CHAIR FOR COMPILER CONSTRUCTION

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Compiler Construction: Backend

Prof. Dr.-Ing. Jeronimo Castrillon
TU Dresden – Cfaed

WS 2022

Chair for Compiler Construction
Helmholtzstrasse 18
jeronimo.castrillon@tu-dresden.de

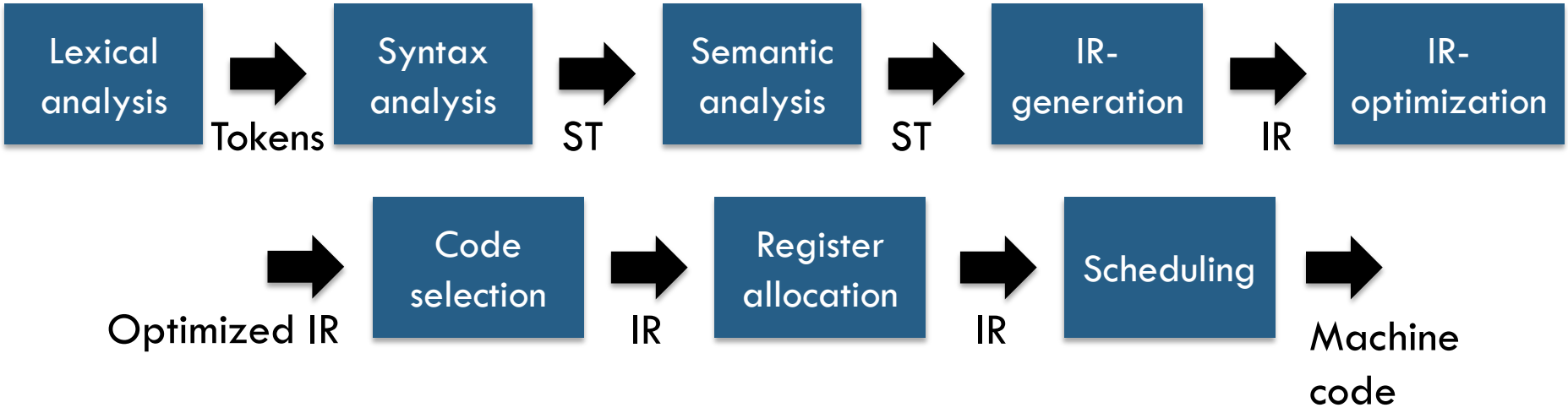
Lecture overview – Backend

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

8. Target architectures

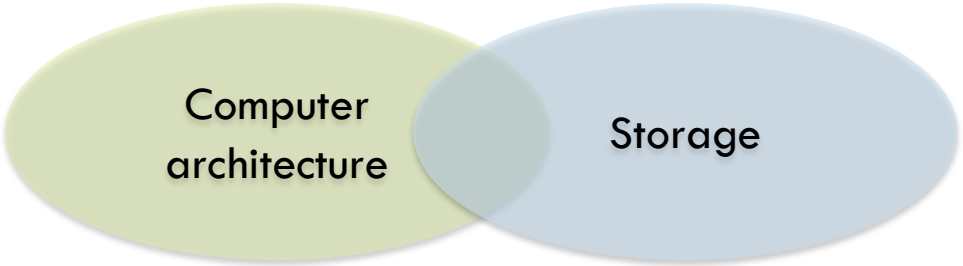
- Introduction
- CISC
- RISC
- Superscalar
- DSP
- VLIW

Why look at computer architecture?



- Backend: target specific optimizations
  - Need to understand challenges posed by architectures
  - Need recognized architecture classes and their requirements for code optimization

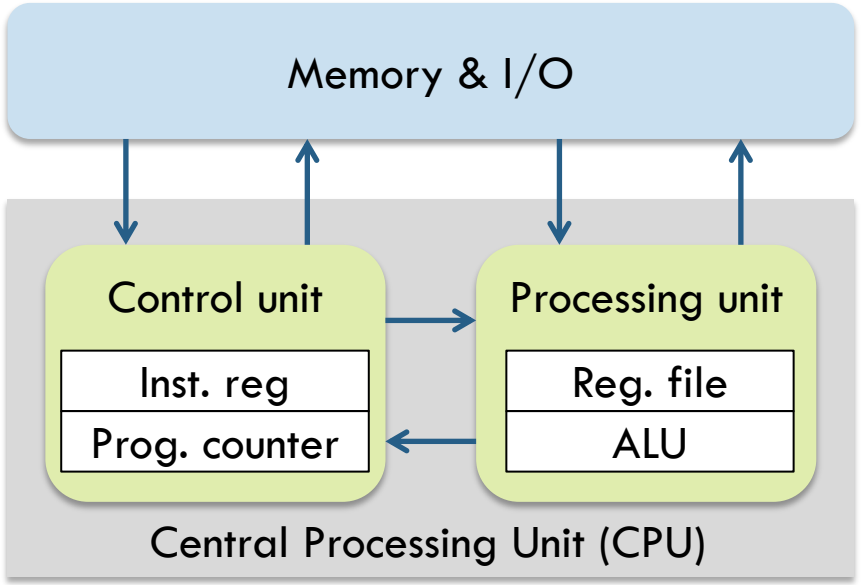
Compiler's coarse architecture view



- Computer architecture
  - Instruction-set architecture (ISA)
    - Programming interface – instructions available, their operations and their side-effects
    - Usually one ISA has different implementations (compatibility)
  - Micro-architecture: Implementation of an ISA, e.g., cache sizes, pipeline, …
- Storage: Registers, caches, disks, …

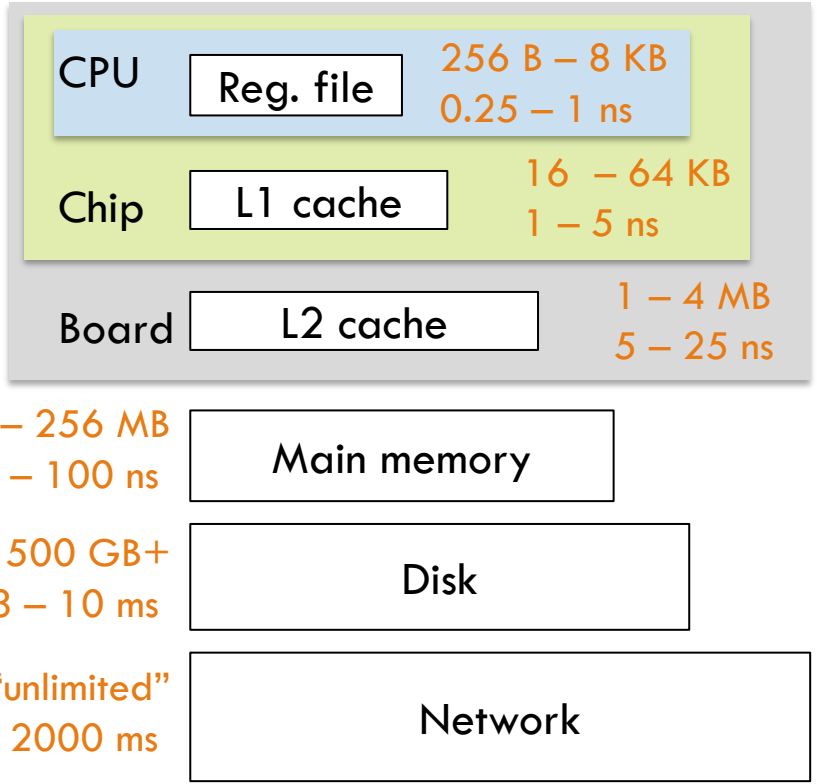
Computer architecture: Von Neumann

- The program is a sequence of instructions in memory
  - Execution controlled by control unit
  - Instruction register (IR) holds current instruction
  - Program counter (PC) points to next instruction in memory
  - Computation in data-path (processing unit)
- Von Neumann: Instructions and data stored in same memory (as opposed to Harvard)



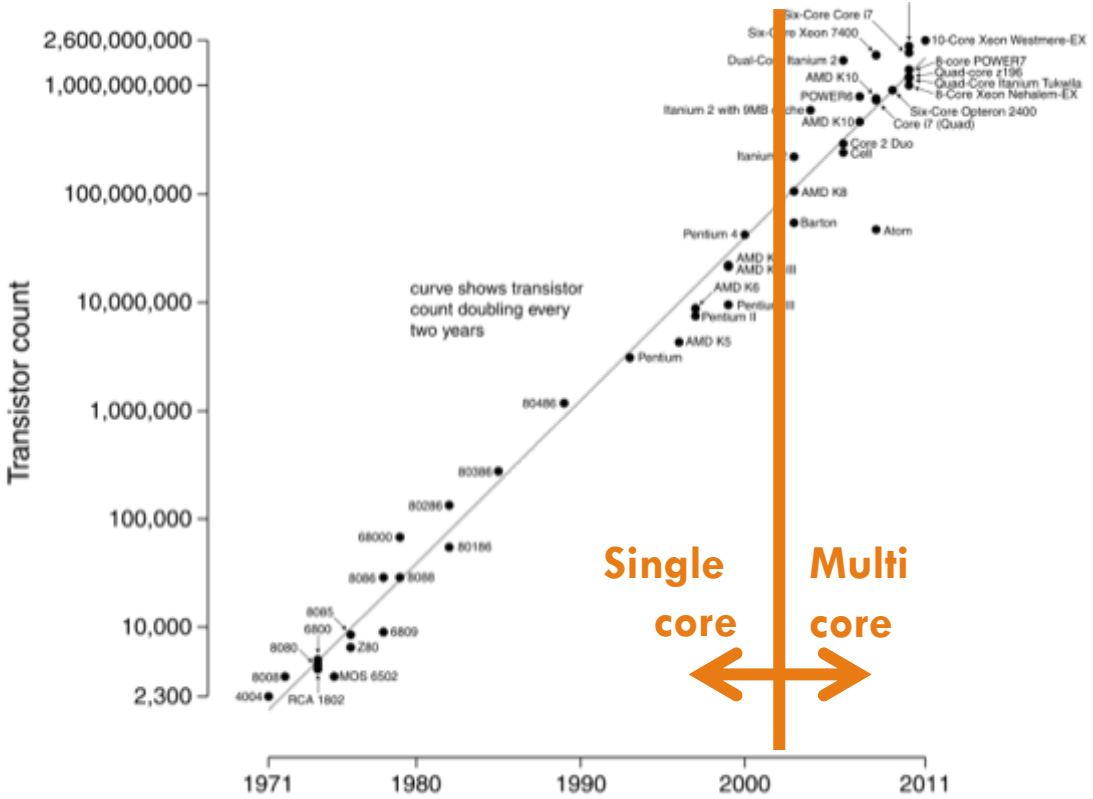
Storage: Hierarchy

- Hierarchy: Exploit temporal and spatial **locality** in programs
- Smaller (closer to CPU) ➔ Faster single access memory
  - Good design: Constant throughput
- Compiler responsibilities
  - Traditional: Register usage
  - Today: Also optimize cache usage



Numbers for 2012, source: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

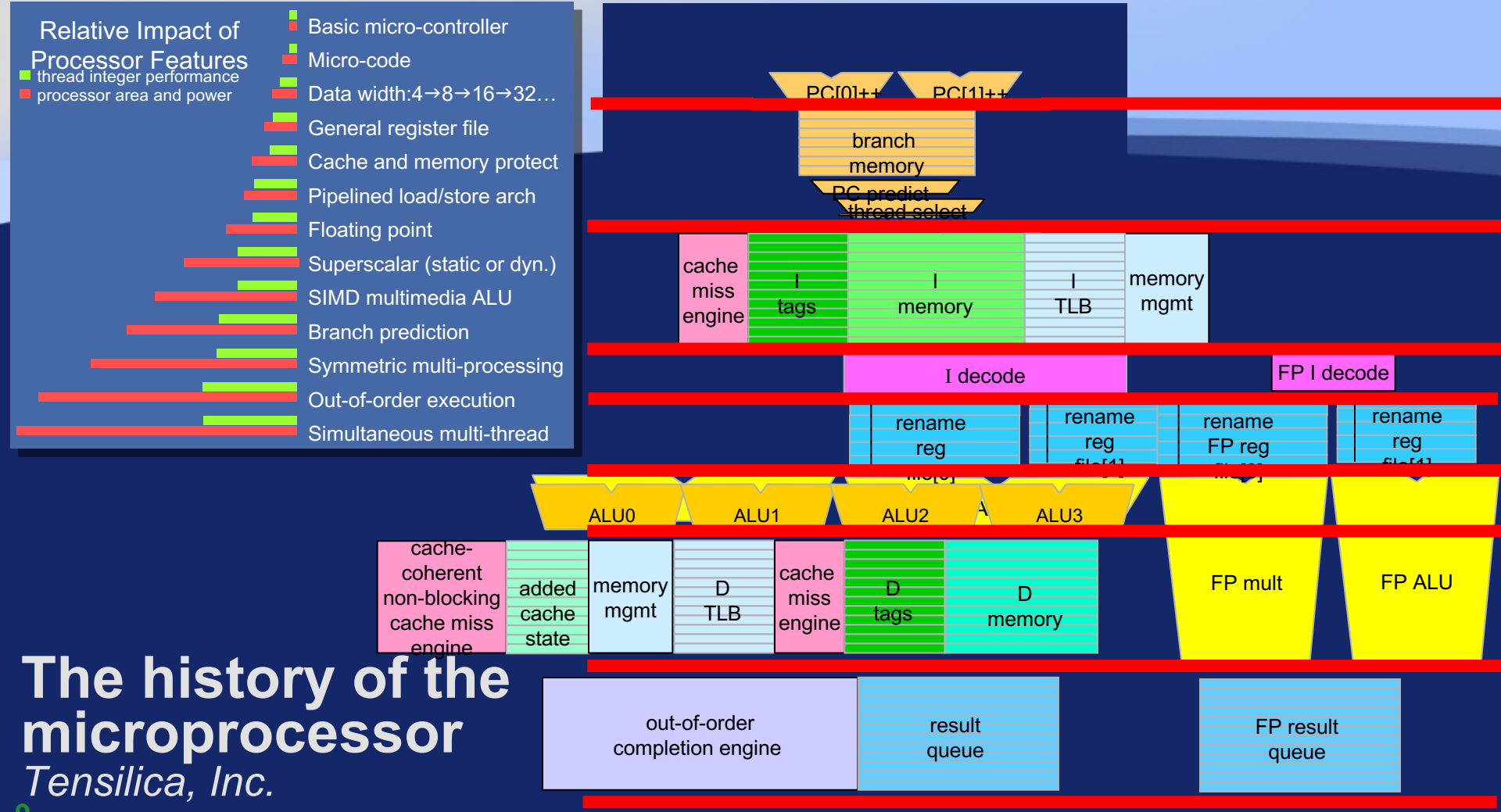
Moore's law



Source: wikipedia.org

- What are we doing with so many transistors?
  - More functionality
  - More complex architectures

➔ Bigger challenge to the compiler, e.g., SIMD, vector instructions, multi-core



The history of the microprocessor

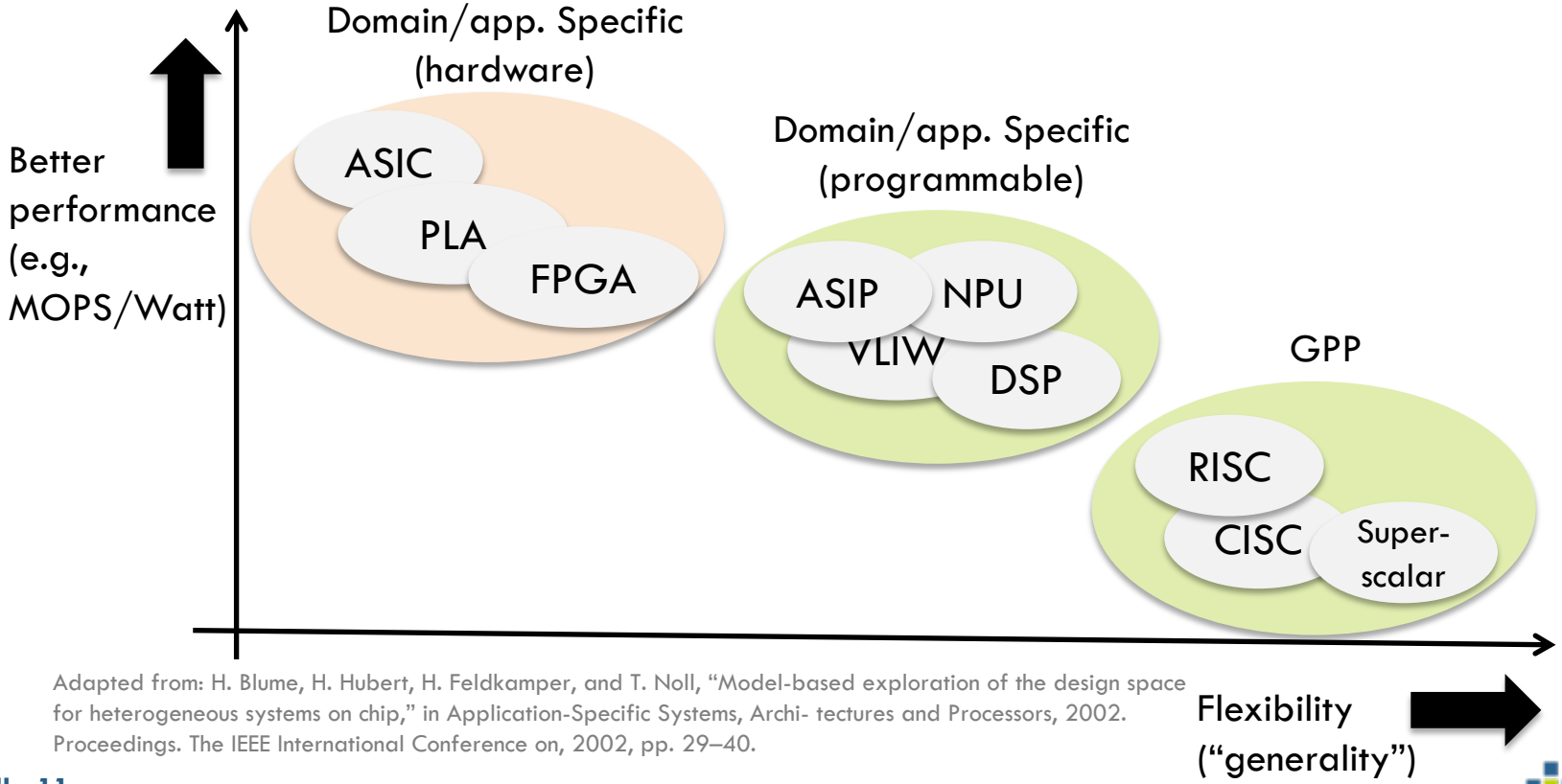
Tensilica, Inc.

Metrics & specialization

- Metrics (see above)
  - Performance (MHz, MIPS, MOPS, ...)
  - Costs (mm^2, number of transistors or gates)
  - Efficiency (MOPS/Watt)
- Specialization: Due to diminishing returns of general purpose processors (see above) and application constraints (e.g., battery powered)
  - General purpose: CISC, RISC, Superscalar
  - Digital signal processors
  - Very long instruction word (VLIW)
  - Network processing units (NPU)
  - Application specific instruction-set processors (ASIPs)

Different architecture classes pose different challenges to the compiler backend

Metrics & specialization (2)



Adapted from: H. Blume, H. Hubert, H. Feldkamper, and T. Noll, "Model-based exploration of the design space for heterogeneous systems on chip," in Application-Specific Systems, Archi- tectures and Processors, 2002. Proceedings. The IEEE International Conference on, 2002, pp. 29–40.

8. Target architectures

- Introduction
- CISC
- RISC
- Superscalar
- DSP
- VLIW

CISC: Complex instruction set computer

- Main characteristic: Multi-cycle, complex instructions, e.g., search inside a list
- Most common CPU architecture in the 1980s

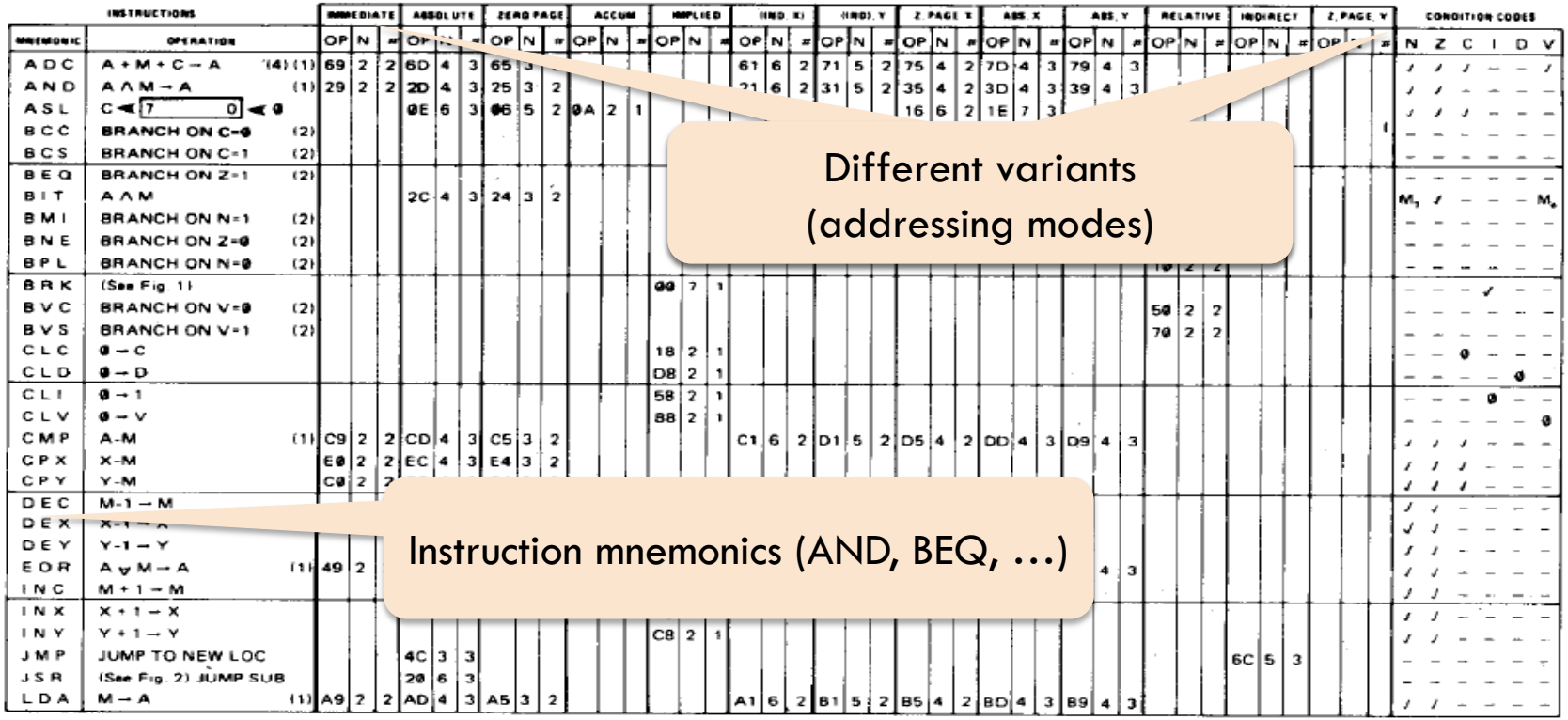
- Motivation/reasons
  - CPU-Memory **speed gap**: CPU could process instructions faster than data arrival
    - ➔ Do many operations per each slow memory access
  - **Semantic gap**: Make assembly closer to programmers (… later came the compiler)

CISC general features

- Von Neumann architecture
- Many instructions
- Many addressing modes
- Memory-to-memory operations
- Special-purpose register, e.g., accumulator and index register
- High-level instructions implemented as **micro-code**

- Challenge for compiler: Recognize patterns and select complex instructions (**code selection**)

CISC Example 1: 6502



INSTRUCTIONS		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM			IMPLIED			(IND, X)			(IND), Y			Z PAGE, X			ABS, X			ABS, Y			RELATIVE			INDIRECT			Z PAGE, Y			CONDITION CODES					
MNEMONIC	OPERATION	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N	Z	C	I	D	V
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3								61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										✓	✓	✓	—	—	✓
AND	A ∧ M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										✓	✓	—	—	—	—
ASL	C ◀ 7 … 0 ◀ 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3											✓	✓	✓	—	—	—
BCC	BRANCH ON C=0 (2)																																							—	—	—	—	—	—
BCS	BRANCH ON C=1 (2)																																							—	—	—	—	—	—
BEQ	BRANCH ON Z=1 (2)																																							—	—	—	—	—	—
BIT	A ∧ M				2C	4	3	24	3	2																														M_7	✓	—	—	—	M_6
BMI	BRANCH ON N=1 (2)																																							—	—	—	—	—	—
BNE	BRANCH ON Z=0 (2)																																							—	—	—	—	—	—
BPL	BRANCH ON N=0 (2)																																							—	—	—	—	—	—
BRK	(See Fig. 11)													00	7	1																								—	—	—	✓	—	—
BVC	BRANCH ON V=0 (2)																															50	2	2						—	—	—	—	—	—
BVS	BRANCH ON V=1 (2)																															70	2	2						—	—	—	—	—	—
CLC	0 → C													18	2	1																								—	—	0	—	—	—
CLD	0 → D													D8	2	1																								—	—	—	—	0	—
CLI	0 → I													58	2	1																								—	—	—	0	—	—
CLV	0 → V													B8	2	1																								—	—	—	—	—	0
CMP	A-M (1)	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										✓	✓	✓	—	—	—
CPX	X-M	E0	2	2	EC	4	3	E4	3	2																															✓	✓	✓	—	—	—
CPY	Y-M	C0	2	2																																					✓	✓	✓	—	—	—
DEC	M-1 → M																																								✓	✓	—	—	—	—
DEX	X-1 → X																																								✓	✓	—	—	—	—
DEY	Y-1 → Y																																								✓	✓	—	—	—	—
EOR	A ⊻ M → A (1)	49	2																																						✓	✓	—	—	—	—
INC	M + 1 → M																																								✓	✓	—	—	—	—
INX	X + 1 → X																																								✓	✓	—	—	—	—
INY	Y + 1 → Y													C8	2	1																									✓	✓	—	—	—	—
JMP	JUMP TO NEW LOC				4C	3	3																														6C	5	3							
JSR	(See Fig. 2) JUMP SUB				20	6	3																																		—	—	—	—	—	—
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										✓	✓	—	—	—	—

