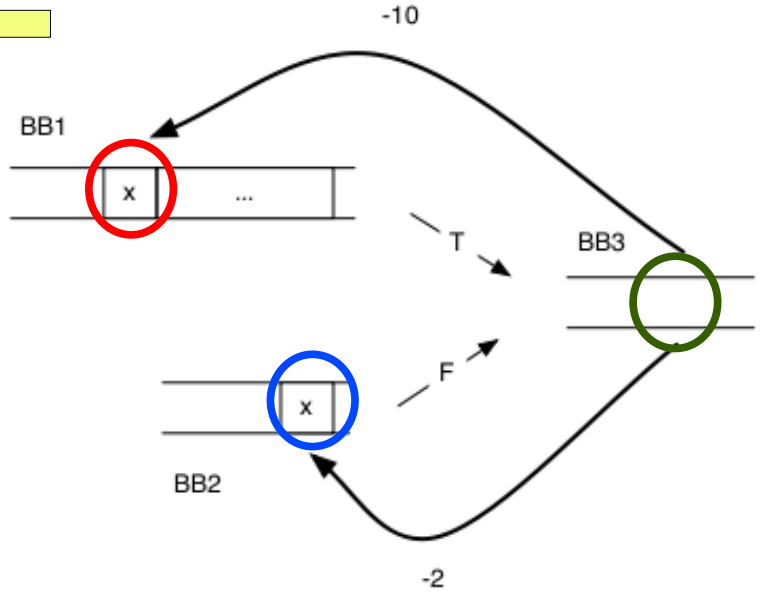
(a). Live variable communication across basic blocks