Different variants (addressing modes)

Instruction mnemonics (AND, BEQ, …)

CISC Example 2: Motorola C6800

- Motorola architecture in use since 1979
  - Atari, Texas Instruments calculators, Palm, …

```
MOVE.B 8(A1,D1.W)+,D5
```

  - Move a byte from address A1+D1 with 8-bit signed offset to register D5
  - After that, increment A1 to next byte (A1 = A1 + 1)

CISC: Effort in the backend

- Code selection: Hard
  - Many instructions ➔ Options/degrees of freedom to implement the IR
- Register allocation: Easy
  - Special registers ➔ No need to allocate them, given by code selection
- Scheduling: Easy
  - No overlapping execution of instructions

8. Target architectures

- Introduction
- CISC
- RISC
- Superscalar
- DSP
- VLIW

RISC: Reduced instruction set computer

- Main characteristic: Simple instructions with regular format, no micro-code
- Common general purpose architecture today
- Motivation/reasons
  - Memory gap reduced by technology (later mitigated with memory hierarchy)
  - Semantic gap closed by compiler technology
  - Principle: Increase throughput by executing many smaller instructions (enabled by pipelining)
- CISC vs. RISC

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycle}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

RISC simpler HW
➔ Faster clock

CPI (clocks per instruction):
RISC ≈ 1, CISC > 2

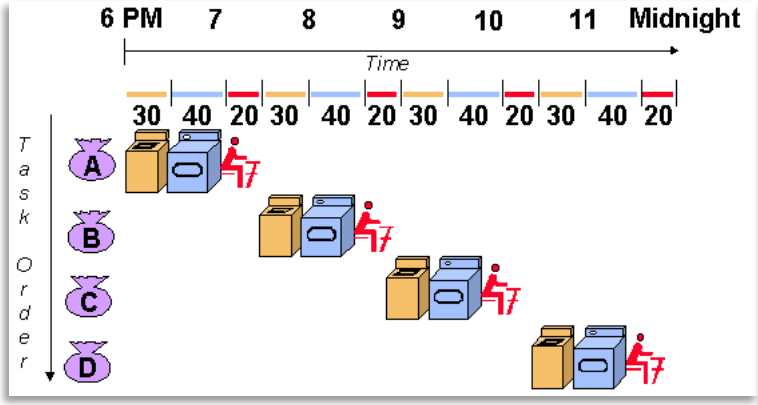
CISC complex ISA
➔ Fewer instructions

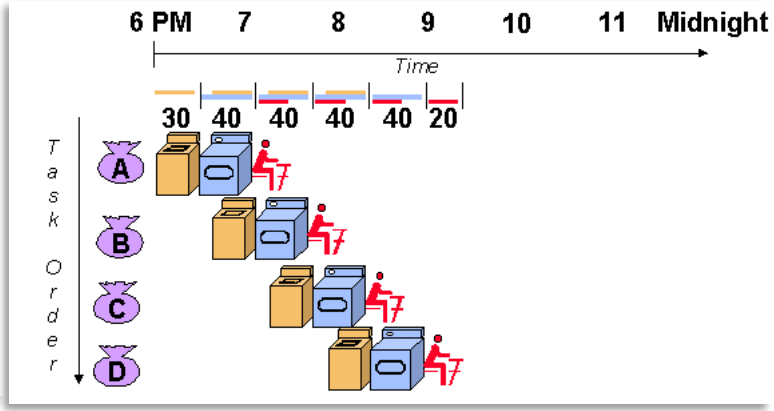
RISC: Load/store architecture

- Common principle of operation by RISC
  - Perform computations locally on registers (i.e., no memory-to-memory instructions)
  - Exploit locality (once registers are loaded with values, re-use them)
  - Eventually load new values and store produced values (load/store)
  - ➔ Relatively large set of registers 16 – 128
  - ➔ Homogeneous register file, i.e., all register are equal for the compiler (e.g., no accumulator)
  - ➔ Good use of registers crucial for performance (**register allocation**)

Pipelining

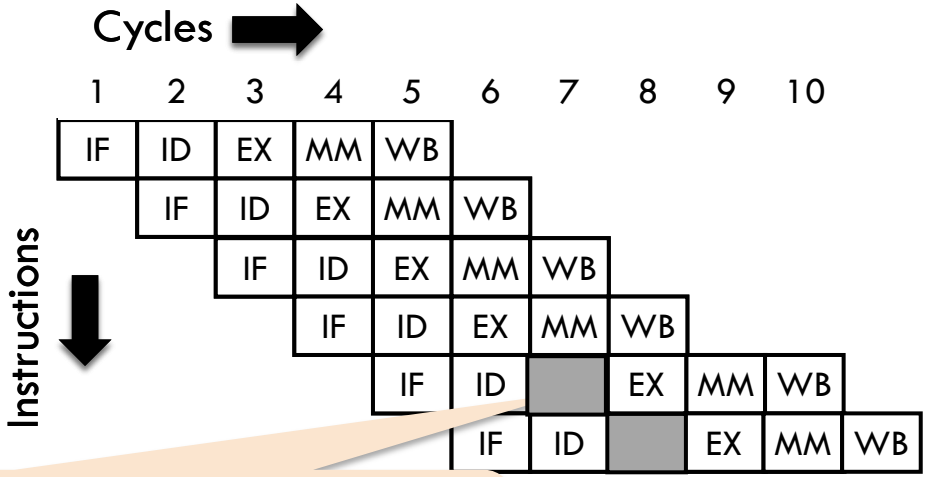
- Introduced in the 1980s, for example for Stanford MIPS processor
- Principle
  - Break instructions into phases (reduce critical path ➔ faster clock)
  - Allow phases of different instructions to execute in parallel
- Prominent "laundry example" (Henessy & Patterson)





Classic RISC pipeline

- Phases/stages
  - Instruction fetch (IF)
  - Instruction decode (ID)
  - Execute instruction (EX)
  - Access memory (MM)
  - Write results back to registers (WB)



Pipeline hazard: Are the operands available? Is there a conflict accessing registers? (more on this later)

- Challenge: Compiler might be responsible for ensuring that the stages can really execute in parallel (**scheduling**)

RISC: Effort in the backend

- Code selection: Easy
  - Simple ISA
- Register allocation: Hard
  - Load/store architecture: registers must be used intelligently to avoid memory delays
- Scheduling: Hard
  - Due to pipeline hazards

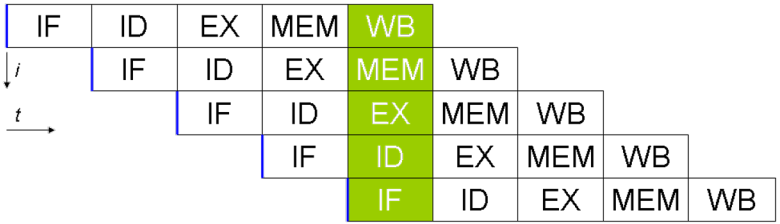
8. Target architectures

- Introduction
- CISC
- RISC
- Superscalar
- DSP
- VLIW

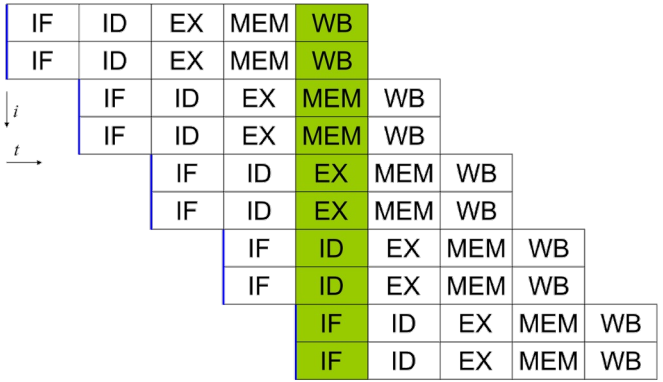
Superscalar

- Main characteristic: Extension of RISC principle – Use multiple functional units to increase parallel execution (i.e., CPI < 1)

Source: wikipedia.org



RISC pipeline

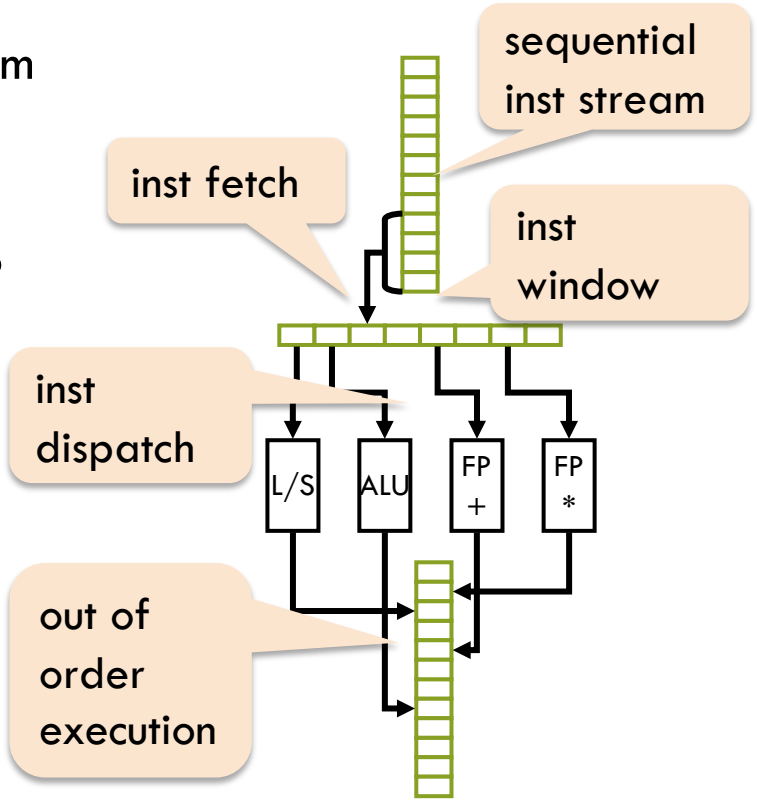


Superscalar pipeline

- Desktop computers are superscalar: Since the first Intel P5 Pentium in 1993

Superscalar characteristics

- Instructions are issued from a sequential stream
- Dynamic dependency checks in hardware
  - ➔ Very “expensive”
- Hardware automatically “send” instructions to functional units
- **Instruction level parallelism (ILP):** CPU processes multiple instructions per cycle (as oppose to scalar RISC)
- **Out of order execution** (optional): Issue and commit of instructions may differ



Superscalar: Effort in the backend

- Common pitfall: The fact that it has HW support does not take away the work of the compiler
- Code selection: Depends on base ISA (CISC or RISC)
- Register allocation: Hard
  - Typically much more registers than by RISC (although alleviated by HW support)
- Scheduling: Hard
  - At a different level than for RISCs due to local HW support

8. Target architectures

- Introduction
- CISC
- RISC
- Superscalar
- DSP
- VLIW

DSP: Digital signal processor

- Main characteristic: Special support for signal processing algorithms
  - Filtering, e.g., finite impulse response (FIR) – inner product
  - Frequency domain processing, e.g., fast Fourier transforms (FFT) – matrix-vector multiply

$$y = \vec{x} \cdot \vec{c} = \sum_{i=1}^{n} x_i \cdot c_i$$

Multiply and accumulate

Read vectors from memory

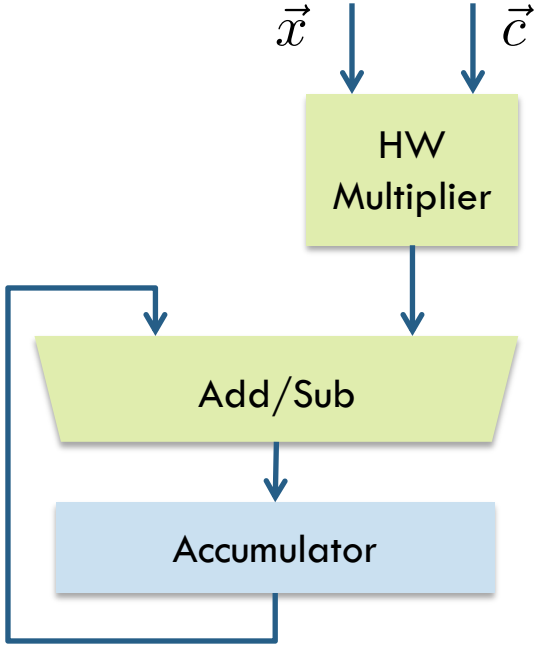
Different number representations

- Present in mobile phones, base stations, TVs, …

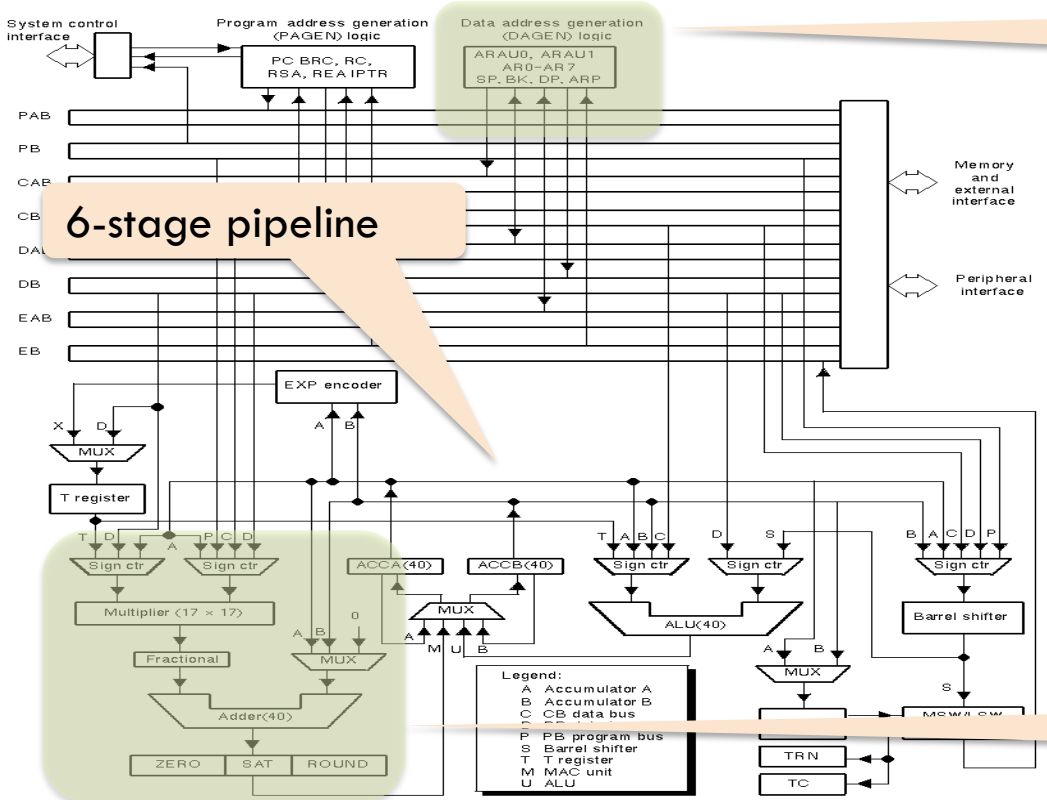
DSP Characteristics

- Data-path (usually also pipelined)
  - Multiply-accumulate hardware
  - Saturation arithmetic
  - Special instructions: Butterfly, Multiply-accumulate (MAC)
- Accessing memories
  - Address generation units (AGUs)
  - Harvard architecture (& mem-to-mem operations)
  - Memory banks
- Others
  - Zero-overhead loops
  - Special registers (accumulator, index registers)

$$y = \vec{x} \cdot \vec{c} = \sum_{i=1}^{n} x_i \cdot c_i$$

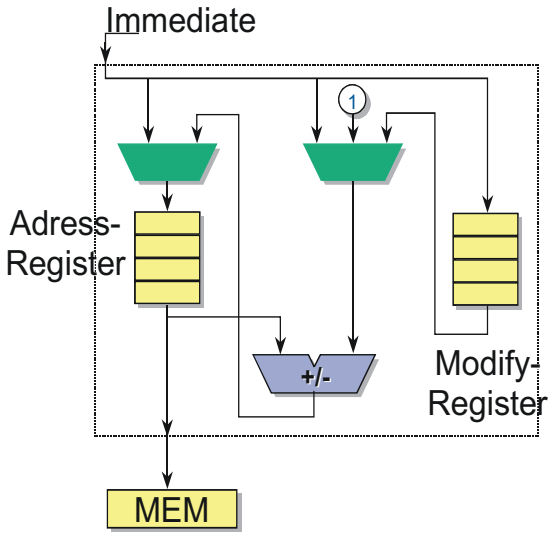


DSP Example: Texas Instruments C54x



6-stage pipeline

Address generation unit (AGU)



Multiply-accumulate (MAC) unit

DSP: Effort in the backend

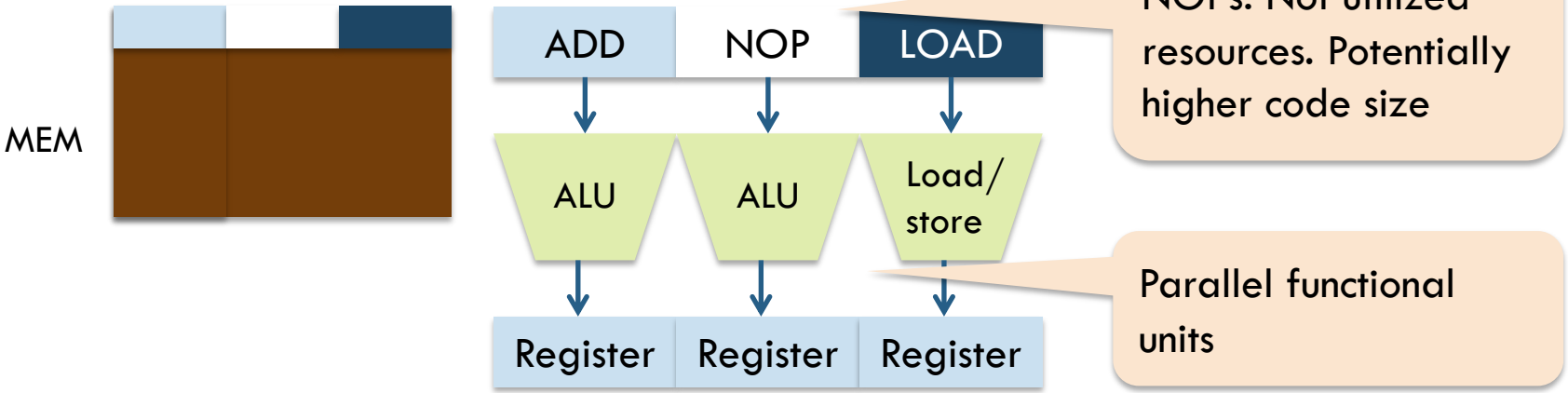
- DSPs are considered the most challenging for compiler backend
- Code selection: Hard, because of special instructions (similar to CISC)
- Register allocation: Hard
  - General purpose registers and dedicated registers (dependent on code selection)
- Scheduling: Hard (due to pipelining similar to RISC)
- Additional optimization for programing address generation units

8. Target architectures

- Introduction
- CISC
- RISC
- Superscalar
- DSP
- VLIW

VLIW: Very large instruction word

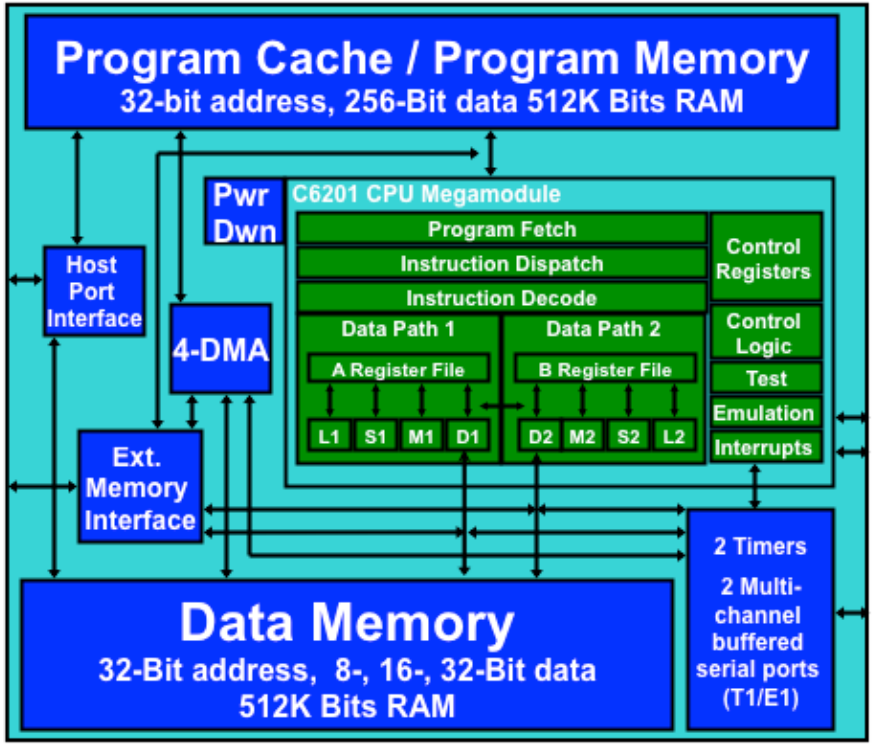
- Main characteristic: One fetch from memory includes several instructions
  - ILP similar to superscalars (CPI < 1), but dependency checks done at **compile time**
  - Single instructions are RISC-like



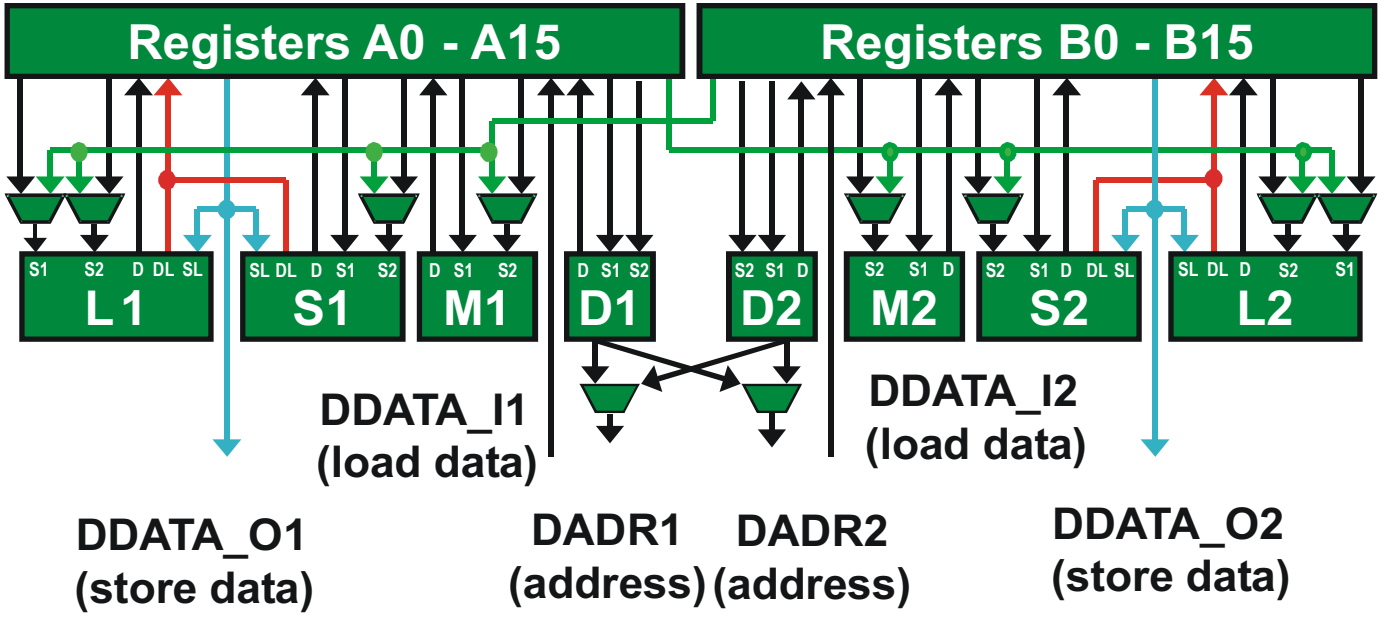
- Used in embedded devices (low energy) and for DSP (VLIW-DSP)

VLIW Example: Texas Instruments C6201 VLIW-DSP

- 8 parallel functional units (8x32 = **256bit instruction word**)
- Data-path separated into two identical clusters
  - Reduce register file complexity
  - Additional constraints for backend (need to insert copy operations)
- Data-path functional units (L, S, M, D) with different supported operations



C6201 Data-path



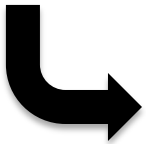
C6201 functional units

.S Unit		.L Unit		.D Unit		.M Unit	
ADD	MVKLH	ABS	NOT	ADD	STB/H/W	MPY	SMPY
ADDK	NEG	ADD	OR	ADDA	SUB	MPYH	SMPYH
ADD2	NOT	AND	SADD	LDB/H/W	SUBA		
AND	OR	CMPEQ	SAT	MV	ZERO		
B	SET	CMPGT		NEG			
CLR	SHL	CMPLT	SUB				
EXT	SHR	LMBD	SUBC				
MV	SSHL	MV	XOR				
MVC	SUB	NEG	ZERO				
MVK	SUB2	NORM					
MVKL	XOR						
MVKH	ZERO						

Degrees of freedom: Need to be handled in the backend (scheduling)

C6201: Code example

```
int dotp(short a[], short b[])
{ int sum0, sum1, i;
  sum0 = sum1 = 0;
  for (i = 0; i < 100; i += 2)
  { sum0 += a[i] * b[i];
    sum1 += a[i+1] * b[i+1]; }
  return sum0 + sum1;
}
```



```
L:    LDH .D2 *++B4(4),B6 || LDH .D1 *++A4(4),A5
      LDH .D2 *+B4(2),B5  || LDH .D1 *+A4(2),A6
      SUB .L2 B0,1,B0
[B0]  B .S1 L
      NOP 1
      MPY .M1X B6,A5,A5
      MPY .M1X B5,A6,A6
      NOP 1
      ADD .L1 A6,A0,A0 || ADD .S1 A5,A3,A3
```

Unrolled loop (factor 2)

Specifies unit to execute instructions

Parallel execution of instructions

Need to know latencies

Code written by hand, can we do better?

C6201: Code example

```
L:    LDH .D2 *++B4(4),B6 || LDH .D1 *++A4(4),A5
      LDH .D2 *+B4(2),B5  || LDH .D1 *+A4(2),A6
      SUB .L2 B0,1,B0
[B0]  B .S1 L
      NOP 1
      MPY .M1X B6,A5,A5
      MPY .M1X B5,A6,A6
      NOP 1
      ADD .L1 A6,A0,A0 || ADD .S1 A5,A3,A3
```

```
L3:
             ADD      .L2     B6,B5,B5
||           MPY      .M2X    B4,A0,B6
||  [ B0]    B        .S1     L3
||           LDH      .D2     *+B7(2),B4
||           LDH      .D1     *+A3(2),A0

             ADD      .L1     A5,A4,A4
||           MPY      .M1X    B4,A0,A5
||  [ B0]    SUB      .L2     B0,1,B0
||           LDH      .D2     *++B7(4),B4
||           LDH      .D1     *++A3(4),A0
```

	0	1	2	3	4	5	6	7	8
D1	LDH	LDH							
D2	LDH	LDH							
M1						MPY	MPY		
M2									
L1									ADD
L2			SUB						
S1				B					ADD
S2									

	0	1	2	3	4	5	6	7	8
D1	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH
D2	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH
M1							MPY		MPY
M2						MPY		MPY	
L1									ADD
L2			SUB		SUB		SUB		SUB
S1				B		B		B	ADD
S2									

SW pipelining: 4.5x speedup (more on this later)

VLIW: Effort in the backend

- Code selection: “Easy” (similar to RISC), harder if it includes DSP instructions
- Register allocation: Hard (similar to RISC and Superscalar)
- Scheduling: Hard
  - The most important phase in VLIW: No HW support for ILP
  - From sequential language (like C), compiler needs to re-structure code to expose more ILP

Where are we?

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

9. Code selection

- Introduction
- Maximal munch
- Tree parsing

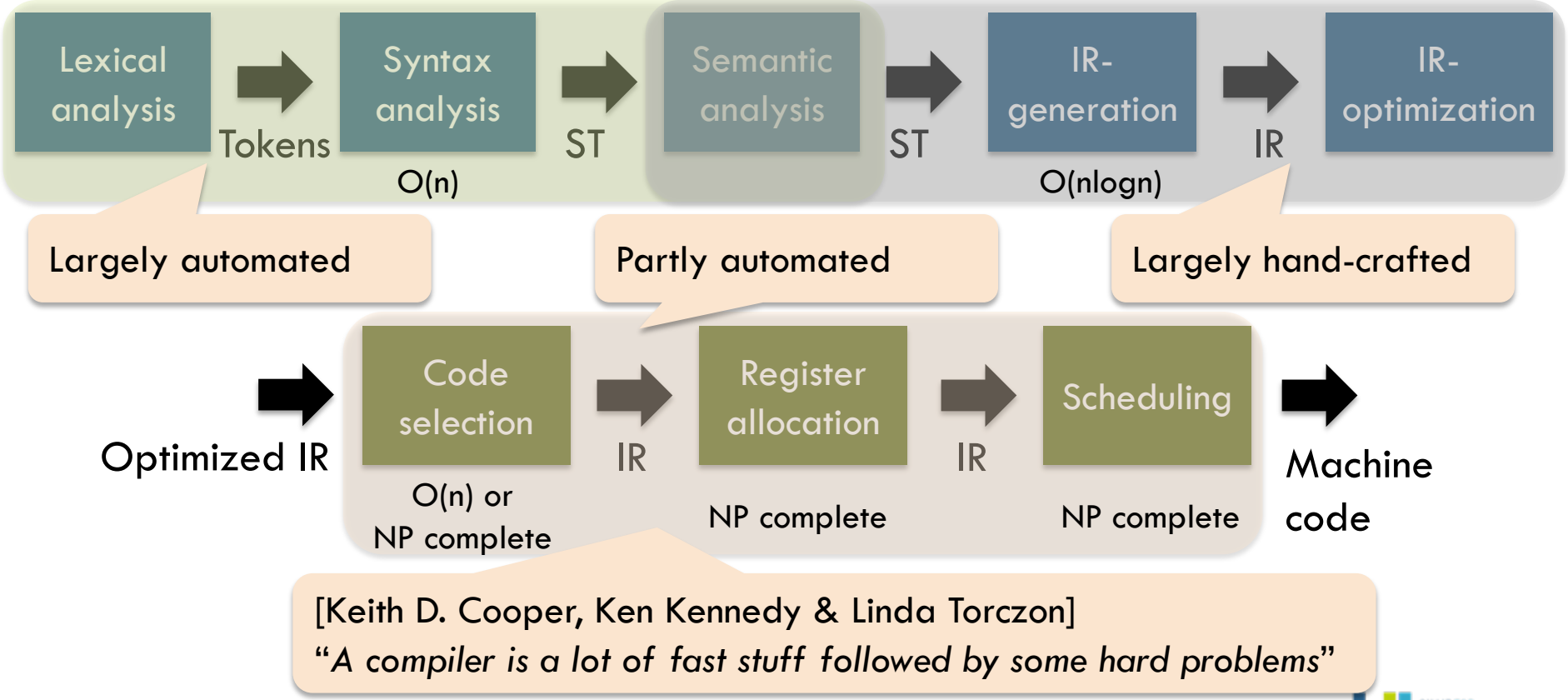
Backend: Recall


```
while (y < z)
{
  int x = a + b;
  y += x;
}
```

```
      x = a + b
Loop: y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

```
      ADD R1, R2, R3
Loop: ADD R4, R1, R4
      SLT R6, R4, R5
      BEQ R6, loop
```

Backend: Recall (2)

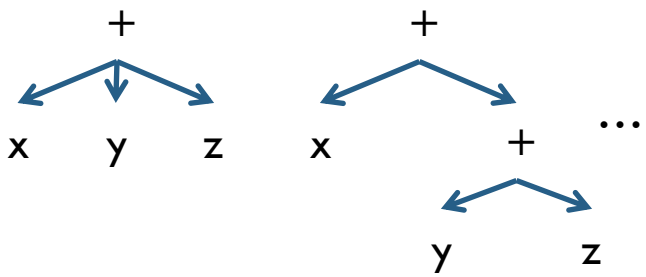


Backend: Overview

- Code selection: Uses pattern matching
  - Map IR code to assembly instructions (assume fixed code shape, i.e., IR is “optimal”)
  - Combine operations, use addressing modes (assume infinite registers)
- Register allocation
  - Assign registers to variables (change storage pattern, i.e., inserts load/stores)
- Scheduling
  - Reorder operations to hide latencies, assume a fixed program (set of operations)
- Phase ordering problem: Phases actually affect each other

Code selection: Code shape

- We assume that the code shape is fixed in the IR
- Example 1
  - x + y + z



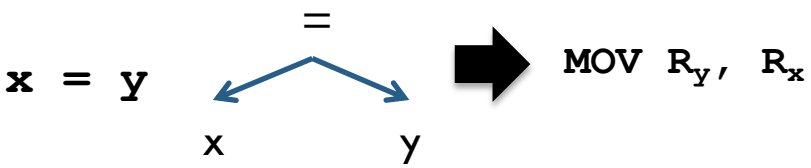
What if x is 2 and z is 3?
➔ Already handled by IR-optimization

- Example 2: Switch-case implementation
  - Sequence of if-then-else
  - For loop
  - Jump table
  - …

Code selection: Definition

Def.: **Code selection** is the process of converting a given IR into an optimal sequence of instructions from a given ISA w.r.t. to a metric, e.g., performance

- Intuition: Degrees of freedom

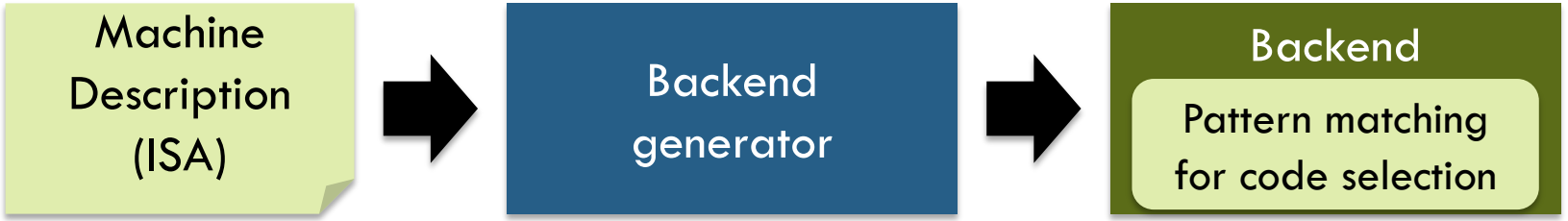



```
ADD Ry, 0, Rx
MUL Ry, 1, Rx
OR  Ry, 0, Rx
...
```

Some architectures don't even have a MOV

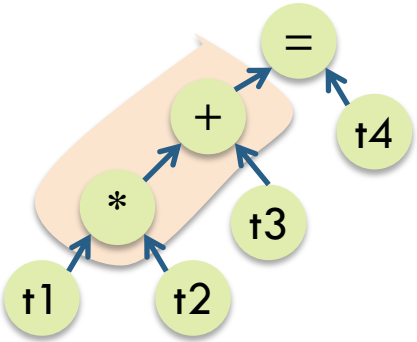
Which one to select? Depends on costs, context,…

- Goal: Build a "**code generator generator**"



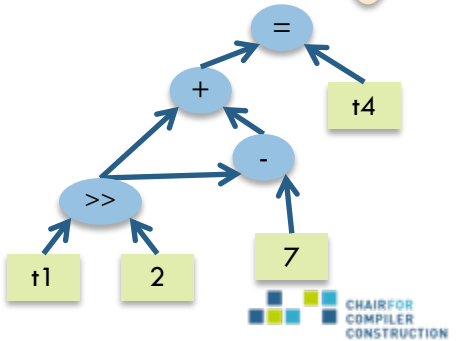
IR vs. assembly

- In general no one-to-one mapping from IR to assembly instructions
  - Ex.1: 32-bit multiplication onto ISA with a 16-bit data path
  - Ex.2: An IR with nodes for “*” and “+”onto ISA with MAC



- IR format: TAC, ASTs, DAGs?
  - TAC: Too fine grained, lose optimization opportunities
  - DAGs representation of basic-blocks: Great potential, but NP complete
  - ASTs: Good trade-off potential/complexity (O(n))

```
t5 = t1 * t2
t4 = t3 + t5
```



Example: Motorola C6800 – CISC

- Recall instruction (complex addressing modes) `MOVE.B 8(A1,D1.W),D5`
- Cost table: Cost = Basic cost + Addressing cost

	.B, .W	.L
D_n	0	0
A_n	0	0
(A_n)	4	8
$d(A_n)$	8	12
$d(A_n,I_x)$	10	14
x (short)	8	12
x (long)	12	16
#x	4	8

Example: Motorola C6800 – CISC (2)

- Recall instruction (complex addressing modes) `MOVE.B 8(A1,D1.W),D5`
- Costs: 4 for MOVE.B, 10 for addressing $d(A_n,I_x)$ – Total **14 cycles**

- Variant 1

```
ADDA #8,A1          Cost: 16
ADDA D1.W,A1        Cost:  8
MOVE.B (A1),D5      Cost:  8
```

➔ Total **32 cycles**

- Variant 2

```
ADDA D1.W,A1        Cost:  8
MOVE.B 8(A1),D5     Cost: 12
```

➔ Total **20 cycles**

Code selection: Strategy

- Need to judge different alternatives by using a cost function
- Code selection as pattern-matching problem
  - ISA represented as patterns (trees)
  - IR represented as AST (tree)

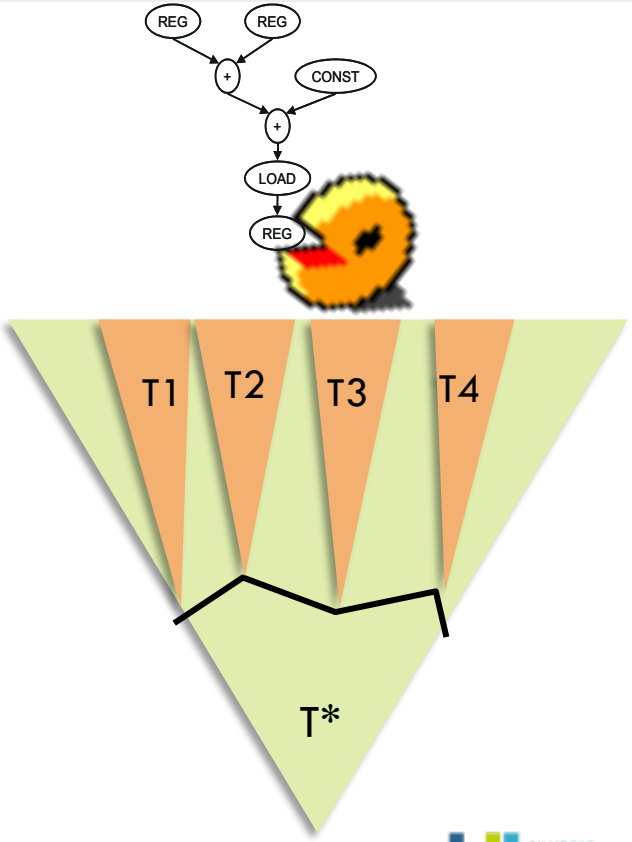
➔ **Code selection has to find a covering of the AST with instances of the ISA patterns with minimal cost**

9. Code selection

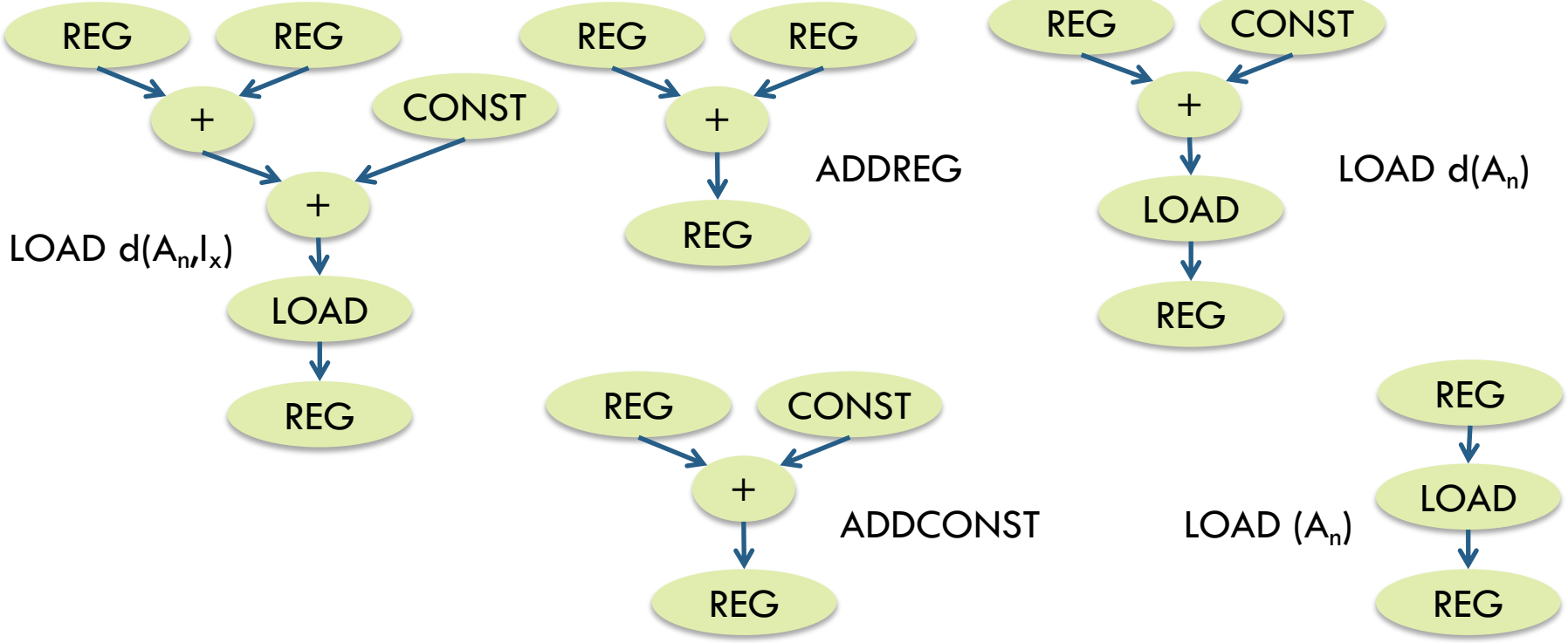
- Introduction
- Maximal munch
- Tree parsing

Maximal munch

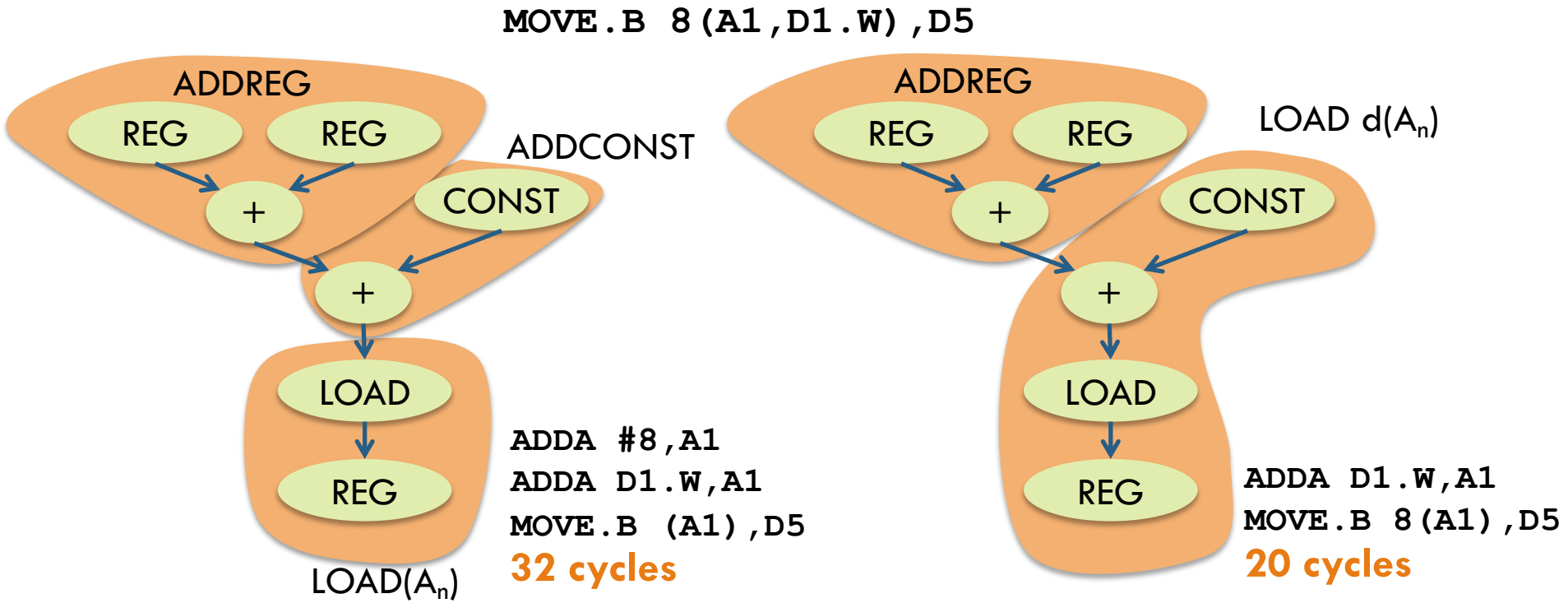
- Greedy heuristic
- Intuition
  - Start covering the IR AST at the root
  - Try to “eat” as much as possible, i.e., find biggest matching pattern from ISA
  - Proceed recursively in the remaining sub-trees in the AST



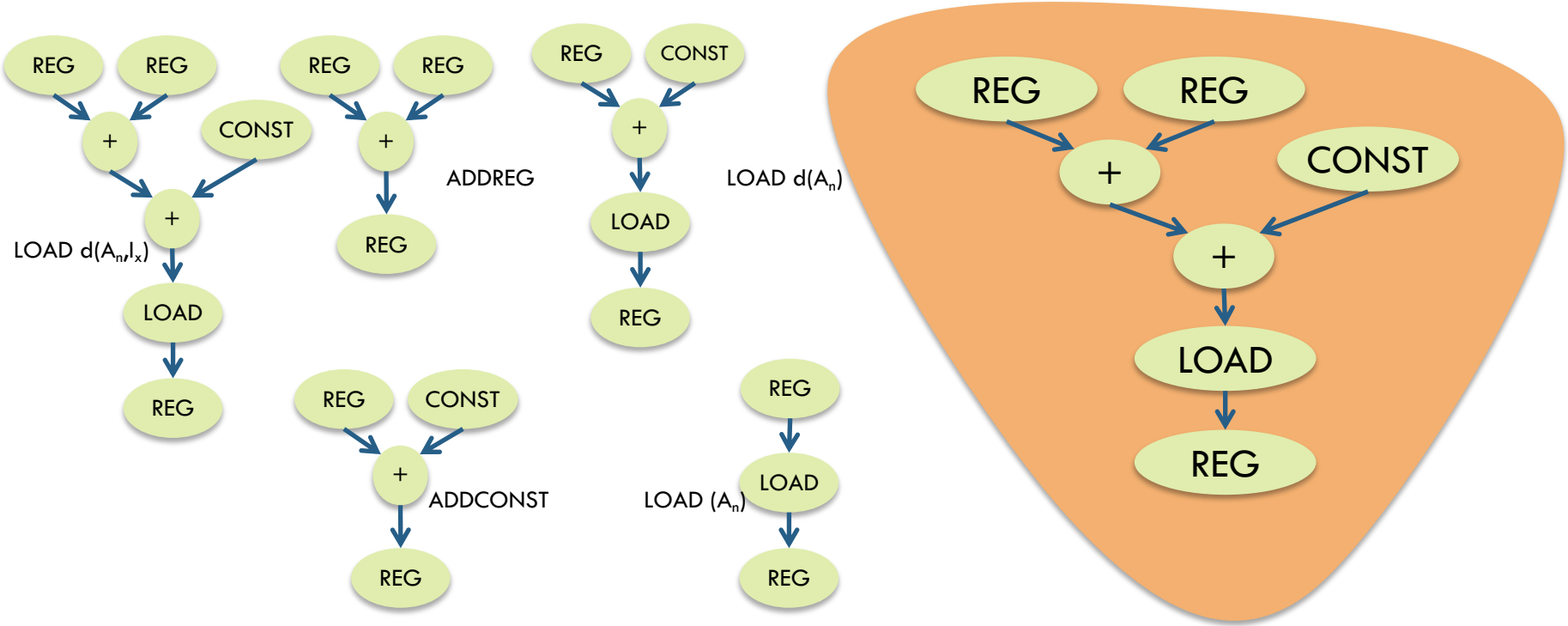
Example: Instruction patterns for C6800



Variants 1 & 2



Maximal munch: Covering