(a). Dead queue elements across basic-blocks



(b). Offset inconsistency solved by a known memory location

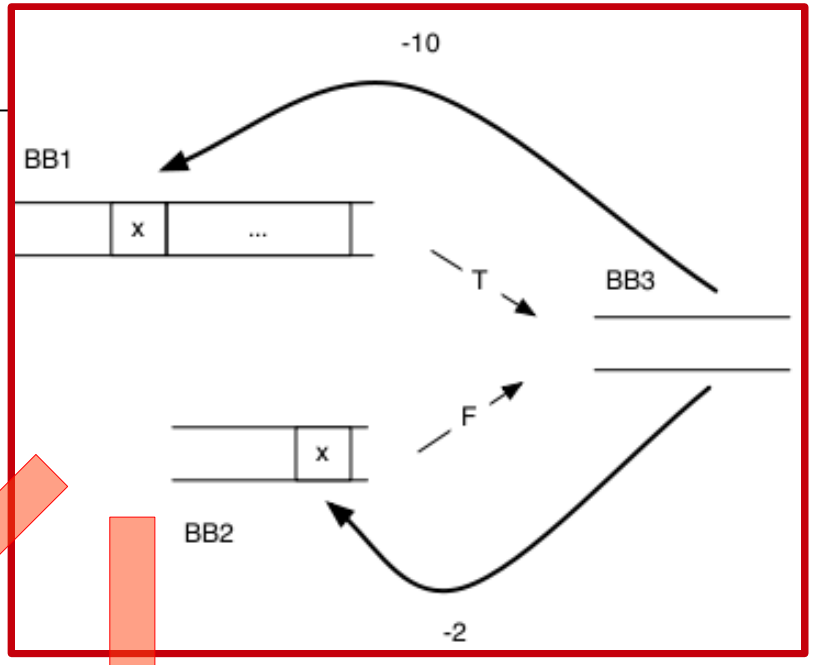


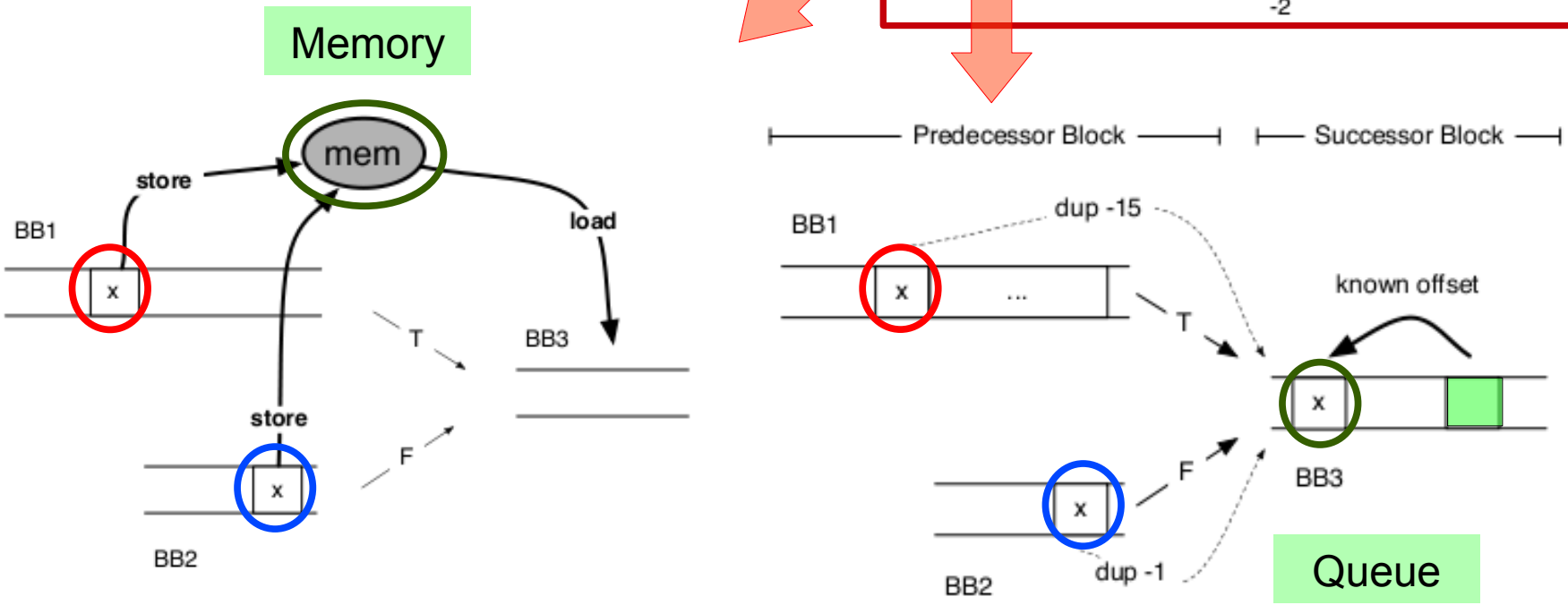
(b). Offset inconsistency problem

Naive solution: Memory sync.