```
MOVE.B 8(A1,D1.W),D5
```

14 cycles

Maximal munch: Algorithm

Proc MaxMunch

Input AST T, ISA pattern set P

Output Set S: Instances from P

R = root(T)

P* = Biggest pattern from P that covers R and a subtree T* from T

$\{T_1, T_2, \ldots, T_n\}$ = sub-trees(T, T*) // List of trees when removing T* from T

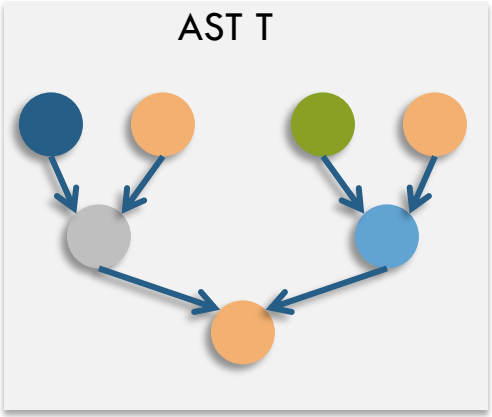
foreach $T_i \in \{T_1, T_2, \ldots, T_n\}$

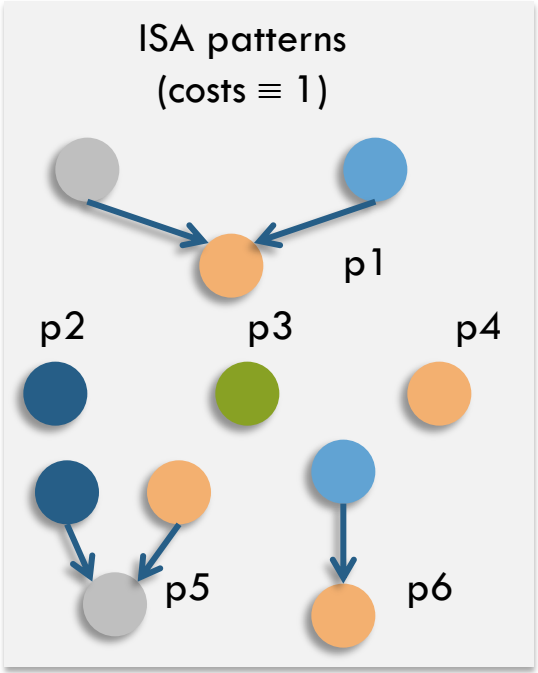
    MaxMunch(T_i)

EmitCode(T*)

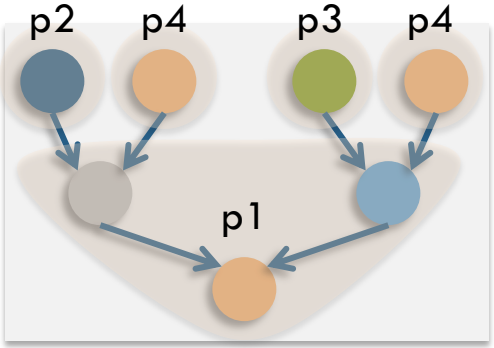
Maximal munch: Analysis

- Complexity for n nodes and k instruction patterns: O(n*k) ≈ O(n)
- Optimality: Not guaranteed
- Example

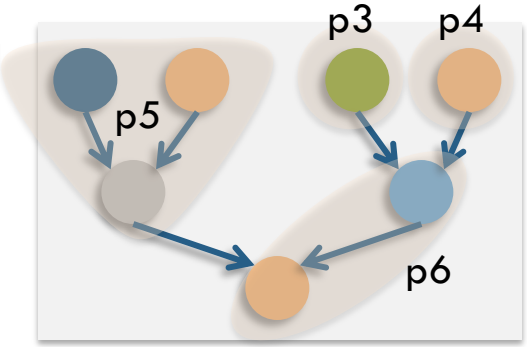




Max. Munch



Optimal



9. Code selection

- Introduction
- Maximal munch
- Tree parsing

Tree parsing: Rationale

- Describe ISA as a **tree grammar**, i.e., special case of a **context-free grammar**
- Tree grammar G: Rules have a tree structure
  - Binary operator reg → PLUS(reg, CONST)
  - Unary operator reg → NEG(reg)
  - No operator reg → CONST
  - Chain rules reg1 → reg2
- Each rule has associated costs (the costs of the instruction it represents)

Operators (inner AST nodes) are **terminal** symbols

Non-terminal symbols: Registers and help symbols

A **tree parser** computes an optimal derivation of an AST w.r.t. the tree-grammar G, i.e., a derivation with minimal total cost

Tree parsing: Intuition

- Example of **dynamic programming**: Global optimum can be constructed from a combination of optimal local solutions
- For a cost-annotated tree grammar G = (N,T,P,S) for an ISA and an input AST T
- Tree parsing bottom-up pass (leaves to root)
  - For current AST node k, select all non-terminals N' ⊂ N, such that there is a rule that N' → k'α (k' represents the content of node k)
  - Record the rule that "covers" k with the least cost
  - Record all possible chain rules
- Tree parsing top-down pass (root to leaves): Cheapest rules are known
  - Find out which arguments are needed for the cheapest rule at node k
  - Insert chain instructions if needed
  - Generate code

Tree parsing example: ISA Grammar G

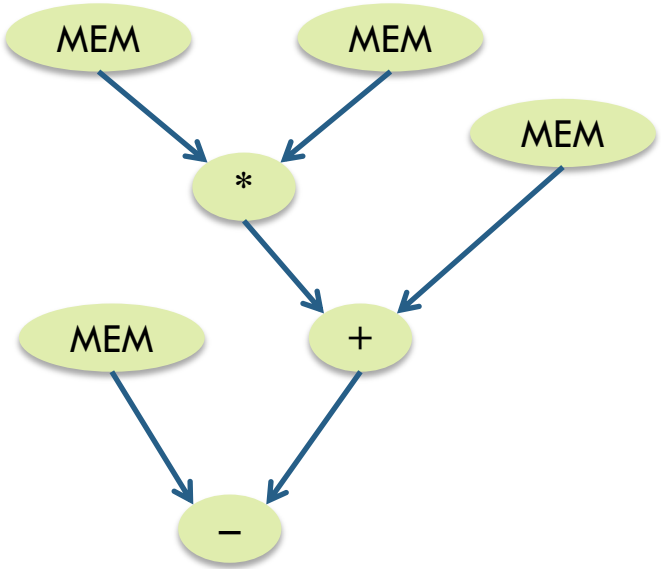
- **Terminals** T = {MEM, *, –, +}
- **Non-terminals** N = {reg1, reg2, reg3}
- **Rules(instructions)** P = …

Start symbol S = reg1

1. "add" (cost = 2): reg1 → + (reg1, reg2)
2. "add" (cost = 2): reg2 → + (reg1, MEM)
3. "sub" (cost = 2): reg1 → – (reg1, reg2)
4. "sub" (cost = 2): reg1 → – (MEM, reg1)
5. "mul" (cost = 2): reg1 → * (reg1,reg2)
6. "mul" (cost = 2): reg2 → * (reg1,MEM)
7. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), reg3)
8. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), MEM)
9. "mov" (cost = 1): reg1-3 → reg1-reg3
10. "load" (cost = 1): reg2 → MEM

reg1-reg3 represent register classes and not physical registers, e.g., VLIW clusters.

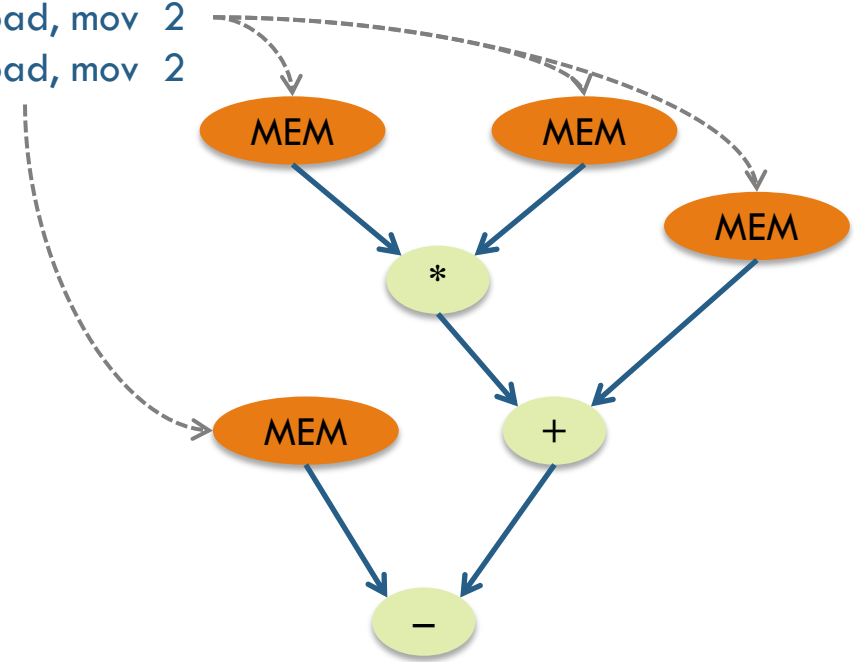
Tree parsing example: Bottom-up pass



1. “add” (cost = 2): reg1 → + (reg1, reg2)
2. “add” (cost = 2): reg2 → + (reg1, MEM)
3. “sub” (cost = 2): reg1 → – (reg1, reg2)
4. “sub” (cost = 2): reg1 → – (MEM, reg1)
5. “mul” (cost = 2): reg1 → * (reg1,reg2)
6. “mul” (cost = 2): reg2 → * (reg1,MEM)
7. “mac” (cost = 2): reg1 →+ (*(reg1,reg2), reg3)
8. “mac” (cost = 2): reg1 →+ (*(reg1,reg2), MEM)
9. “mov” (cost = 1): reg1-3 → reg1-reg3
10. “load” (cost = 1): reg2 → MEM

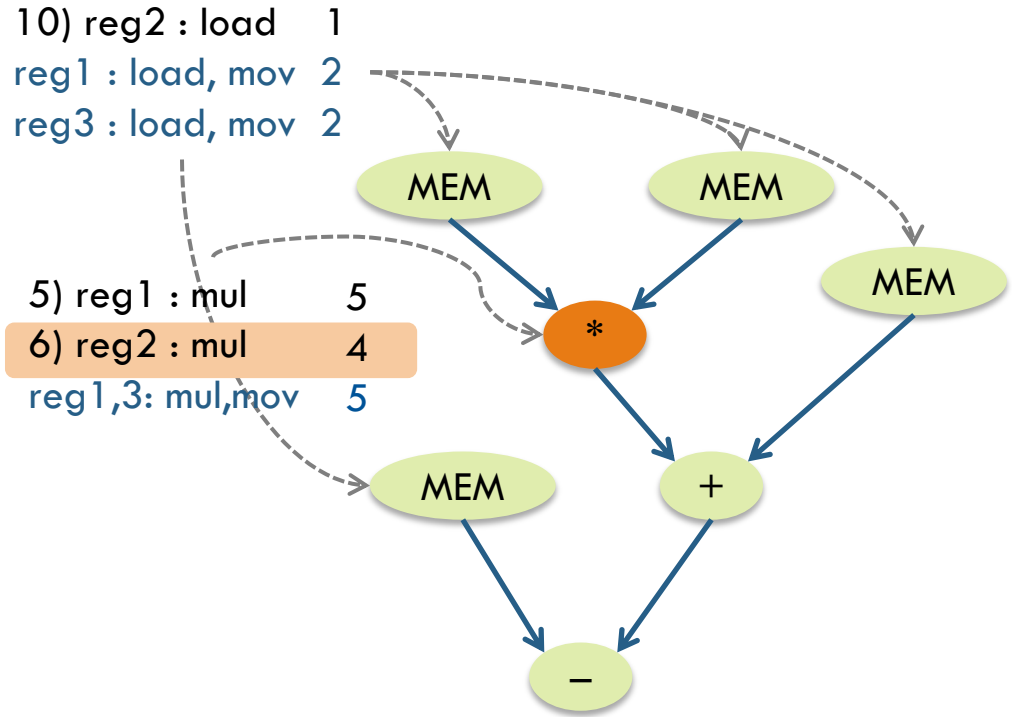
Tree parsing example: Bottom-up pass

10) reg2 : load 1
reg1 : load, mov 2
reg3 : load, mov 2



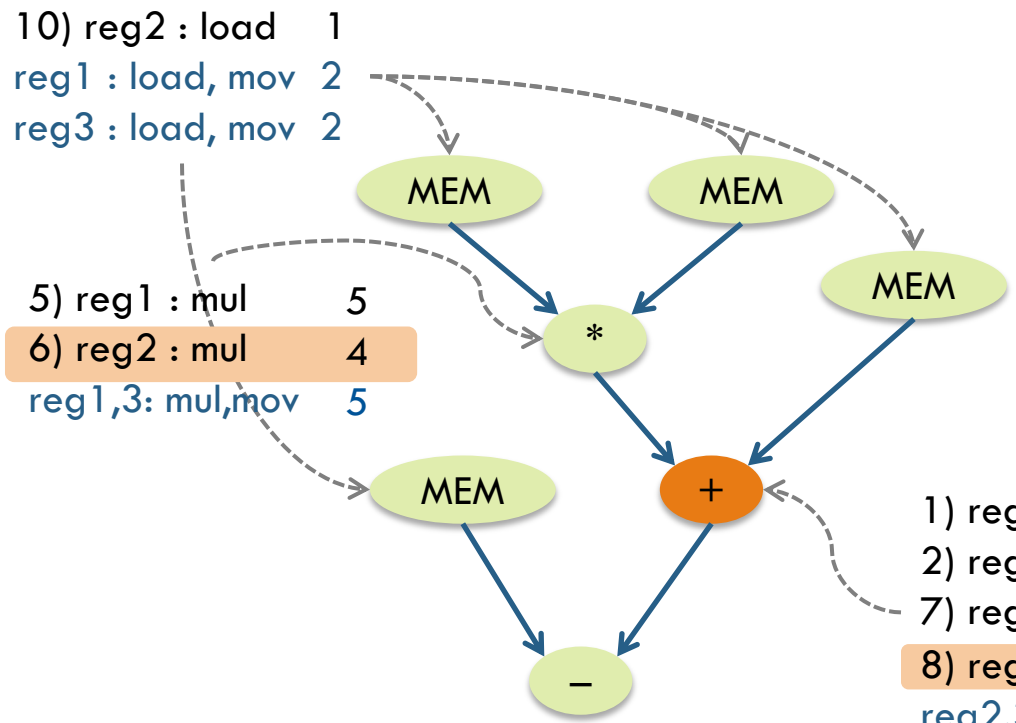
1. "add" (cost = 2): reg1 → + (reg1, reg2)
2. "add" (cost = 2): reg2 → + (reg1, MEM)
3. "sub" (cost = 2): reg1 → – (reg1, reg2)
4. "sub" (cost = 2): reg1 → – (MEM, reg1)
5. "mul" (cost = 2): reg1 → * (reg1,reg2)
6. "mul" (cost = 2): reg2 → * (reg1,MEM)
7. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), reg3)
8. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), MEM)
9. **"mov" (cost = 1): reg1-3 → reg1-reg3**
10. **"load" (cost = 1): reg2 → MEM**

Tree parsing example: Bottom-up pass



1. "add" (cost = 2): reg1 → + (reg1, reg2)
2. "add" (cost = 2): reg2 → + (reg1, MEM)
3. "sub" (cost = 2): reg1 → – (reg1, reg2)
4. "sub" (cost = 2): reg1 → – (MEM, reg1)
5. **"mul" (cost = 2): reg1 → * (reg1,reg2)**
6. **"mul" (cost = 2): reg2 → * (reg1,MEM)**
7. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), reg3)
8. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), MEM)
9. **"mov" (cost = 1): reg1-3 → reg1-reg3**
10. "load" (cost = 1): reg2 → MEM

Tree parsing example: Bottom-up pass



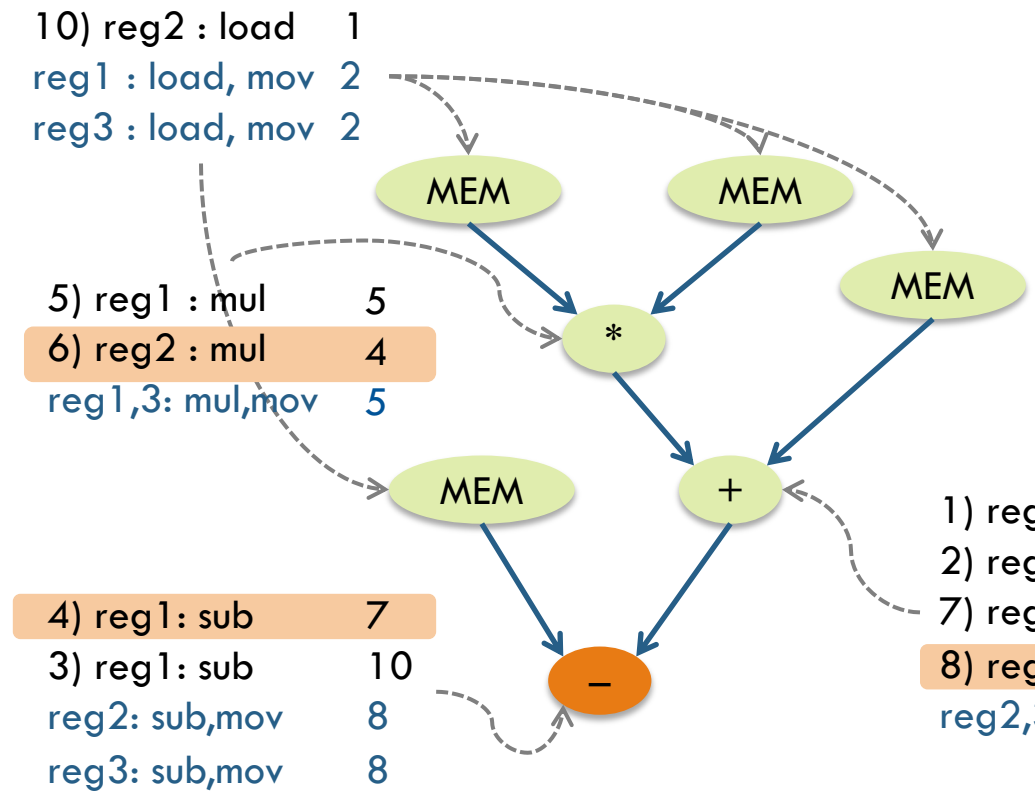
10) reg2 : load 1
reg1 : load, mov 2
reg3 : load, mov 2

5) reg1 : mul 5
6) reg2 : mul 4
reg1,3: mul,mov 5

1) reg1: add 8
2) reg2: add 7
7) reg1: mac 7
8) reg1: mac 5
reg2,3 : mac, mov 6

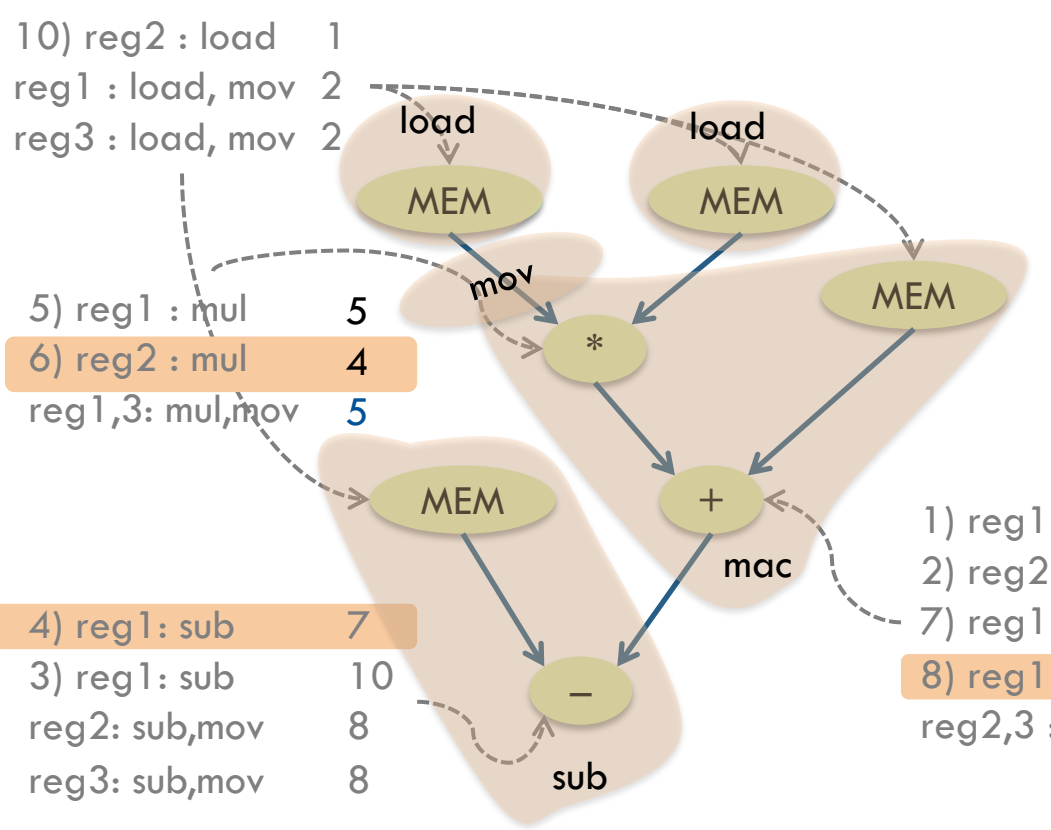
1. **"add" (cost = 2): reg1 → + (reg1, reg2)**
2. **"add" (cost = 2): reg2 → + (reg1, MEM)**
3. "sub" (cost = 2): reg1 → – (reg1, reg2)
4. "sub" (cost = 2): reg1 → – (MEM, reg1)
5. "mul" (cost = 2): reg1 → * (reg1,reg2)
6. "mul" (cost = 2): reg2 → * (reg1,MEM)
7. **"mac" (cost = 2): reg1 →+ (*(reg1,reg2), reg3)**
8. **"mac" (cost = 2): reg1 →+ (*(reg1,reg2), MEM)**
9. **"mov" (cost = 1): reg1-3 → reg1-reg3**
10. "load" (cost = 1): reg2 → MEM

Tree parsing example: Bottom-up pass



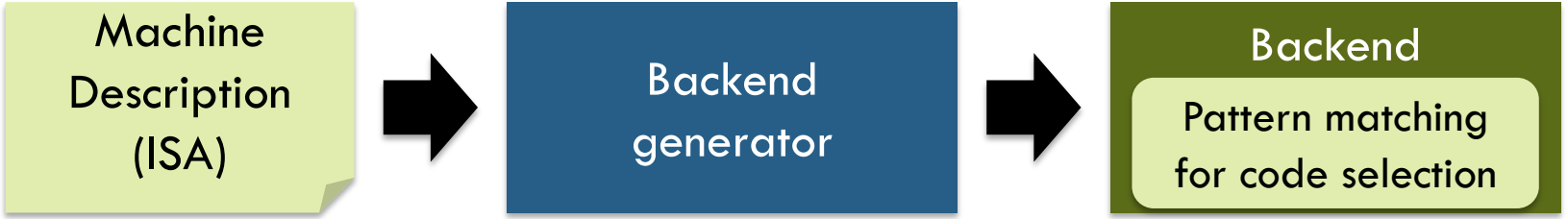
1. "add" (cost = 2): reg1 → + (reg1, reg2)
2. "add" (cost = 2): reg2 → + (reg1, MEM)
3. **"sub" (cost = 2): reg1 → – (reg1, reg2)**
4. **"sub" (cost = 2): reg1 → – (MEM, reg1)**
5. "mul" (cost = 2): reg1 → * (reg1,reg2)
6. "mul" (cost = 2): reg2 → * (reg1,MEM)
7. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), reg3)
8. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), MEM)
9. **"mov" (cost = 1): reg1-3 → reg1-reg3**
10. "load" (cost = 1): reg2 → MEM

Tree parsing example: Top-down pass



1. "add" (cost = 2): reg1 → + (reg1, reg2)
2. "add" (cost = 2): reg2 → + (reg1, MEM)
3. "sub" (cost = 2): reg1 → – (reg1, reg2)
4. "sub" (cost = 2): reg1 → – (MEM, reg1)
5. "mul" (cost = 2): reg1 → * (reg1,reg2)
6. "mul" (cost = 2): reg2 → * (reg1,MEM)
7. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), reg3)
8. "mac" (cost = 2): reg1 →+ (*(reg1,reg2), MEM)
9. "mov" (cost = 1): reg1-3 → reg1-reg3
10. "load" (cost = 1): reg2 → MEM

Tree parser generator: iburg tool



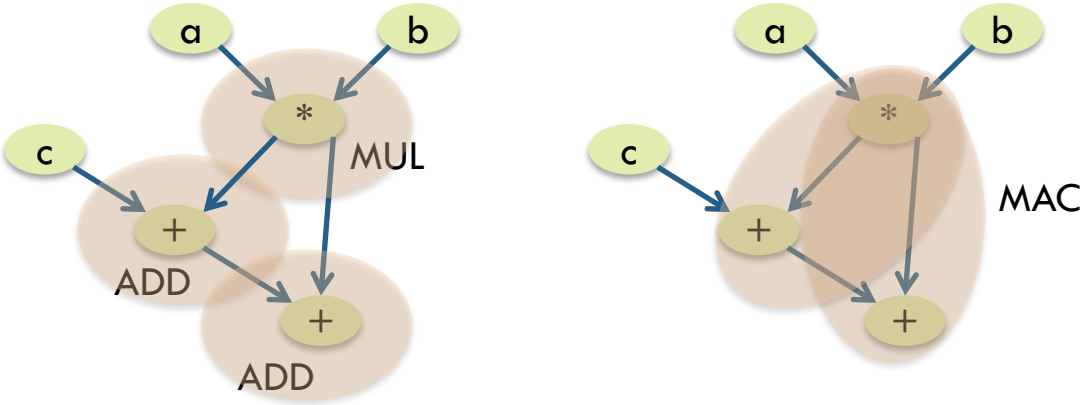
- Similar to lex and yacc
- Input: Tree grammar (.brg)
- Output: ISA.c, ISA.h to link with the compiler backend

grammar	→	{ *dcl* } %% { *rule* }
dcl	→	%start *nonterm*
	|	%term { *identifier* = *integer* }
rule	→	*nonterm* : *tree* = *integer* [*cost*] ;
cost	→	(*integer*)
tree	→	*term* (*tree* , *tree*)
	|	*term* (*tree*)
	|	*term*
	|	*nonterm*

C.W. Fraser, D.R. Hanson, T.A. Proebsting: *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems, vol. 1, no. 3, 1992

Tree parsing: Analysis

- Complexity: O(n), with n the number of AST nodes (similar to maximal munch)
- Optimality: Finds optimal derivation
- Tree parsing is in use in many compilers (automatic generation is a plus)
  - Optimizations: Dynamic programming from compile-time to compile-compile-time
- Optimality is restricted to AST representation: DAGs not supported



Code selection: Further reading

- Code selection for DAGs
- NP-completeness: Reduction to SAT
  - Use a tree-grammar that contains basic logic functions (AND, OR, …)
  - Covering a DAG (boolean function) can be mapped to the SAT problem
- Simple approaches to provide support for DAGs

Where are we?

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

11. Register allocation

- Introduction
- Local register allocation
- Global: Linear scan
- Global: Graph coloring
- Others

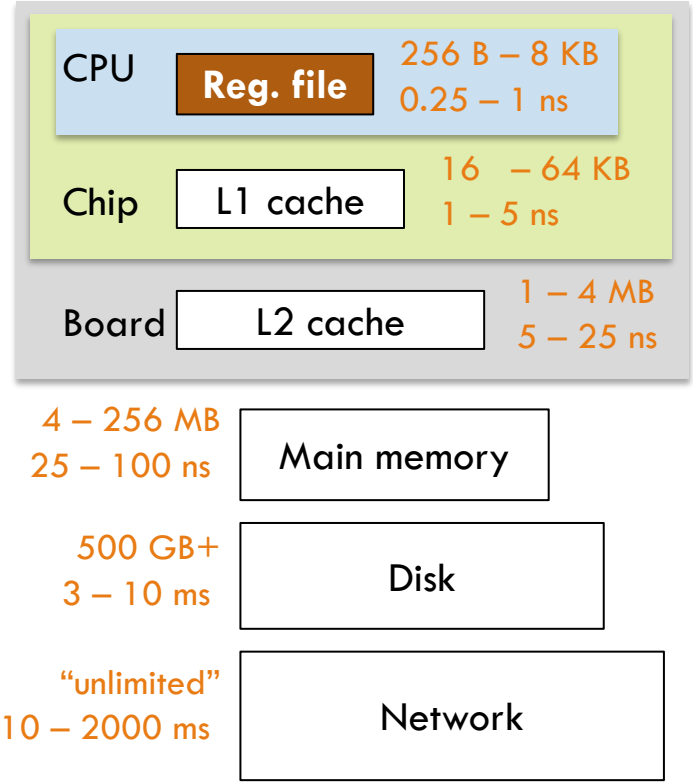
Backend: Recall

- Code selection: Uses pattern matching
  - Map IR code to assembly instructions (assume fixed code shape, i.e., IR is “optimal”)
  - ➔ For complexity reasons, assume infinite registers
- **Register allocation**
  - **Assign registers to variables (change storage pattern, i.e., inserts load/stores)**
- Scheduling
  - Reorder operations to hide latencies, assume a fixed program (set of operations)

Register allocation: Intuition

- ISA instructions take registers as arguments
- If not in register: Need to load from memory
- Registers are scarce resources ➔ **Use wisely**

- **Problem:** For a program with n virtual registers, how can we optimally use k physical registers? (interesting when n >> k)



Register allocation: Intuition (2)

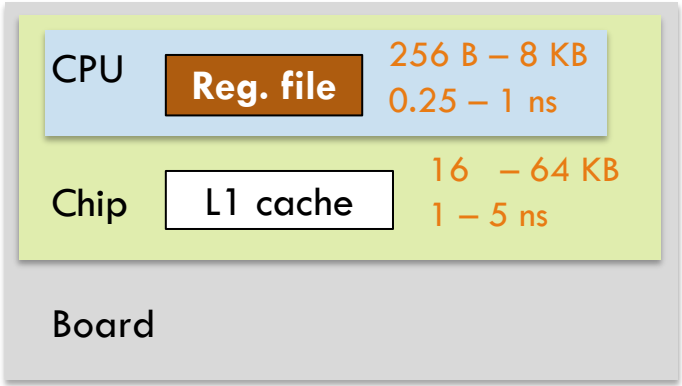
- **Example:** ISA with 4 registers

```
      x = a + b
Loop: y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

```
      ADD R1, R2, R3
      STR R2, @ADR2
Loop: ADD R4, R1, R4
      STR R1, @ADR1
      LD  R1, @z
      SLT R2, R4, R1
      LD  R1, @ADR1
      BEQ R2, Loop
      LD  R2, @ADR2
```

```
      ADD R1, R2, R3
Loop: ADD R4, R1, R4
      SLT R6, R4, R5
      BEQ R6, Loop
```

```
      ADD R1, R2, R3
      STR R2, @ADR2
      STR R3, @ADR3
      LD  R3, @z
Loop: ADD R4, R1, R4
      SLT R2, R4, R3
      BEQ R2, Loop
      LD  R2, @ADR2
      LD  R3, @ADR3
```



No memory access in the loop
➔ **Speedup 6x!!**

In general, a good allocation can improve performance by **an order of magnitude**

Challenges for register allocation

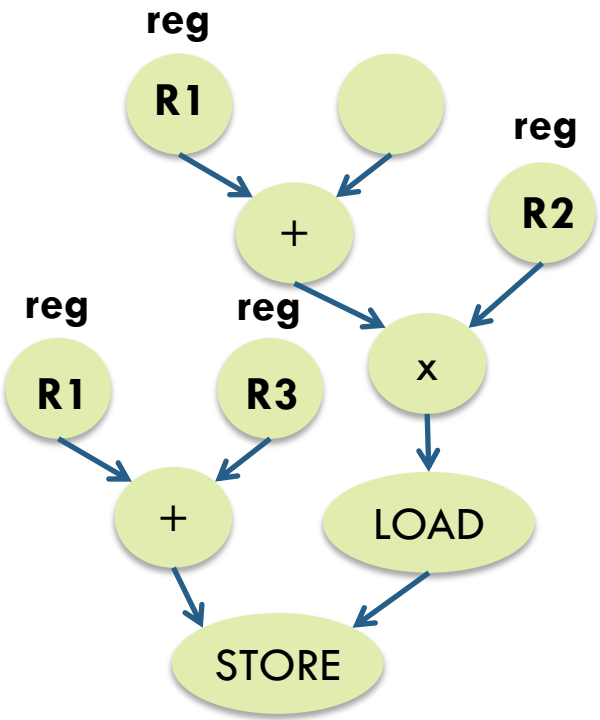
- Scarcity: Usually more variables/virtual registers than registers ➔ Need a way to share registers
- Registers & architecture classes
  - x86: Originally 4 general-purpose registers and many specialized
  - MIPS: 24 general-purpose and 8 special purpose registers
  - Some registers can be used together (2x32-bit register used as 1x64-bit register)
  - Some operations enforce the use of a register (accumulator, CX in x86 for loop count)
  - Conventions (recall):
    - Registers reserved for program counter, stack/frame pointer
    - Registers reserved to pass arguments to functions

Register allocation: Definitions

Def. **Register allocation** is the process of deciding which values/variables/virtual registers from a program are to be stored in a register to obtain an optimal register utilization. (It also manages data transfer in and out of registers)

Def. **Register assignment** is the process of actually selecting a physical register for a virtual register.

As with analysis & optimization: Distinguish among local (basic-block), global (function) & inter-procedural



Allocation vs. Assignment

- For different architectures, different effort
  - Assignment is trivial for RISC architectures with homogeneous registers
  - More difficult for architectures with special registers (CISC, DSP)
- Processes feature different complexity
- **Distinction seldom used today:** "register allocation" usually refers to both processes

Local Allocation
- Simplified cases ⇒ O(n)
- Real cases ⇒ NP-Complete

Global Allocation
- NP-Complete for 1 register
- NP-Complete for *k* registers

(most sub-problems are NPC, too)

Local Assignment
- Single size, no spilling ⇒ O(n)
- Two sizes ⇒ NP-Complete

Global Assignment
- NP-Complete

Source: Keith D. Cooper, Ken Kennedy & Linda Torczon

Spill code

- If there are K registers available and the program requires more than K registers at a given point
  ➔ Need to store values temporarily in memory
- Stored values need to be loaded later before they are used again

```
Loop: ADD R4, R1, R4
      STR R1, @ADR1
      LD  R1, @z
      SLT R2, R4, R1
      LD  R1, @ADR1
      BEQ R2, Loop
      LD  R2, @ADR2
```

Def. **Spill code** refers to additional instructions added to the program for storing and re-loading values to and from memory

- Spill code is implemented with load and store instructions
  ➔ Minimizing spilled registers is important for register allocation

Simple allocation: No allocation

- Example: LLVM IR

```
a = x + x < y;
b = x + x == y;
c = x + x > y;
```

```
%2 = load i32* %x, align 4
%3 = load i32* %x, align 4
%4 = add nsw i32 %2, %3
%5 = load i32* %y, align 4
%6 = icmp slt i32 %4, %5
%7 = zext i1 %6 to i32
store i32 %7, i32* %a, align 4
%8 = load i32* %x, align 4
%9 = load i32* %x, align 4
%10 = add nsw i32 %8, %9
%11 = load i32* %y, align 4
%12 = icmp eq i32 %10, %11
%13 = zext i1 %12 to i32
store i32 %13, i32* %b, align 4
%14 = load i32* %x, align 4
%15 = load i32* %x, align 4
%16 = add nsw i32 %14, %15
%17 = load i32* %y, align 4
%18 = icmp sgt i32 %16, %17
%19 = zext i1 %18 to i32
store i32 %19, i32* %c, align 4
```

- Advantage
  - Very simple and efficient to implement
  - Do not worry about registers and conventions
- Disadvantage
  - Extreme low performance
  - Unacceptable for real compilers

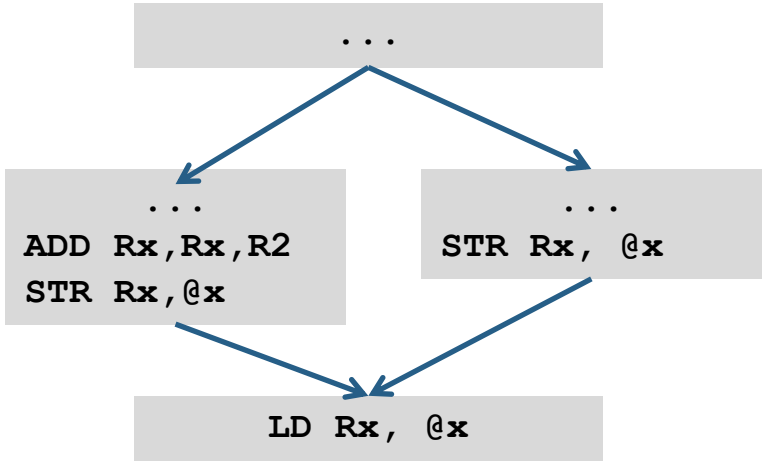
11. Register allocation

- Introduction
- Local register allocation
- Global: Linear scan
- Global: Graph coloring
- Others

Local allocation

- Focus only on registers inside a basic-block
- Easy to get good results

- Problems
  - Inefficiencies can arise at the basic-block boundaries
  - Need to introduce fixes/optimizations
  - It is possible to iterate to improve allocation



Top-down approach

- Intuition: Assign registers to the values that are more frequently used
  - Count uses of values
  - Make priority list of values
  - Assign physical registers to values according to the priority list
  - If no more physical registers available ➔ Introduce spill code
- Motivation similar to that of the **register** keyword in C

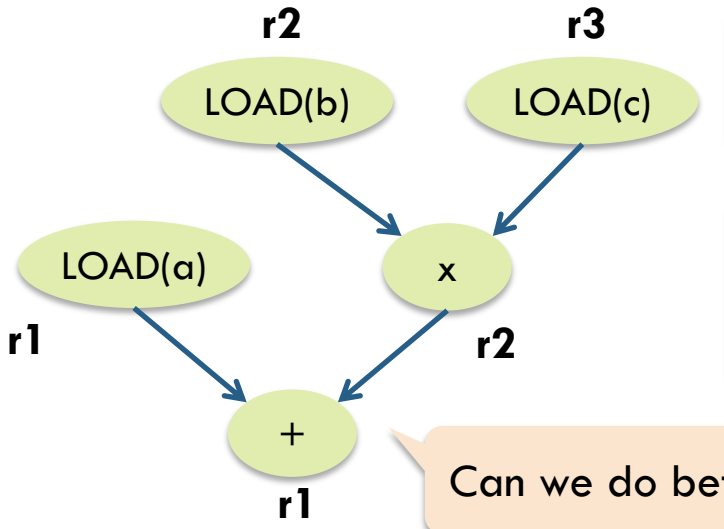
- Drawbacks
  - Tend to use more registers than actually needed
  - Block registers for the entire basic-block (does not consider **liveness,** more on this later)

Bottom-up approach

- Intuition: Assign registers on demand as they are used. Keep values used soon in registers
  - Start with empty list of used registers
  - Walk the AST (assembly) and assign registers on demand (add to list of used regs)
  - If no register is available free one (replacement)
    - Spill the value that is used the farthest in the future
- Algorithm similar to page replacement

Example: Bottom-up allocation for trees (leaves-root)

- Motivation: Work directly on the ASTs pre-processed by the code selector
- Good as academic example, but far from optimal



Children walked in arbitrary order

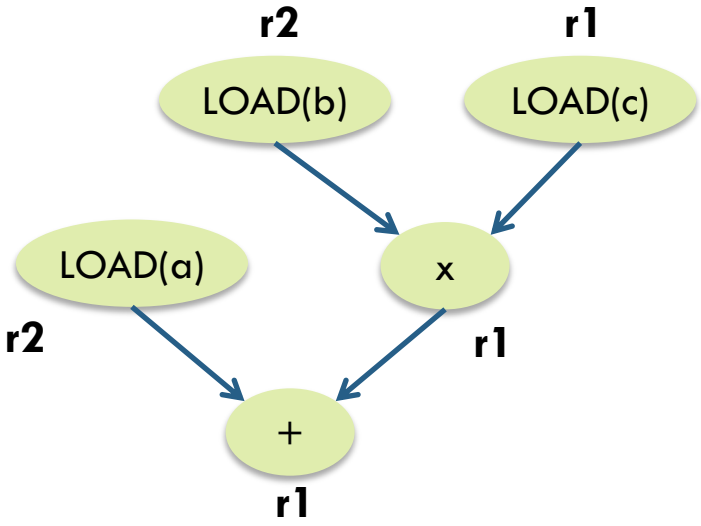
n is used to re-use registers as much as possible

Can we do better?

```
n = 0, A = ∅
Proc        SimpleAlloc(T_r)
  // T_r root of a subtree of AST T
Output      Assignment A (r_n, node)
foreach child u of T_r
                SimpleAlloc(u)
foreach child u of T_r
                n = n - 1
n = n + 1
A = A U (r_n, T_r)
```

Example: Analysis

- Only two registers if children are walked right to left
  - Results depend on the **order** sub-trees are evaluated
- Solution
  - **Dynamic programming** approach to determine the **register needs** of each sub-tree
  - SimpleAlloc processes first sub-trees with higher register needs



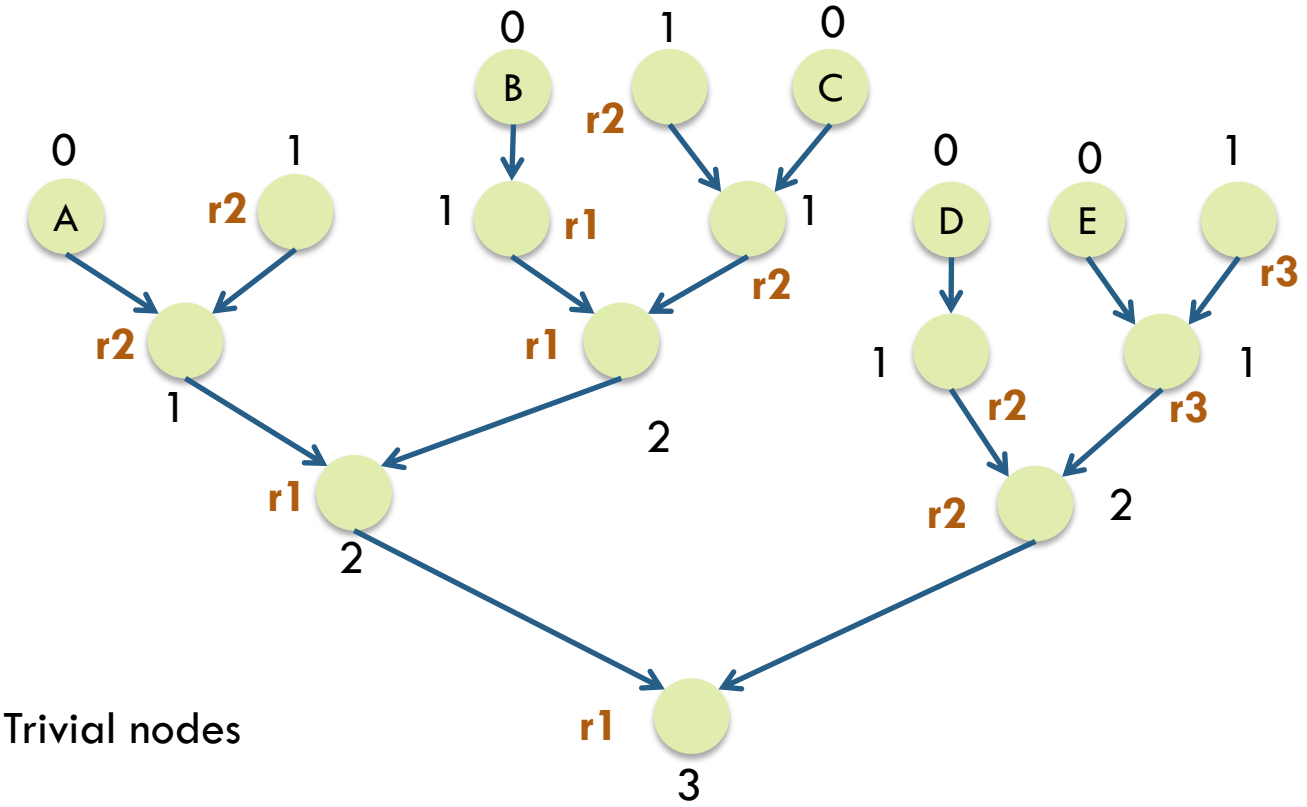
DP for register needs

```
Proc  Label(TreeNode Tr)
foreach  child u of Tr
   Label(u)
if Tr is trivial then need[Tr] = 0
else if Tr  has children l, r then
   if need[l] == need[r] then need[Tr] = need[l] + 1
   else need[Tr] = max(1, need[l], need[r])
else if Tr  has one child u then need[Tr] = max(1, need[u])
else if Tr has no children then need[Tr] = 1
```

need[x] stores how many register are required by the sub-tree with root at x

Trivial means that the register is already known or that it is a constant encoded in the instruction (immediate)

Example: Register needs



A, ..., E = Trivial nodes

11. Register allocation

- Introduction
- Local register allocation
- Global: Linear scan
- Global: Graph coloring
- Others

Global register allocation

- Recall goal: Try to hold as many variables in registers as possible
  - Reduces memory writes/reads
- Register consistency
  - At each program point, each variable must be in the same location (**not** necessarily each variable in same location across program points, as in top-down)
  - At each program point, each register holds at most one **live** variable (i.e., variables can share a register if not live at the same time)

```
%2 = load i32* %x, align 4
%3 = load i32* %x, align 4
%4 = add nsw i32 %2, %3
%5 = load i32* %y, align 4
%6 = icmp slt i32 %4, %5
%7 = zext i1 %6 to i32
store i32 %7, i32* %a, align 4
%8 = load i32* %x, align 4
%9 = load i32* %x, align 4
%10 = add nsw i32 %8, %9
%11 = load i32* %y, align 4
%12 = icmp eq i32 %10, %11
%13 = zext i1 %12 to i32
store i32 %13, i32* %b, align 4
%14 = load i32* %x, align 4
%15 = load i32* %x, align 4
%16 = add nsw i32 %14, %15
%17 = load i32* %y, align 4
%18 = icmp sgt i32 %16, %17
%19 = zext i1 %18 to i32
store i32 %19, i32* %c, align 4
```

Live interval

- Recall: A variable is **live** at a program point if its value is read afterwards before it is written again

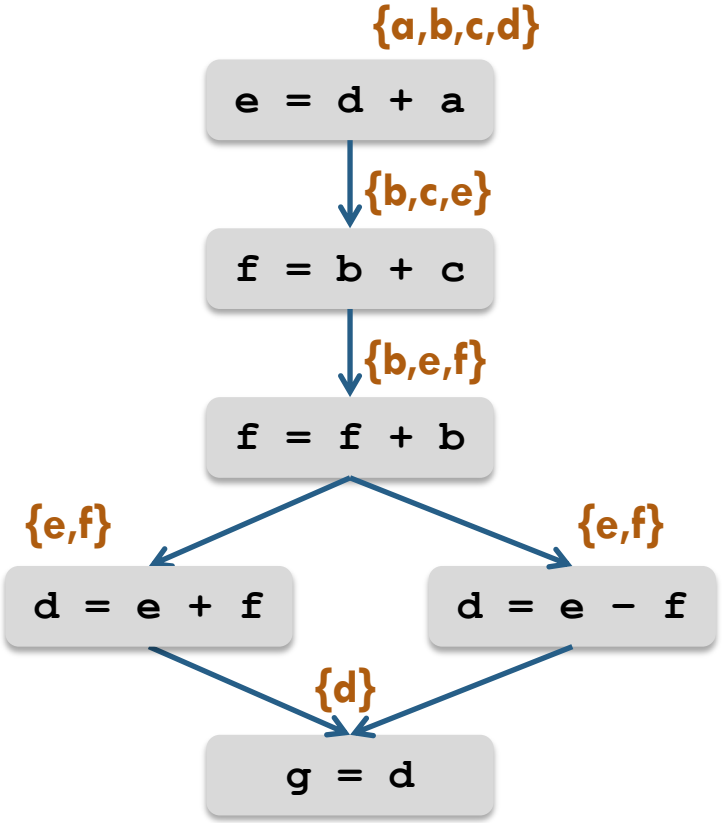
> Def. the **live interval** of a variable is the smallest range of the **IR** that contains all program points at which the variable is live

- Live intervals are a property of the IR not of the control flow graph
- Intuition: Variables with non-overlapping intervals can share the same register

Live intervals: Example

a	b	c	d	e	f	g

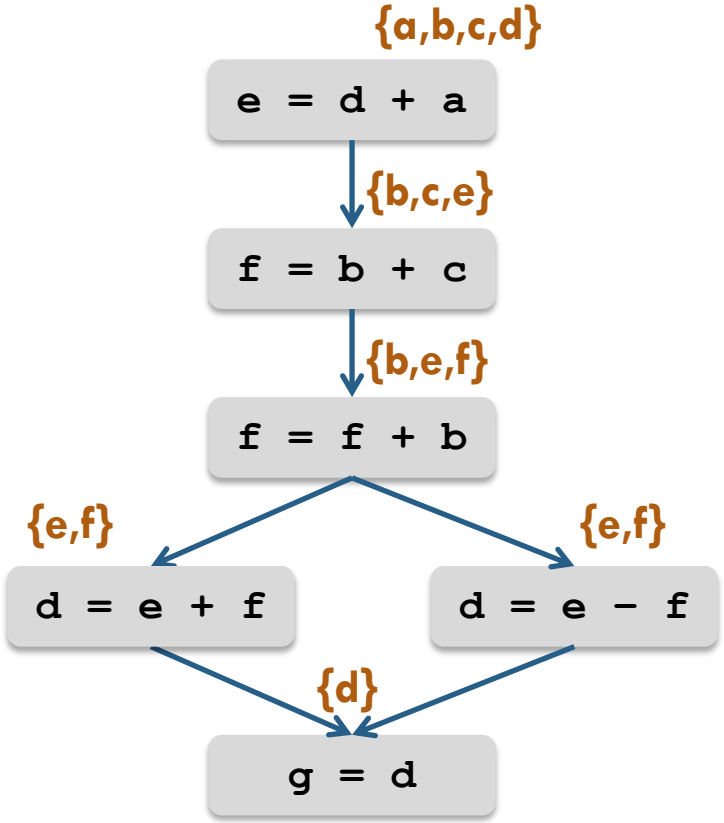
```
    e = d + a
    f = b + c
    f = f + b
    if e goto L0
    d = e + f
    goto L1
L0:d = e - f
L1:g = d
```



Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

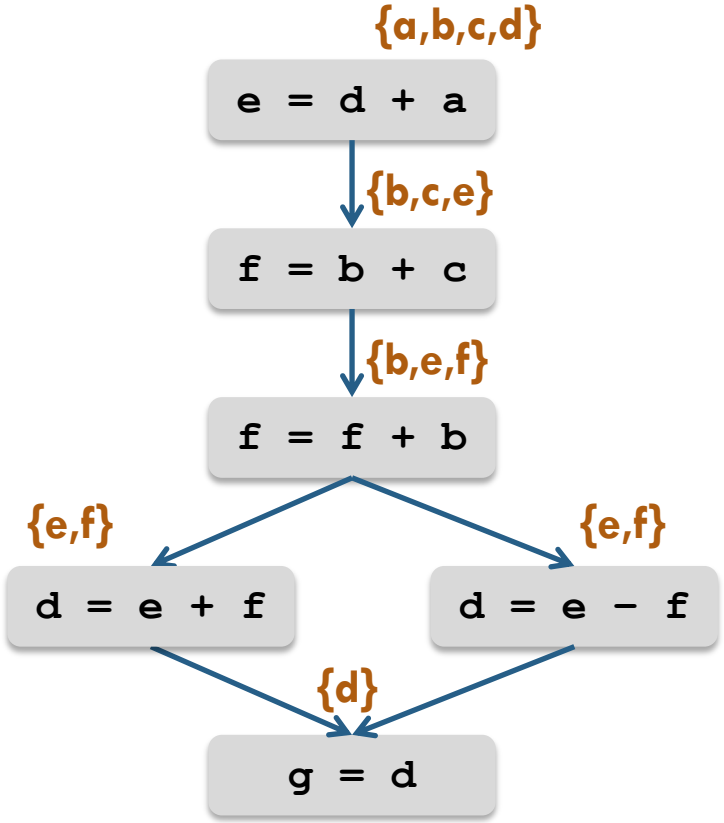
Live intervals: Example

	a	b	c	d	e	f	g
`e = d + a`							
`f = b + c`							
`f = f + b`							
`if e goto L0`							
`d = e + f`							
`goto L1`							
`L0:d = e - f`							
`L1:g = d`							



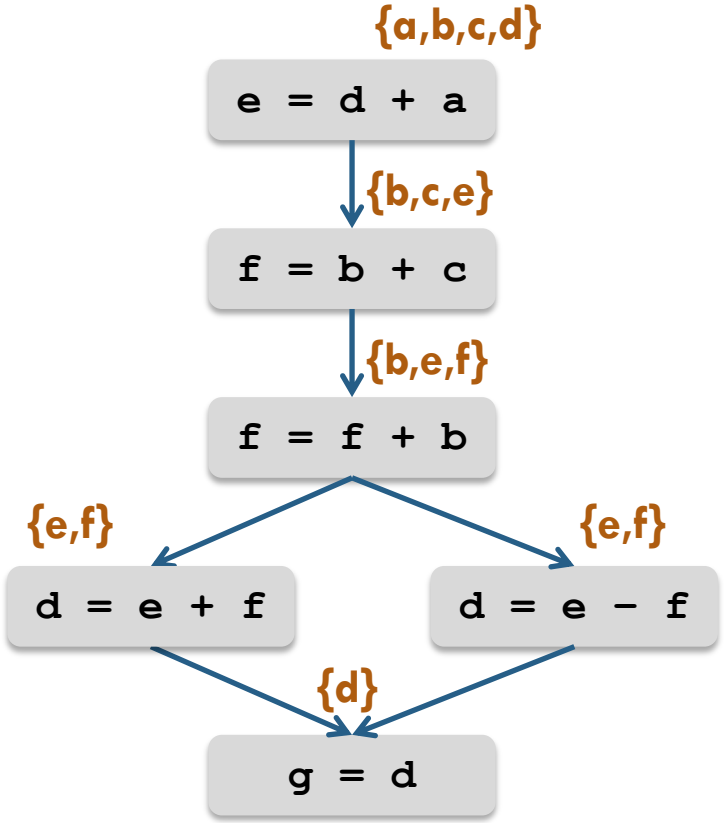
Live intervals: Example

	a	b	c	d	e	f	g
`e = d + a`	■	■					
`f = b + c`		■					
`f = f + b`		■					
`if e goto L0`							
`d = e + f`							
`goto L1`							
`L0:d = e - f`							
`L1:g = d`							