Offset inconsistency problem

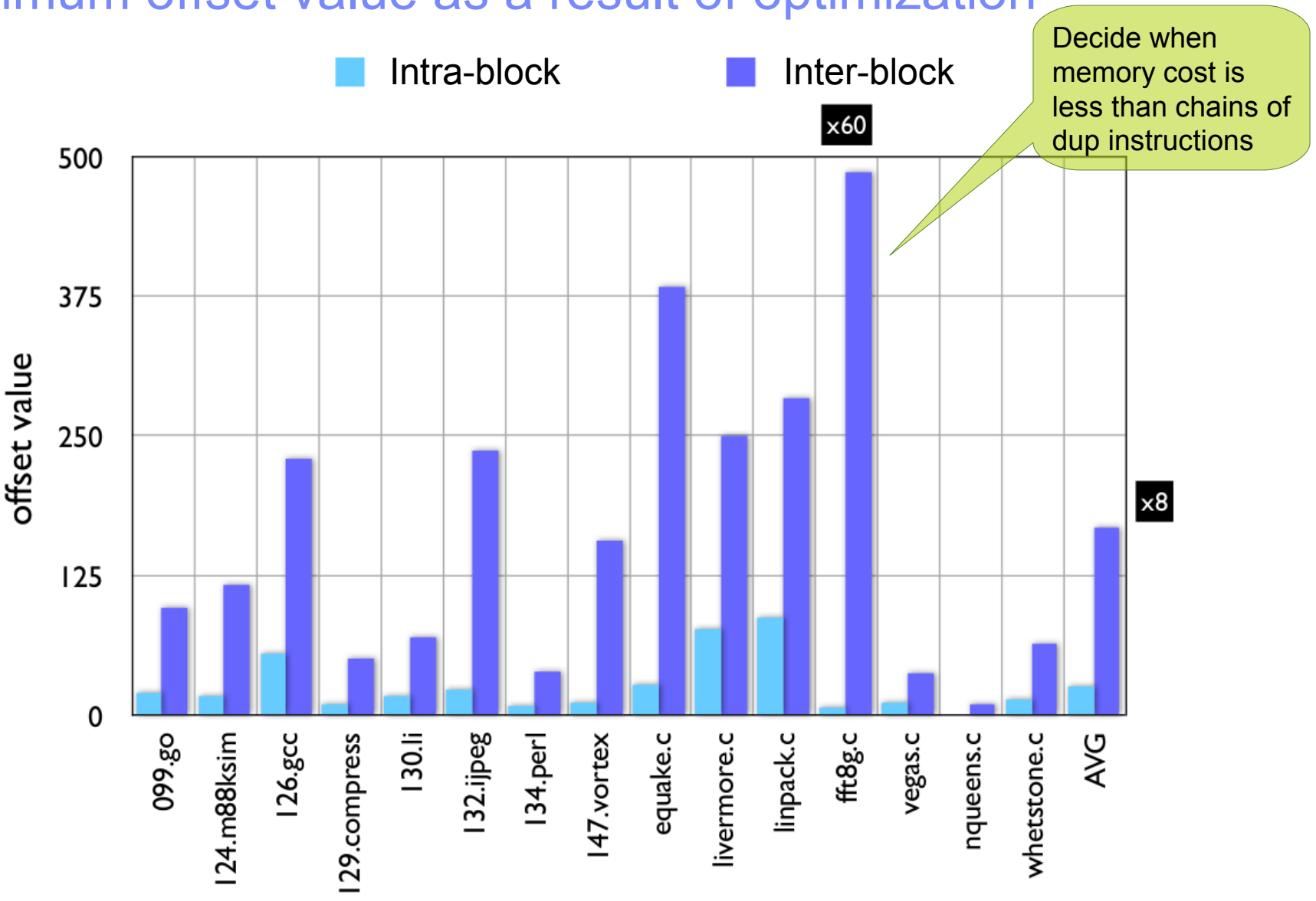
- Analogous to Live-Range Splitting



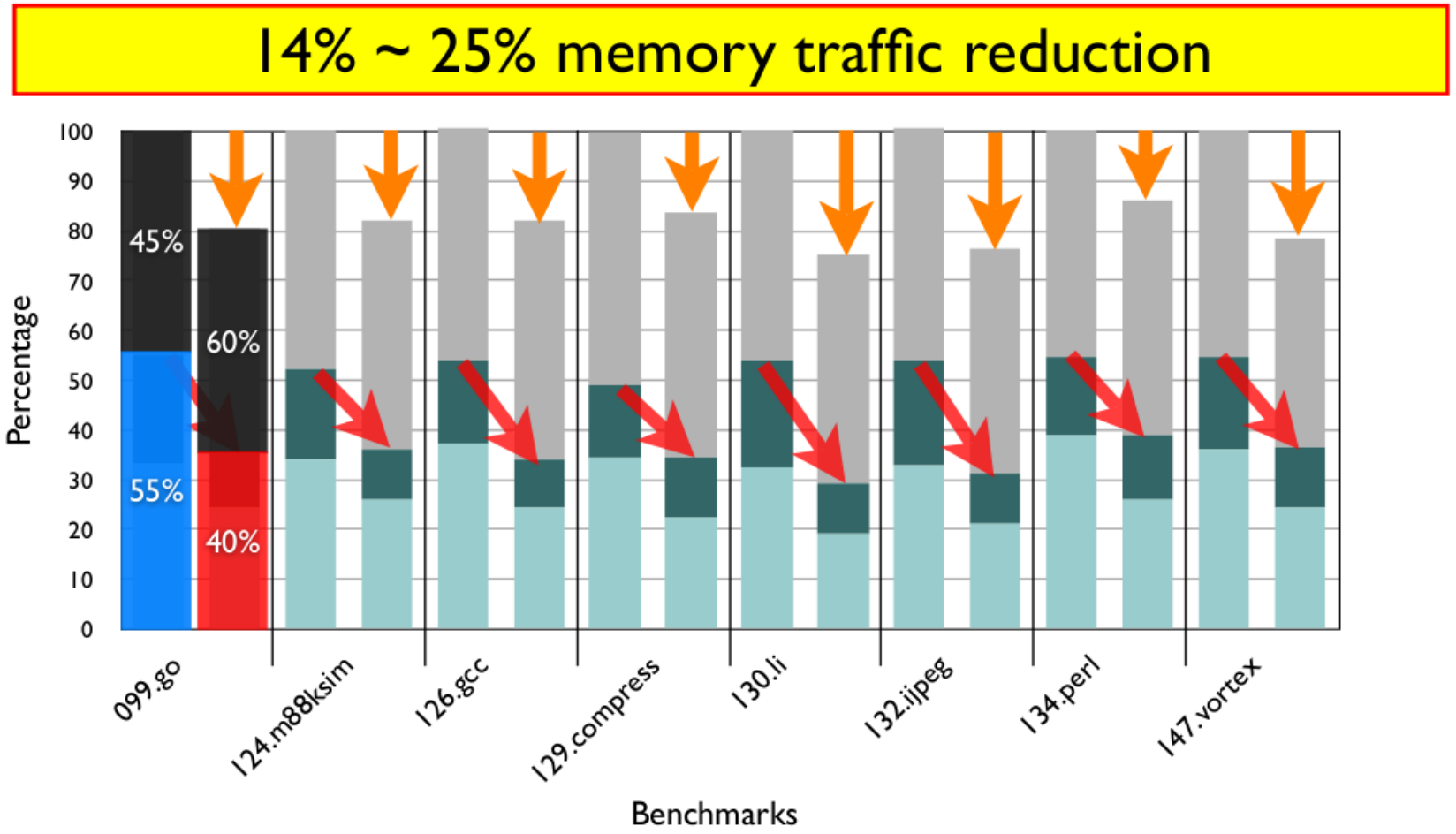


(b). Offset inconsistency solved by a known memory location

Maximum offset value as a result of optimization



Memory traffic reduction