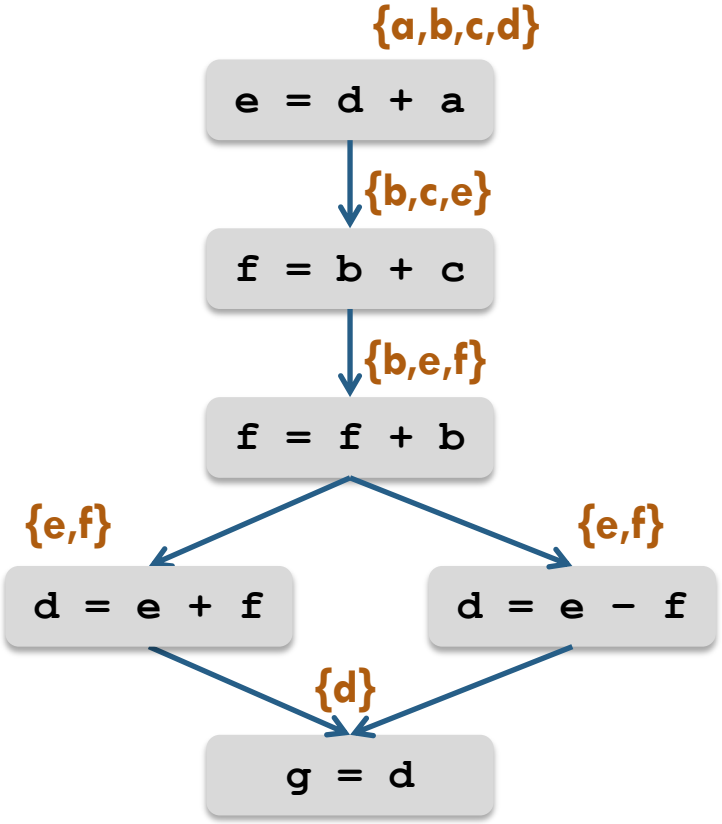
Live intervals: Example

	a	b	c	d	e	f	g
`e = d + a`	■	■	■				
`f = b + c`		■	■				
`f = f + b`		■					
`if e goto L0`							
`d = e + f`							
`goto L1`							
`L0:d = e - f`							
`L1:g = d`							



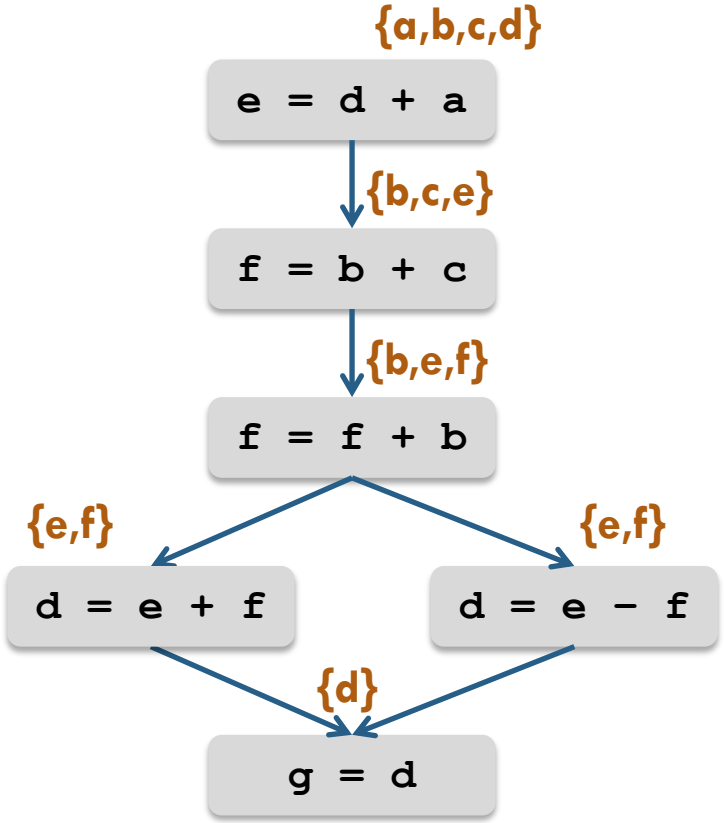
Live intervals: Example

	a	b	c	d	e	f	g
`e = d + a`	■	■	■	■			
`f = b + c`		■	■				
`f = f + b`		■					
`if e goto L0`							
`d = e + f`							
`goto L1`				■			
`L0:d = e - f`							
`L1:g = d`				■			



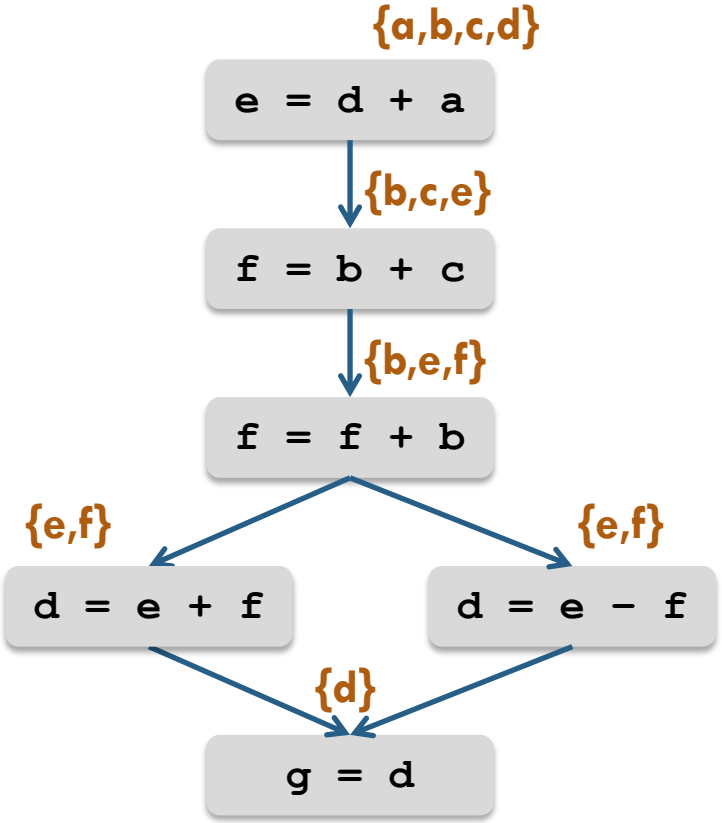
Live intervals: Example

	a	b	c	d	e	f	g
`e = d + a`	■	■	■	■			
`f = b + c`		■	■	■			
`f = f + b`		■		■			
`if e goto L0`				■			
`d = e + f`				■			
`goto L1`				■			
`L0:d = e - f`				■			
`L1:g = d`				■			



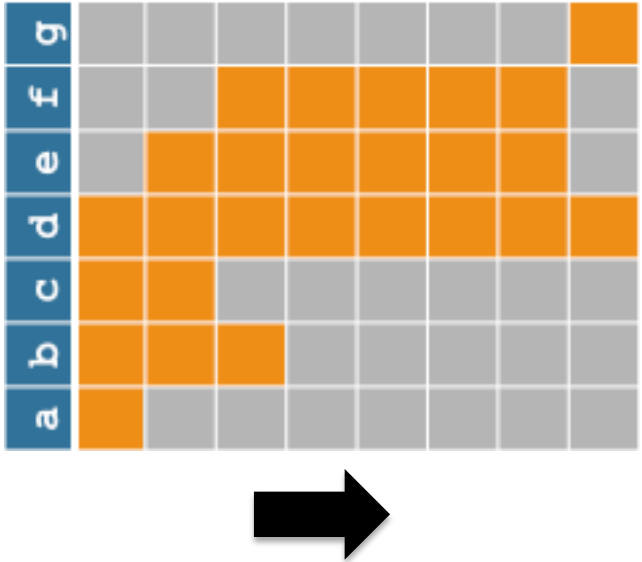
Live intervals: Example

	a	b	c	d	e	f	g
`e = d + a`							
`f = b + c`							
`f = f + b`							
`if e goto L0`							
`d = e + f`							
`goto L1`							
`L0:d = e - f`							
`L1:g = d`							



Linear scan algorithm (also left-edge algorithm)

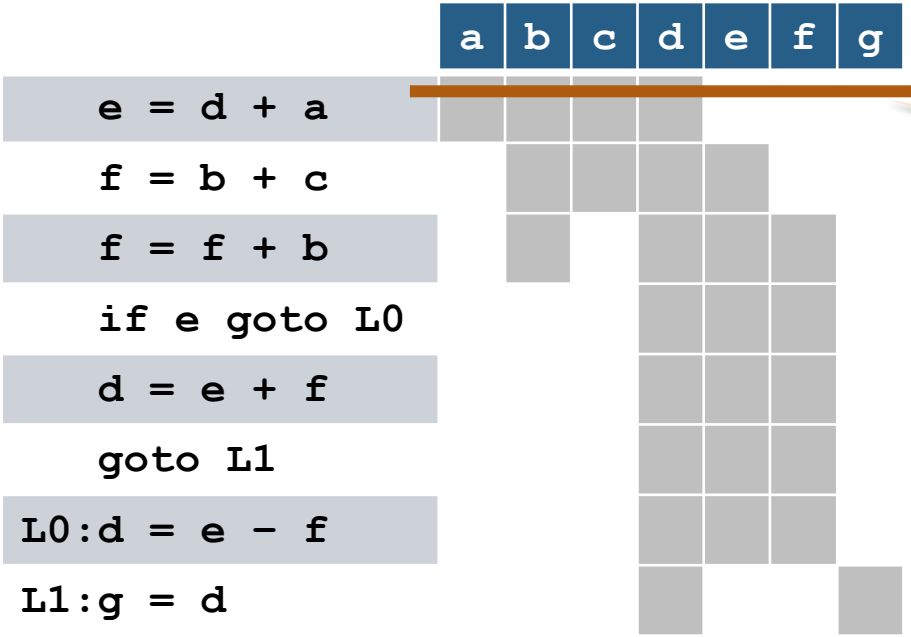
- Given the live intervals assigns registers
  - Walk intervals left to right
  - Start of interval: Assign free register
  - End of the interval: Release register
  - If not enough registers, spill to memory



- Analysis
  - Complexity: Linear (fast!), used in JIT compilers (Java Hotspot™)
  - Not optimal, intervals are too coarse (use **ranges** directly, see later)

Wimmer, Christian, and Michael Franz. "Linear scan register allocation on SSA form." *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010.

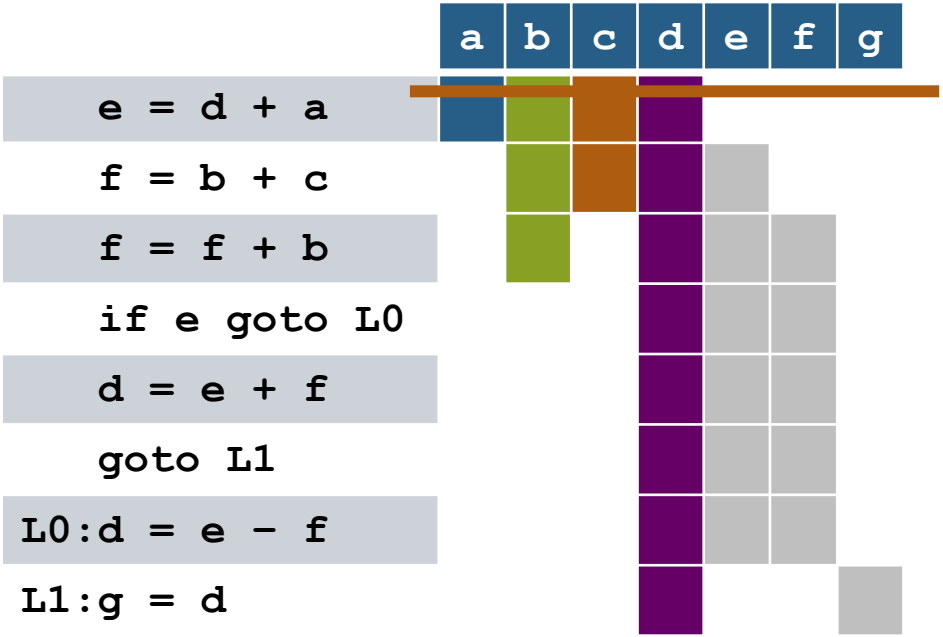
Register allocation for the example



Free registers

Four intervals start, pick one by one a free register from the list

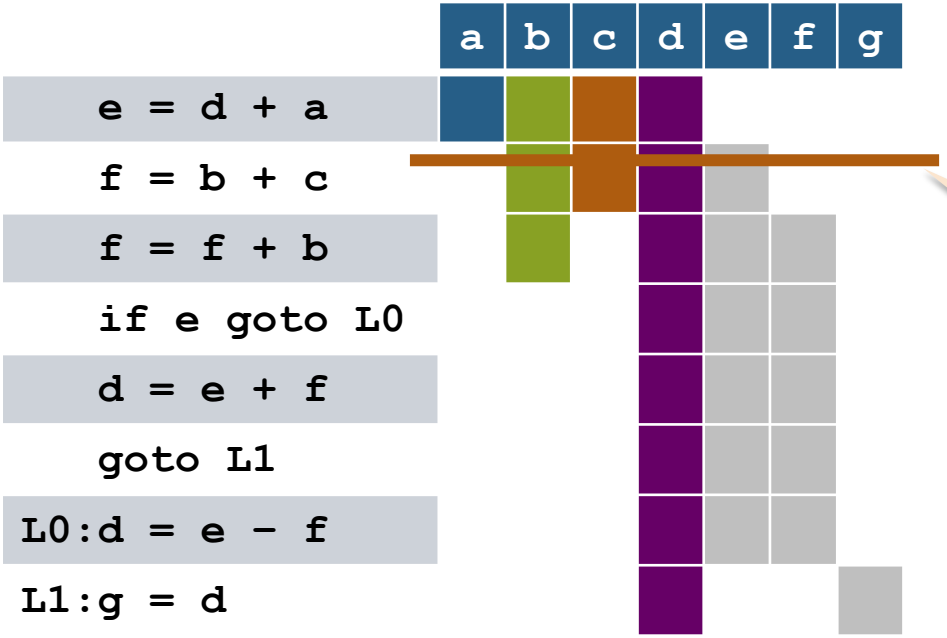
Register allocation for the example



Free registers

R1	R2	R3	R4

Register allocation for the example



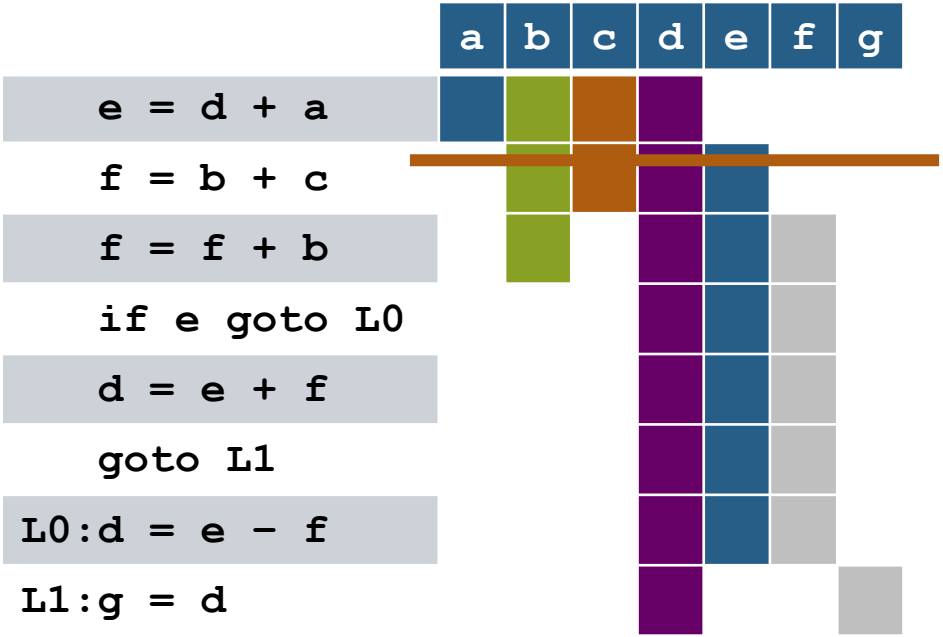
Free registers

R1	R2	R3	R4

Releases register for a, not live anymore

At the same time, e's interval starts and gets R1

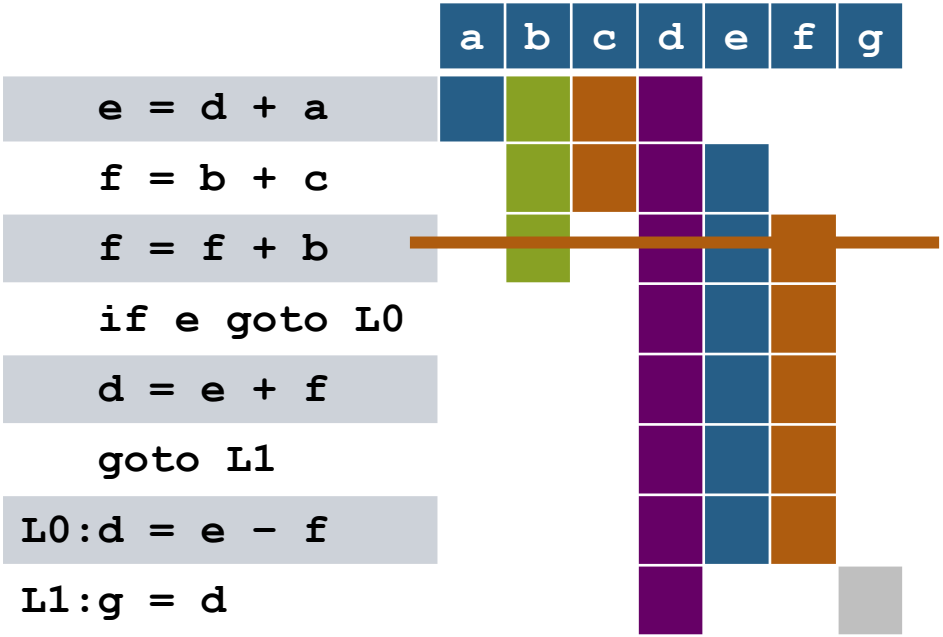
Register allocation for the example



Free registers

R1	R2	R3	R4

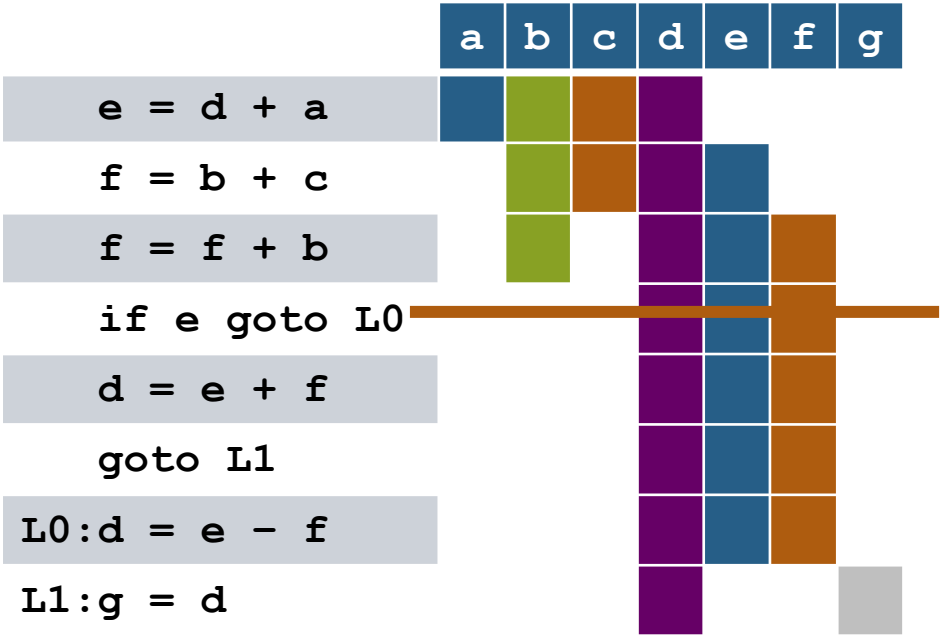
Register allocation for the example



Free registers

R1	R2	R3	R4

Register allocation for the example

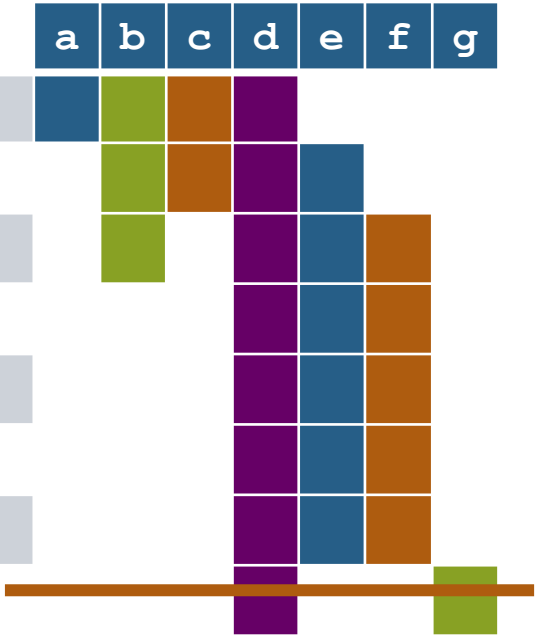


Free registers

R1	R2	R3	R4

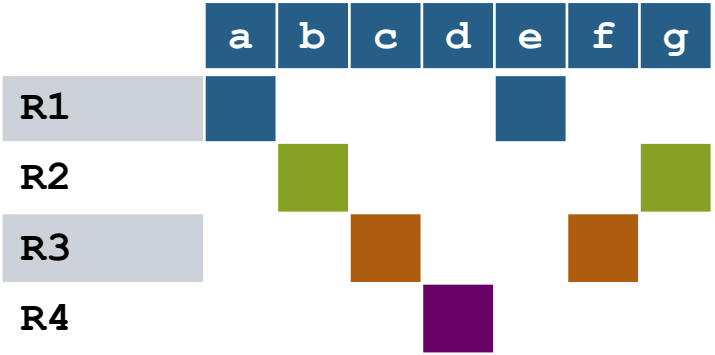
Register allocation for the example

```
    e = d + a
    f = b + c
    f = f + b
    if e goto L0
    d = e + f
    goto L1
L0:d = e - f
L1:g = d
```



Free registers

Allocation



11. Register allocation

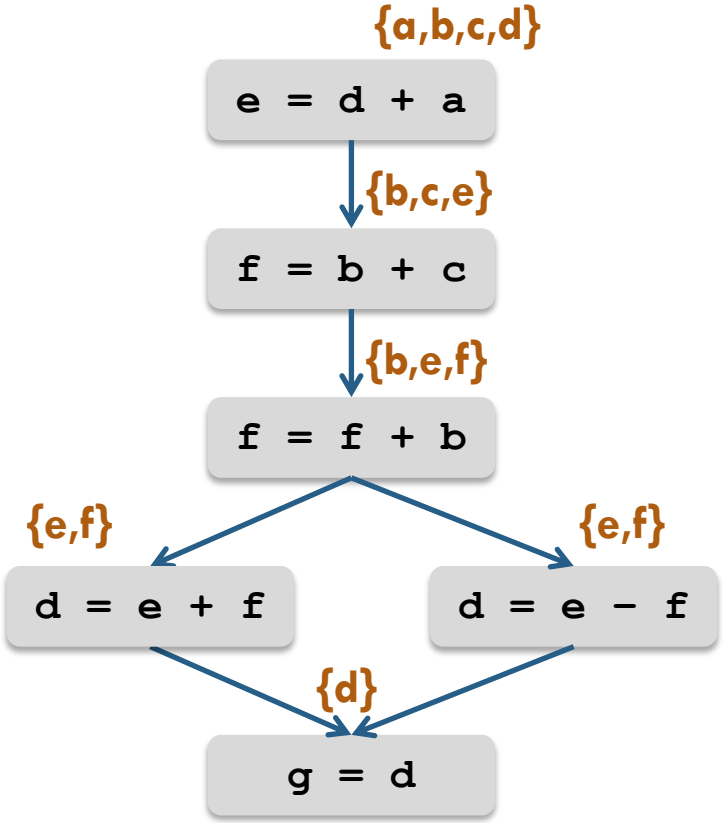
- Introduction
- Local register allocation
- Global: Linear scan
- Global: Graph coloring
- Others

Live ranges vs. live intervals

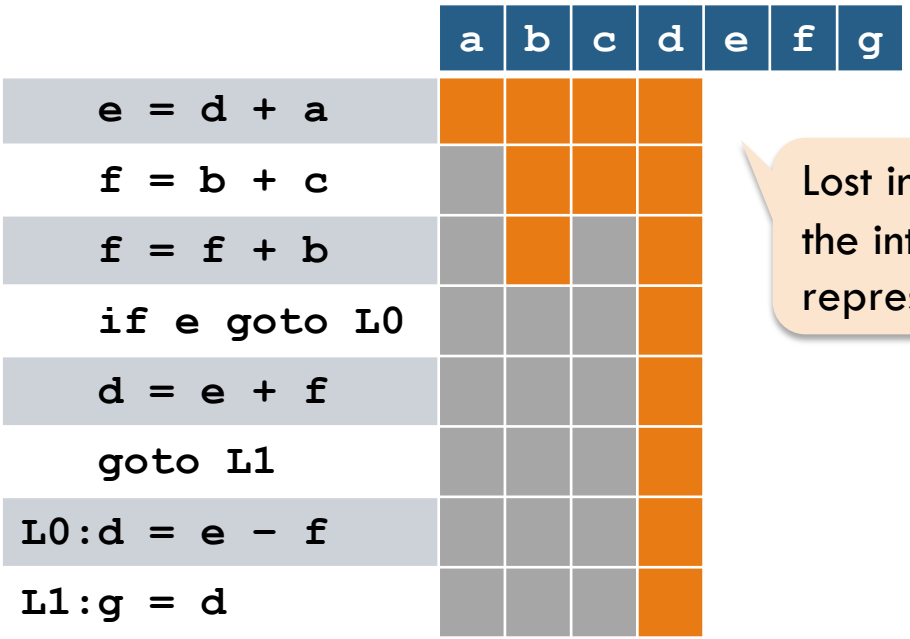
- A different approach to allocation using **live ranges** instead of **intervals**