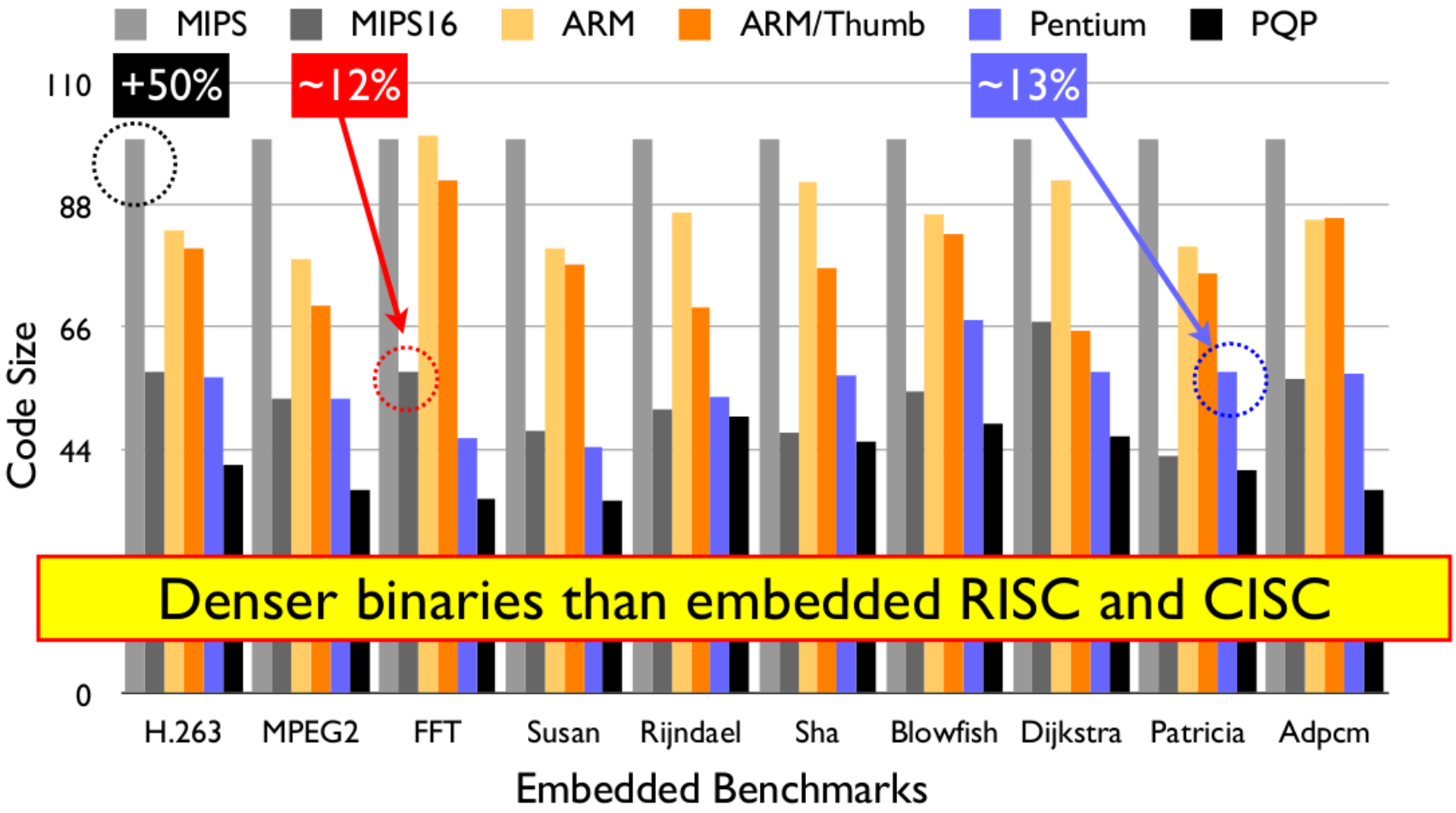
Status of the project

- Queue Compiler
  - Completed
  - Compiles any program
  - +10 papers published
- Virtual Machine
  - Completed
  - Works quite well with queue compiler generated code
- Hardware
  - No cycle-accurate simulator, unfortunately
  - FPGA implementations work but only with toy benchmarks
  - +5 papers published
- Only 2 active researchers working on queue machines in the world
  - Arquimedes Canedo, IBM Research - Tokyo
  - Prof. Ben Abderazek, University of Aizu
  - *Prof. Masahiro Sowa, University of Electro-Communications.

If I had to do it again...