Def. the **live interval** for a variable is the smallest range of the **IR** that contains all program points at which the variable is live

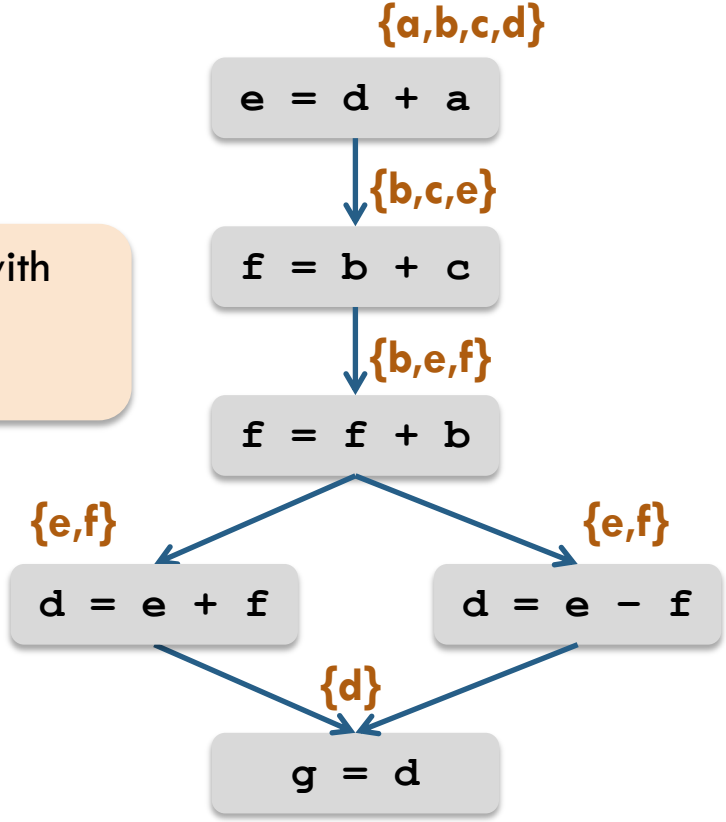
Def. the **live range** for a variable is the set of program points at which the variable is live



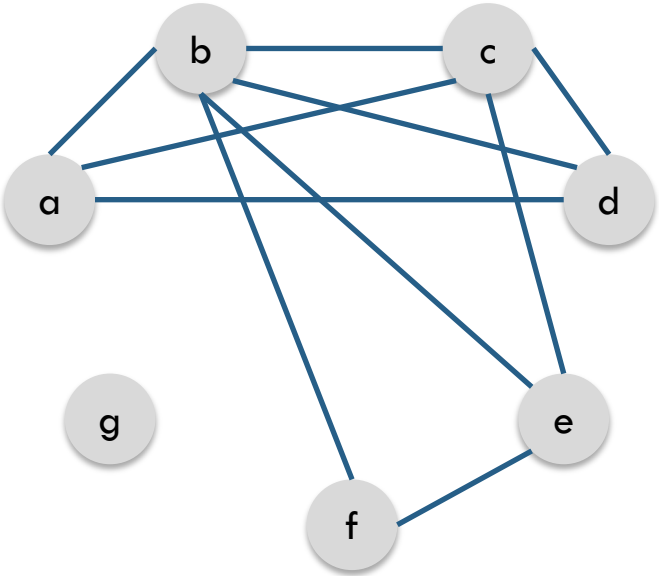
Live ranges vs. live intervals (2)



Lost information with the interval representation



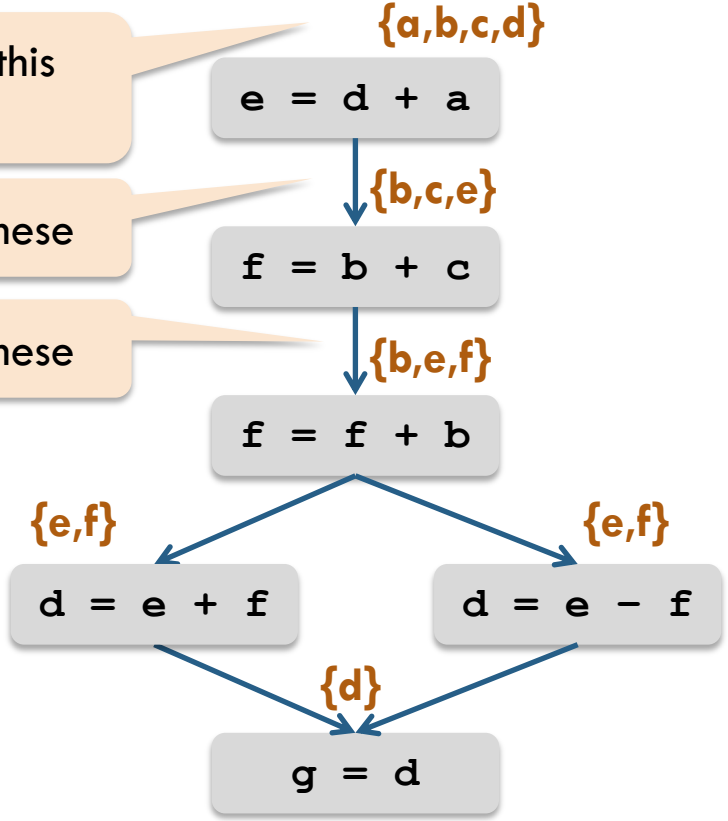
RA & graph coloring: Intuition



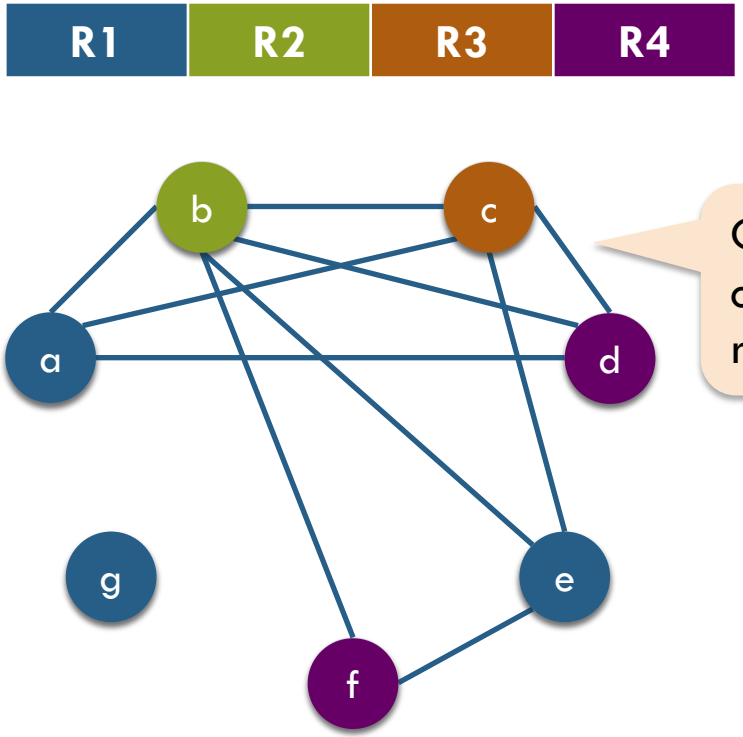
These variables cannot be at this point in same register

Neither these

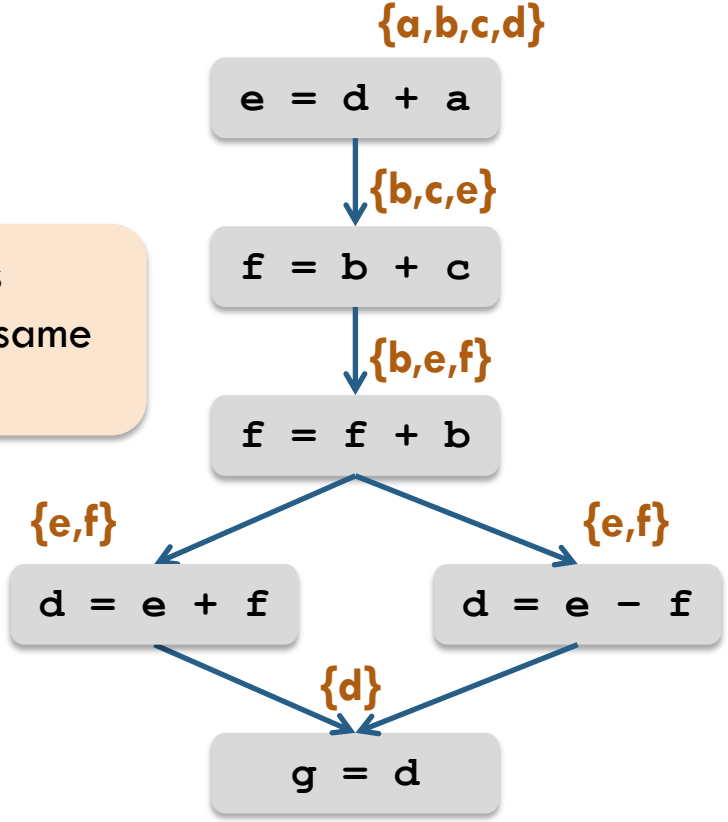
Neither these



RA & graph coloring: Intuition



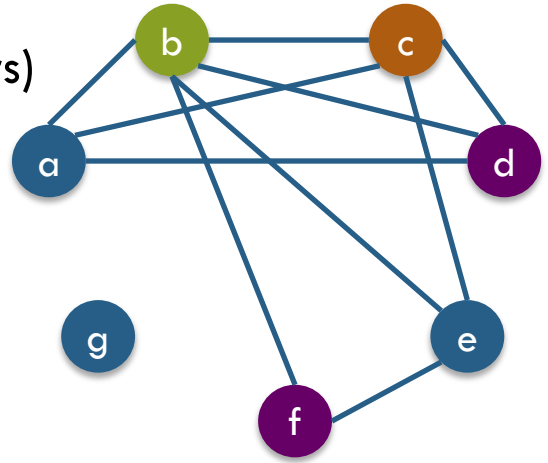
Connected nodes cannot be in the same register



RA & graph coloring

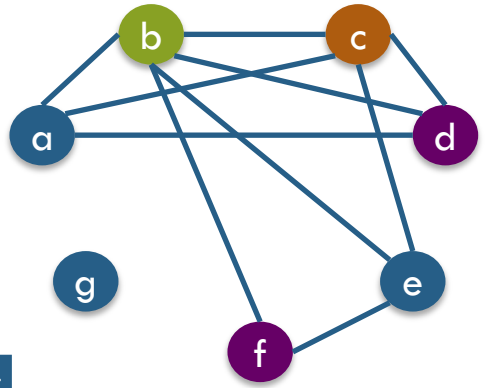
- History
  - Cocke 1971: Relation between register allocation and graph coloring
  - Chaitin 1981: Implementation in the IBM 370 PL/I compiler
  - Briggs 1992: Improvements (implemented in many compilers today, e.g., GCC)

- Finding the chromatic number (minimal number of colors) of a graph is NP complete: **Need a heuristic**
- Idea
  - K registers ➔ K colors
  - Try to color the graph with K colors (K-Coloring)
  - If failed: Insert spill code and try again



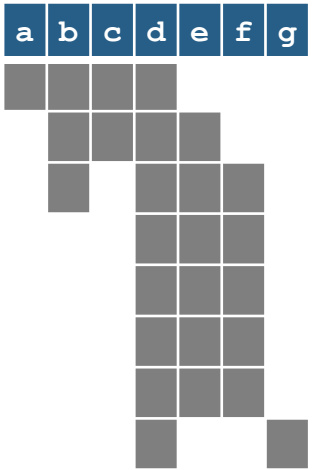
Graph coloring: Definitions

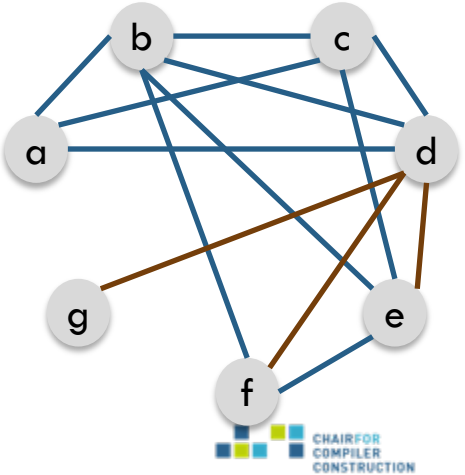
Def. Given a set of virtual registers V, a **register interference graph** (RIG) is an undirected graph G = (V,E) in which (u,v) ∈E iff u and v are simultaneously live at given program point (overlapping live ranges)



Def. An **interval graph** is an undirected graph G = (V,E) for which one can define a bijective function f: V → I, with I an ordered set of intervals, so that

(u,v) ∈E ⇔ f(u) ∩ f(v) ≠∅

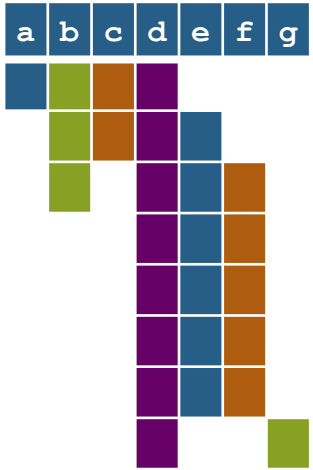


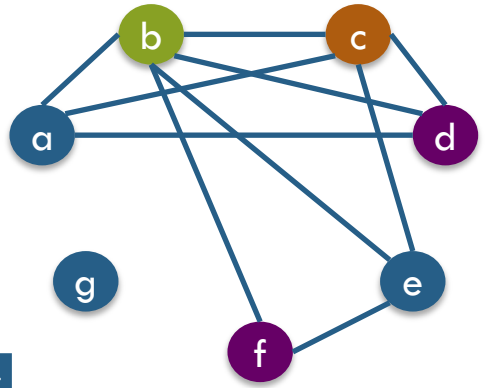


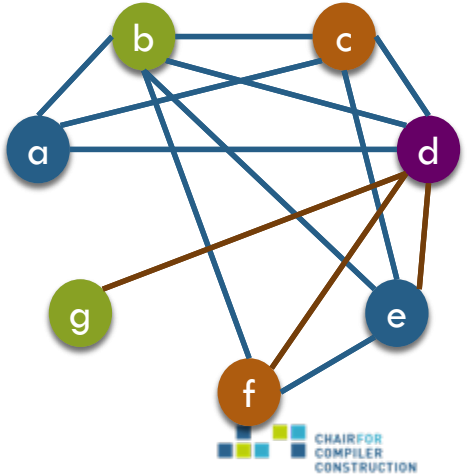
Linear scan contradicts NP completeness?

Def. An **interval graph** is an undirected graph G = (V,E) for which one can define a bijective function f: V → I, with I an ordered set of intervals, so that

$$(u,v) \in E \Leftrightarrow f(u) \cap f(v) \neq \emptyset$$







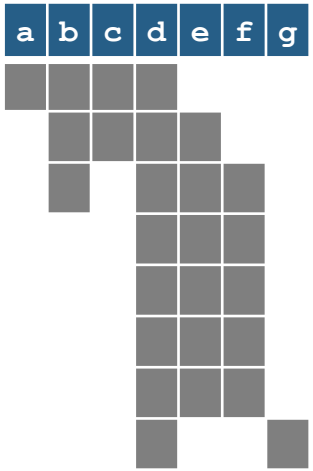
- Linear scan did a coloring in linear time
- It is known that interval graphs can be colored in linear time

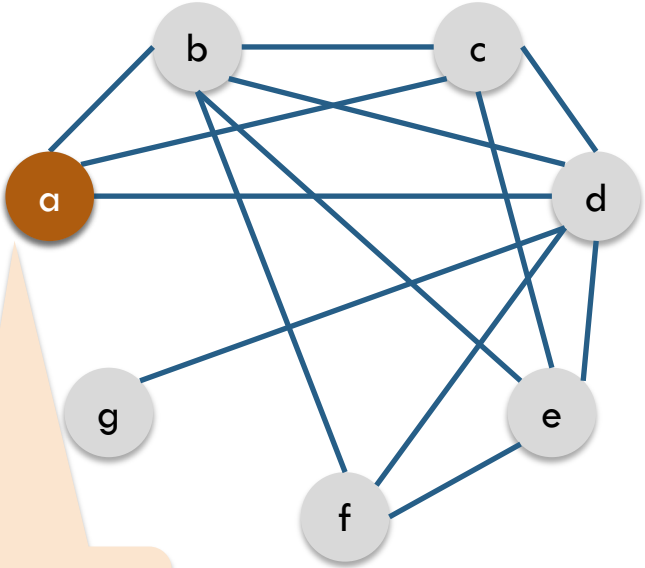
Chaitin's graph coloring algorithm

- Intuition: K-coloring
  - If a node X has less than K neighbors, remove it and color the remaining graph
  - Once the graph is colored, add X back to the graph: A color will be available
- Algorithm
  - **Build**: Construct the interference graph
  - **Simplify**: Find node X with deg(X) < K, remove it and recursively **simplify graph**
  - **Spill**: If there is no X, deg(X)<K, select a node for **potential** spill (whether the node is spilled, is determined later), remove it and try **simplify graph** again
  - **Select**: Once the graph is empty (via simplify/spill), select nodes and assign colors
    - Simplified node ➔ Color it. Spilled node ➔ Spill only if neighbors already have K colors
  - **Start over:** If spill was inserted, live ranges change ➔ re-build graph and re-color

Coloring example: Simplify

- Interval graph



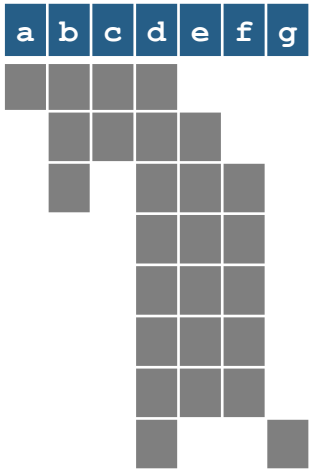


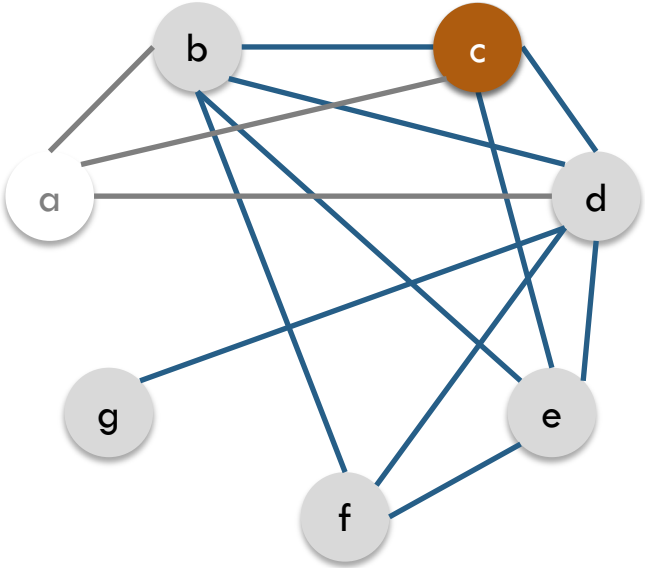
Registers

Node has less than 4 neighbors: Simplify!