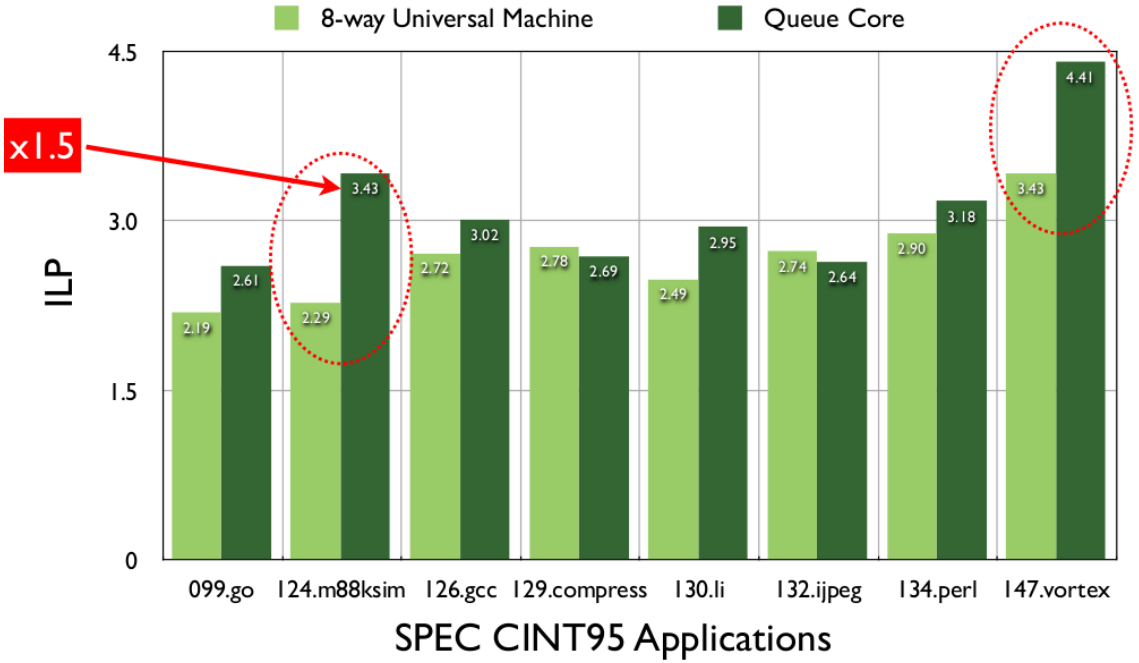
- Hardware (Priority)
  - Explore the hardware aspects of queue machines
  - Develop a robust cycle-accurate simulator
  - If it looks promising...
  - Find an application and develop the queue processor
  - As building blocks for many-core architectures
  - Explore streaming capabilities (Signal processor)
- Compiler
  - Optimize for a generic architecture, then generate queue code
  - Fine tune the instruction scheduling decisions.
  - Queue register allocation
- Software
  - As a virtual machine

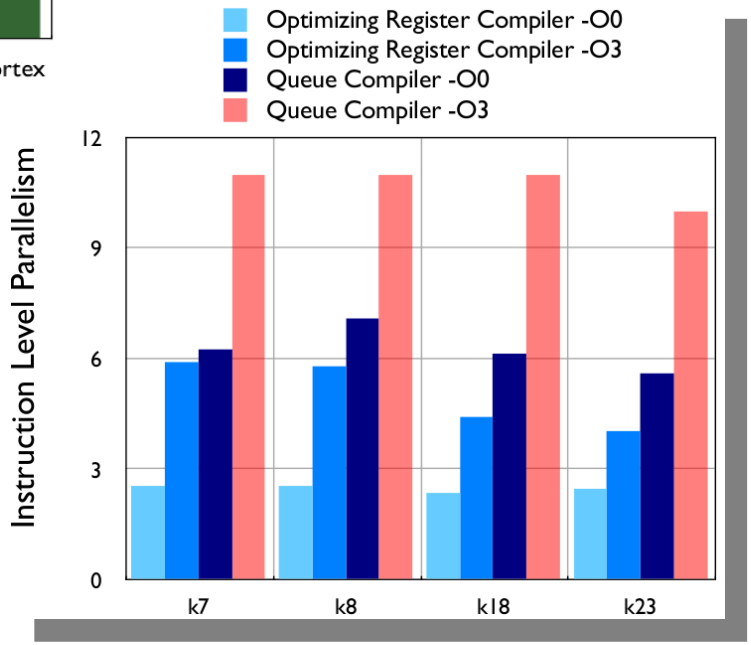
Very attractive for embedded applications



* Size of the text segment.

Very attractive for parallel processing





Thank you