Coloring example: Simplify

- Interval graph



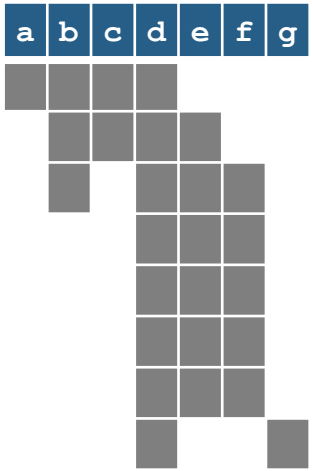


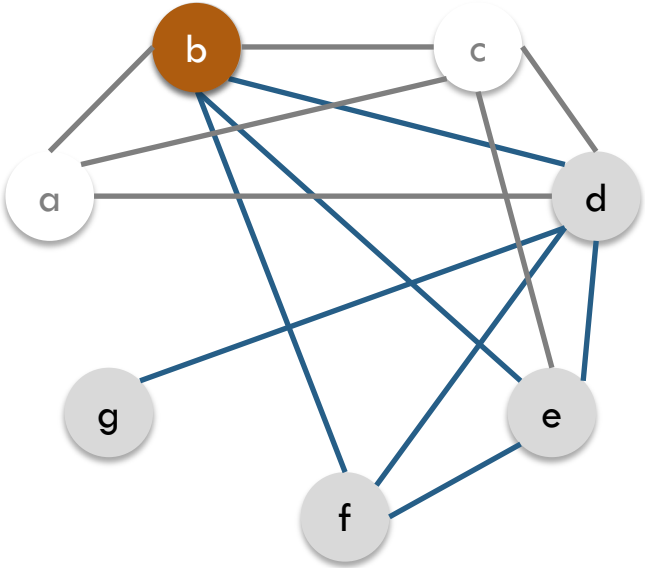
Registers

Node stack a

Coloring example: Simplify

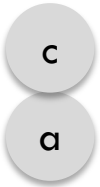
- Interval graph





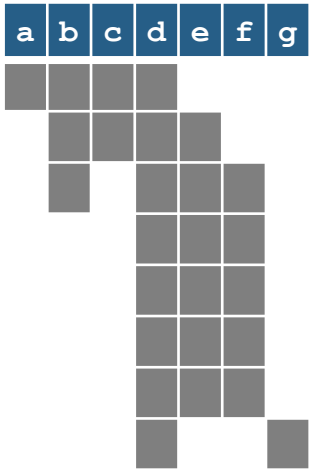
Registers

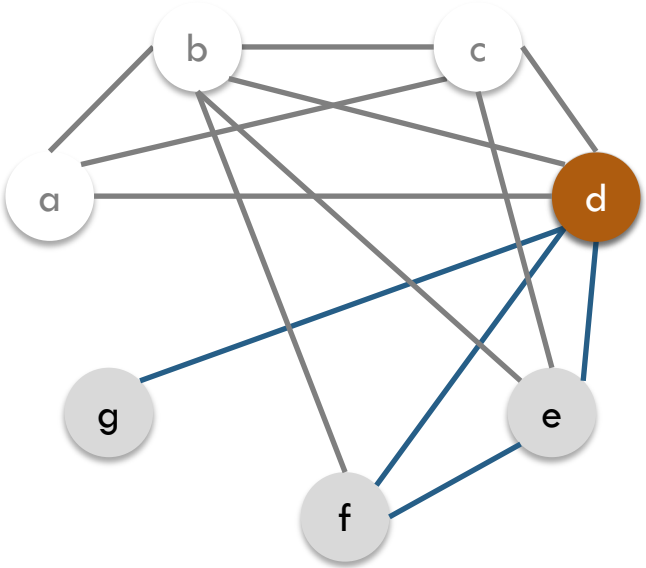
Node stack



Coloring example: Simplify

- Interval graph



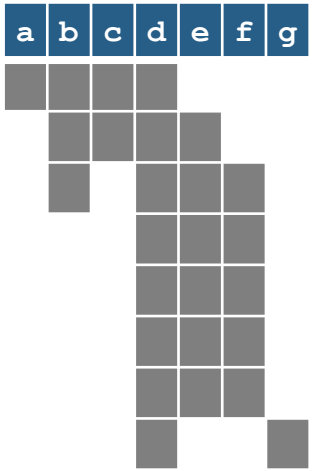


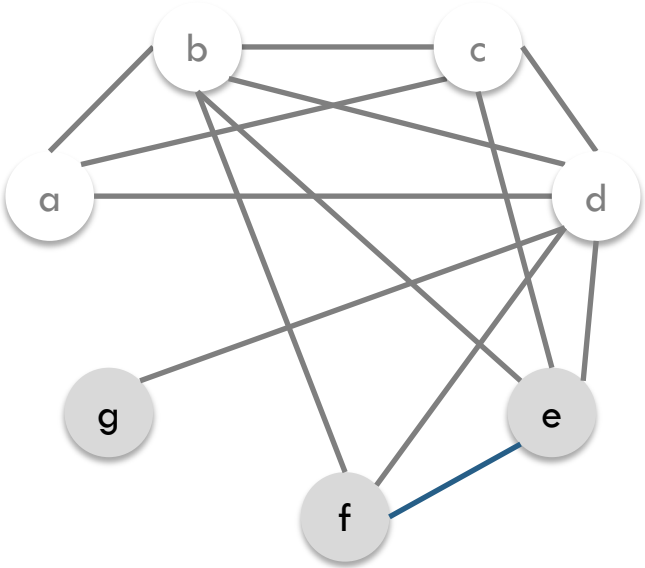
Registers

Node stack

Coloring example: Simplify

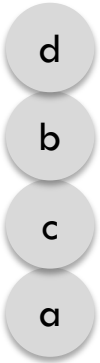
- Interval graph





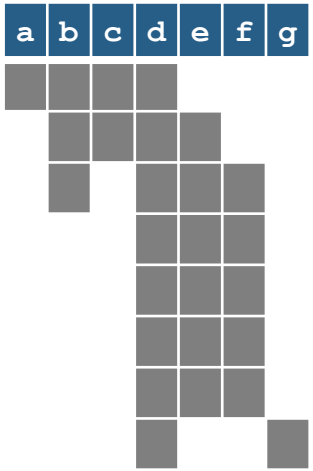
Registers

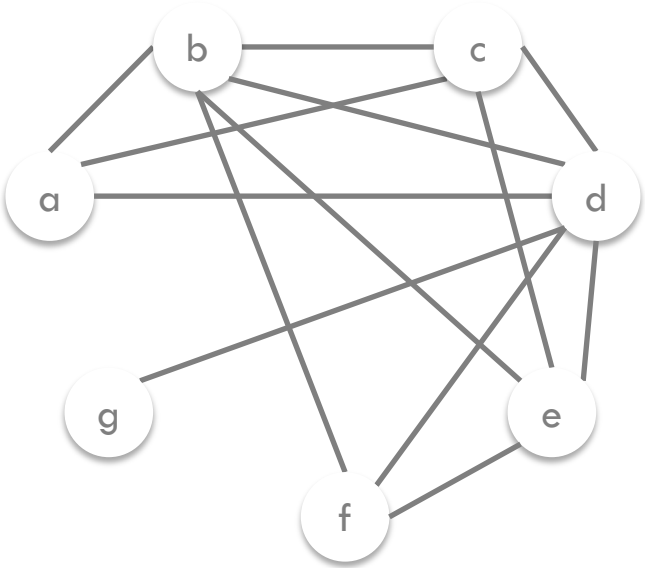
Node stack



Coloring example: Simplify

- Interval graph





Registers

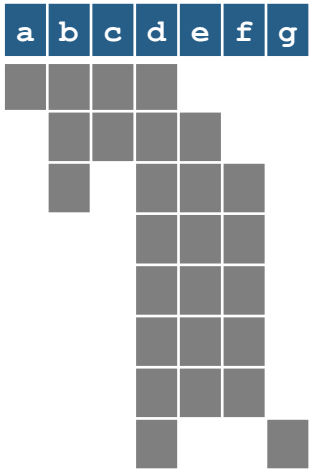
R1	R2	R3	R4

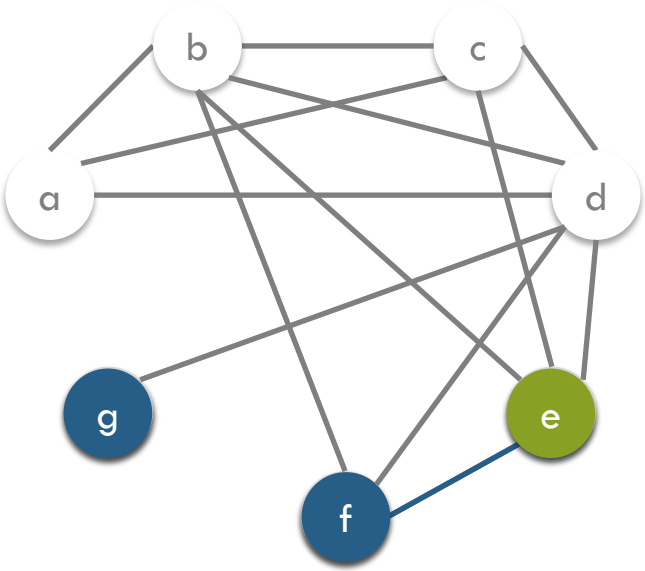
Node stack

f
e
g
d
b
c
a

Coloring example: Select

- Interval graph



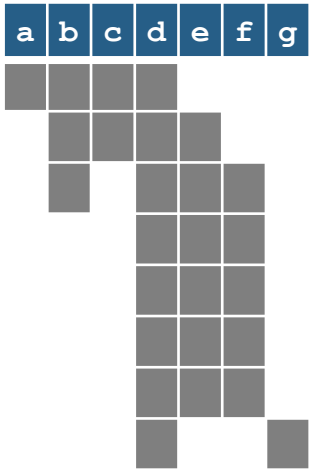


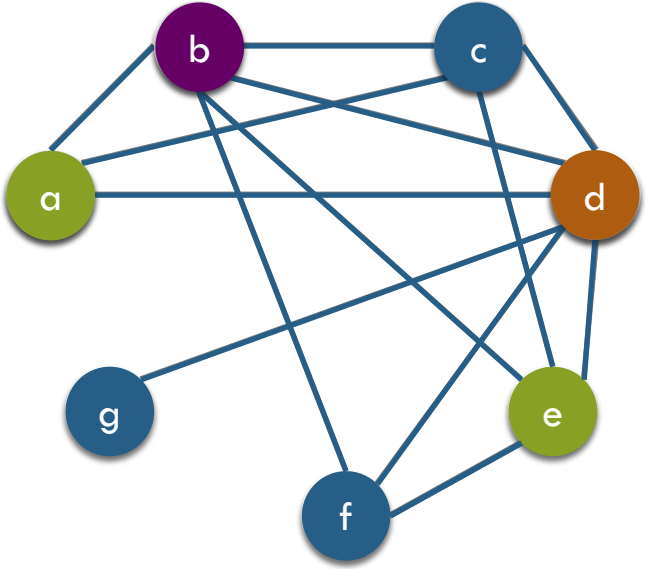
Registers

Node stack

Coloring example: Select

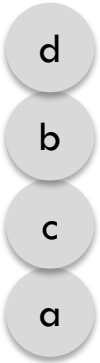
- Interval graph





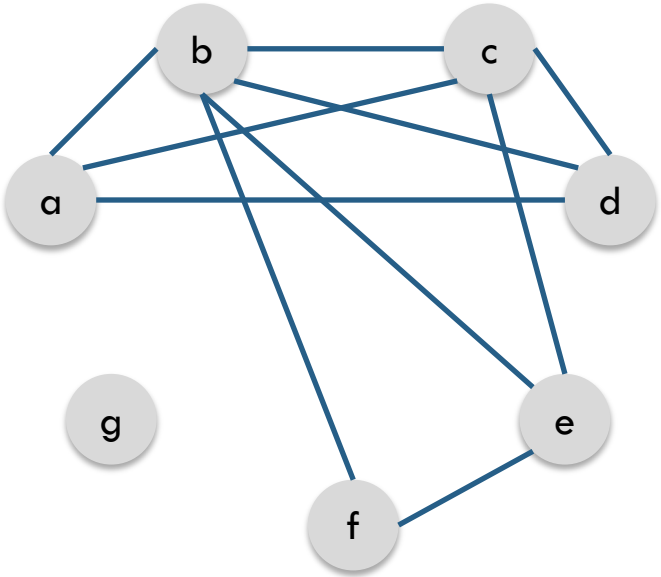
Registers

Node stack



Graph coloring: Example 2

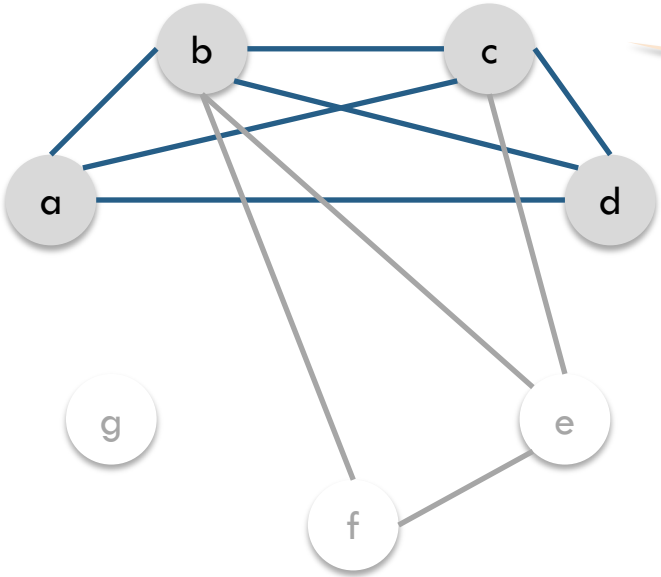
- Interference graph with 3 registers



Registers

Graph coloring: Example 2 – Simplify

- Interference graph with 3 registers



Registers

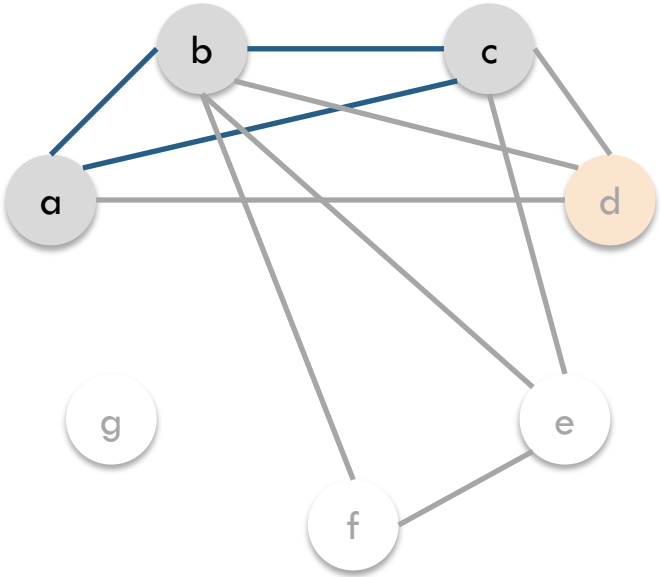
R1	R2	R3

Which one to simplify?

Node stack

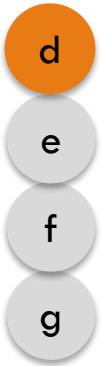
Graph coloring: Example 2 – Simplify

- Interference graph with 3 registers



Registers

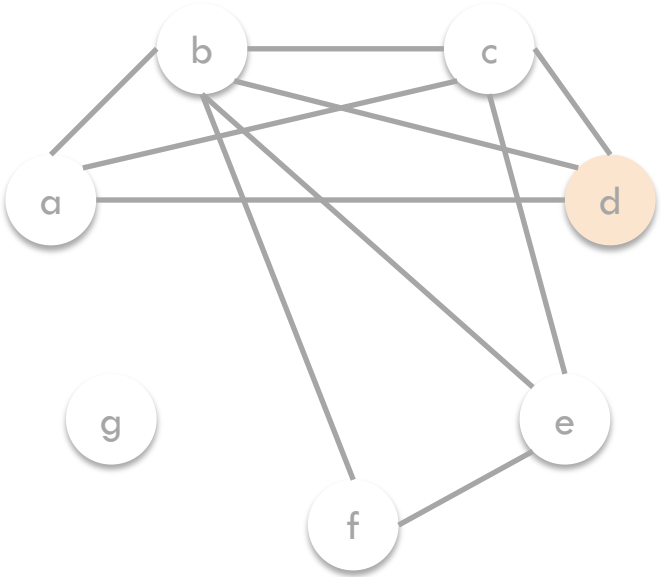
Potential spill

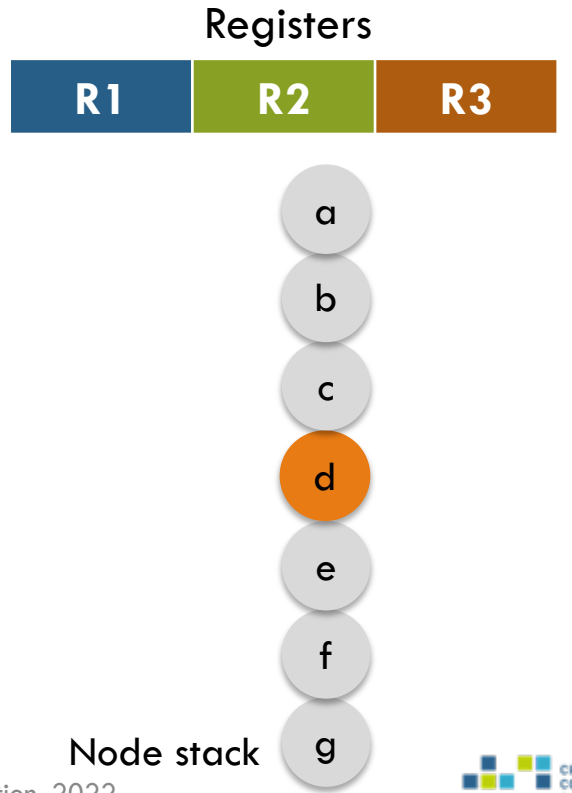


Node stack

Graph coloring: Example 2 – Simplify

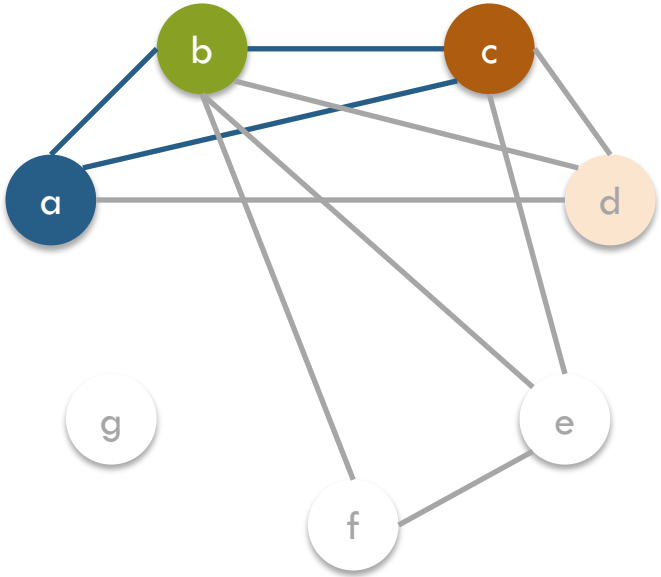
- Interference graph with 3 registers





Graph coloring: Example 2 – Select

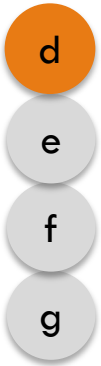
- Interference graph with 3 registers



Registers

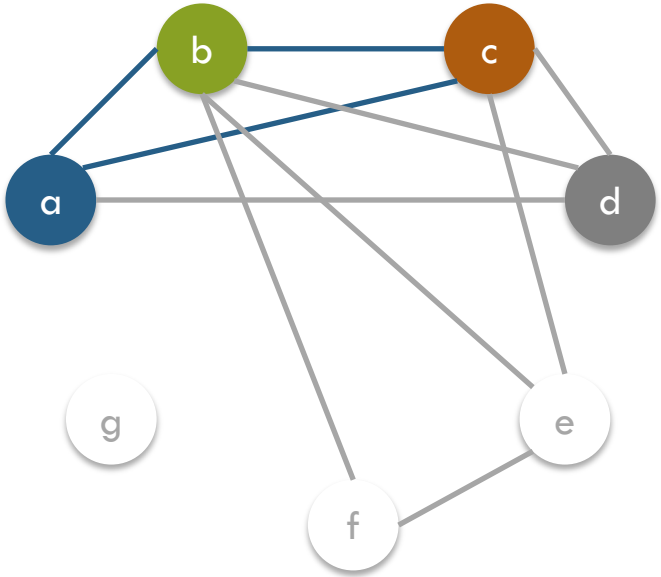
Spill really?

Node stack



Graph coloring: Example 2 – Select

- Interference graph with 3 registers

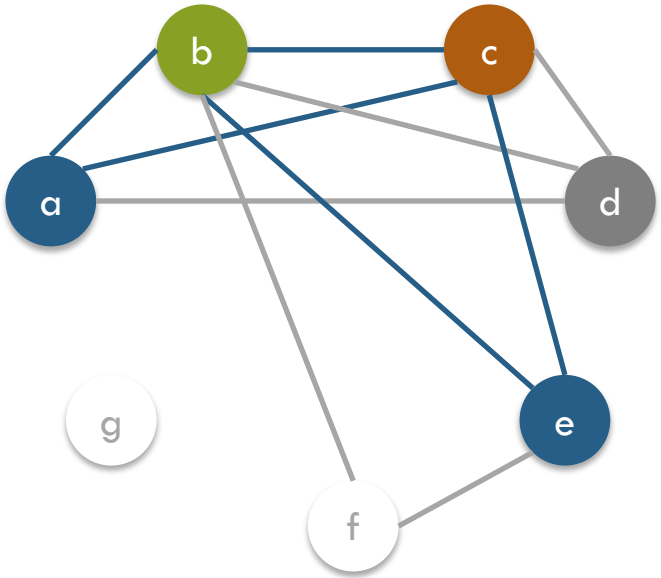


Registers

Node stack

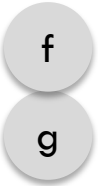
Graph coloring: Example 2 – Select

- Interference graph with 3 registers



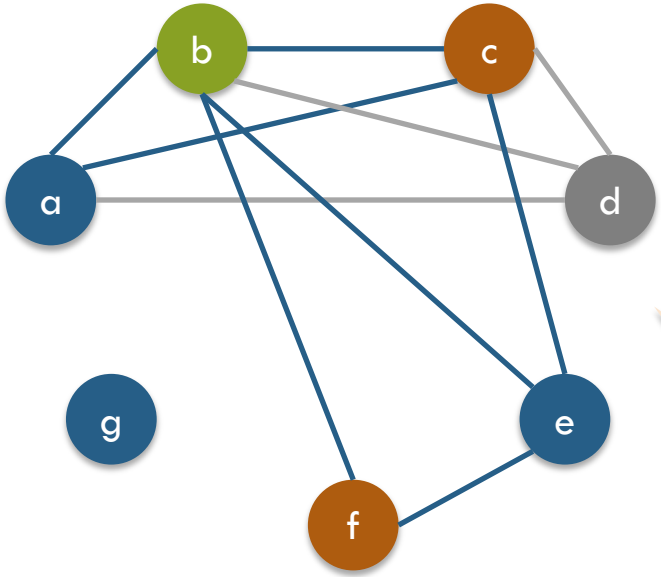
Registers

Node stack



Graph coloring: Example 2 – Select

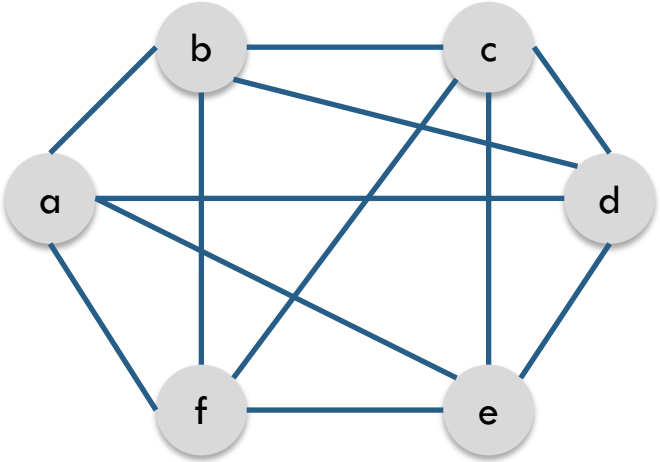
- Interference graph with 3 registers



Registers

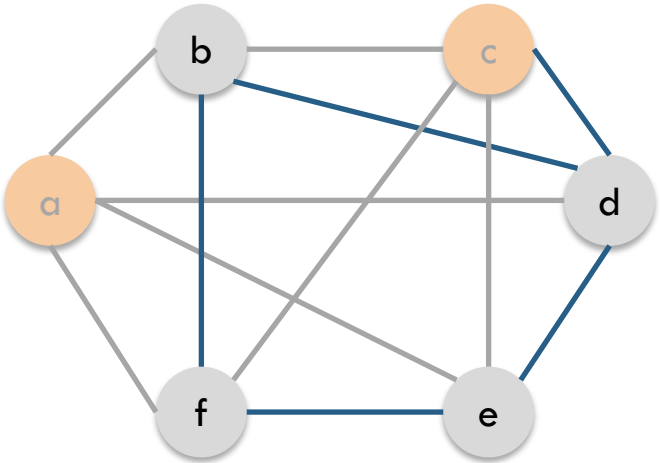
Start-over

Graph coloring: Last example – Optimality?



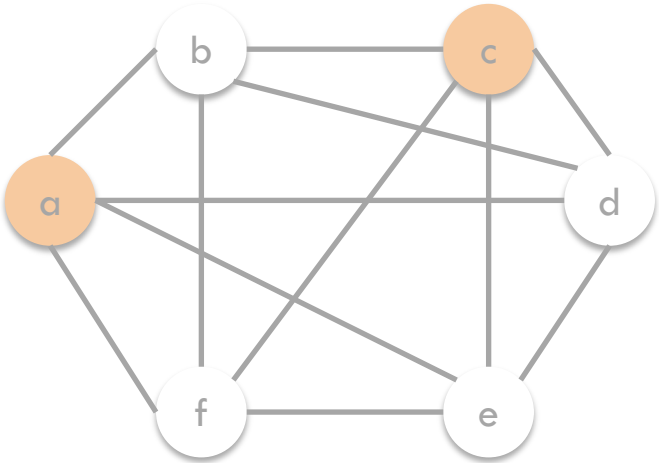
Registers

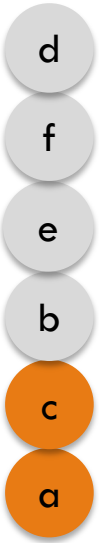
Graph coloring: Last example – Optimality?



Registers

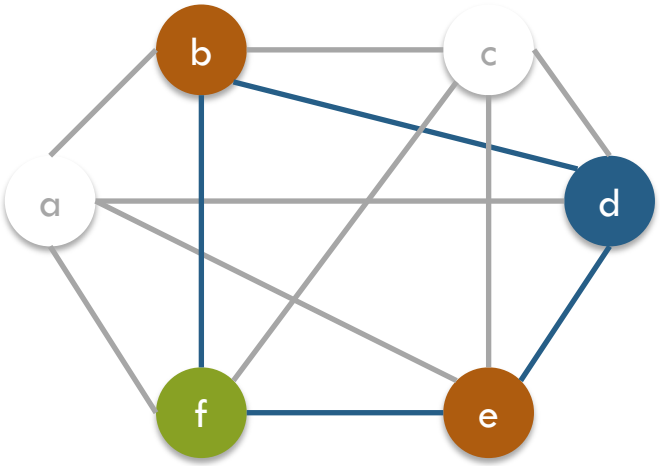
Graph coloring: Last example – Optimality?

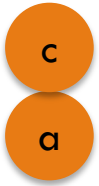




Registers

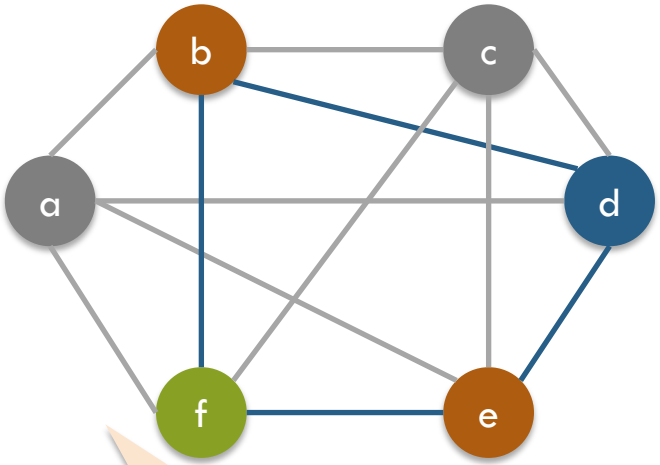
Graph coloring: Last example – Optimality?





Registers

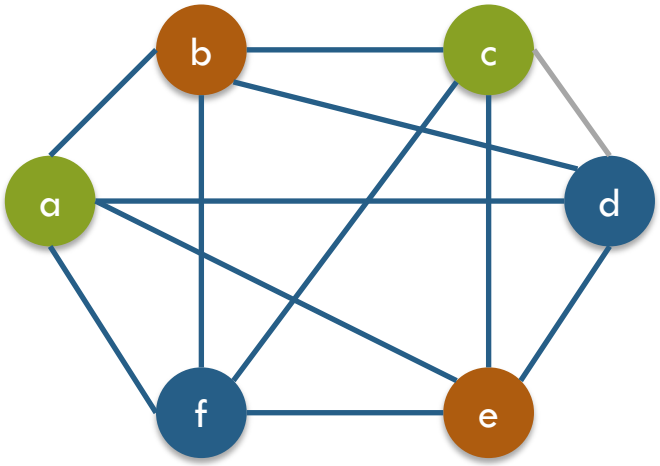
Graph coloring: Last example – Optimality?



Two spills!

Registers

Graph coloring: Last example – Optimality?



Registers

But coloring is possible!

Algorithm analysis

- Complexity? $O(n^2)$
  - Single iteration visit each node twice
  - If spill, start over
- Quality
  - Not optimal (underlying problem is NP complete)
  - Excellent results for many control-flow graphs
  - Heuristic: May produce pathological worst-case allocations (overspills in tight cases)

Further reading

- Heuristic for selecting nodes to spill in case no one with deg(X) < K
- Coalescing: Kind of copy propagation
- Aggressive coalescing: For any two pair of nodes (following safety rules)
- Dealing with pre-colored nodes (calling conventions)
- Top-down coloring
- Check GCC documentation (you'll now be able to understand most of it)

> … this is done in several subpasses: The integrated register allocator (IRA) … integrated because **coalescing,** register **live range splitting,** and hard register preferencing are done on-the-fly during **coloring**… The allocator helps to choose better pseudos for **spilling** based on their **live ranges** … IRA is a **regional register allocator** which is transformed into **Chaitin-Briggs** allocator if there is one region… the reload pass also optionally eliminates the **frame pointer** and inserts instructions to **save and restore … registers around calls.**

https://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html

11. Register allocation

- Introduction
- Local register allocation
- Global: Linear scan
- Global: Graph coloring
- Others

Other approaches to RA

- Bin-packing algorithm

  [Traub, Omri, Glenn Holloway, and Michael D. Smith. *Quality and speed in linear-scan register allocation.* Vol. 33. No. 5. ACM, 1998]

  - Aggressive version of linear scan
  - Uses live ranges instead of live intervals
  - Spill only some uses of variables ➔ Requires updates to dataflow analysis

- Hierarchical register allocation

  [Callahan, David, and Brian Koblenz. "Register allocation via hierarchical graph coloring." *ACM SIGPLAN Notices.* Vol. 26. No. 6. ACM, 1991]

  - Kind of local allocation: Going from inner structures of the control flow
  - Give priority to variables used in deep nests (dynamic uses)
  - Estimate costs associated with spilling a variable
    - Via profiling
    - Via coarse estimation: 10x with every loop nest

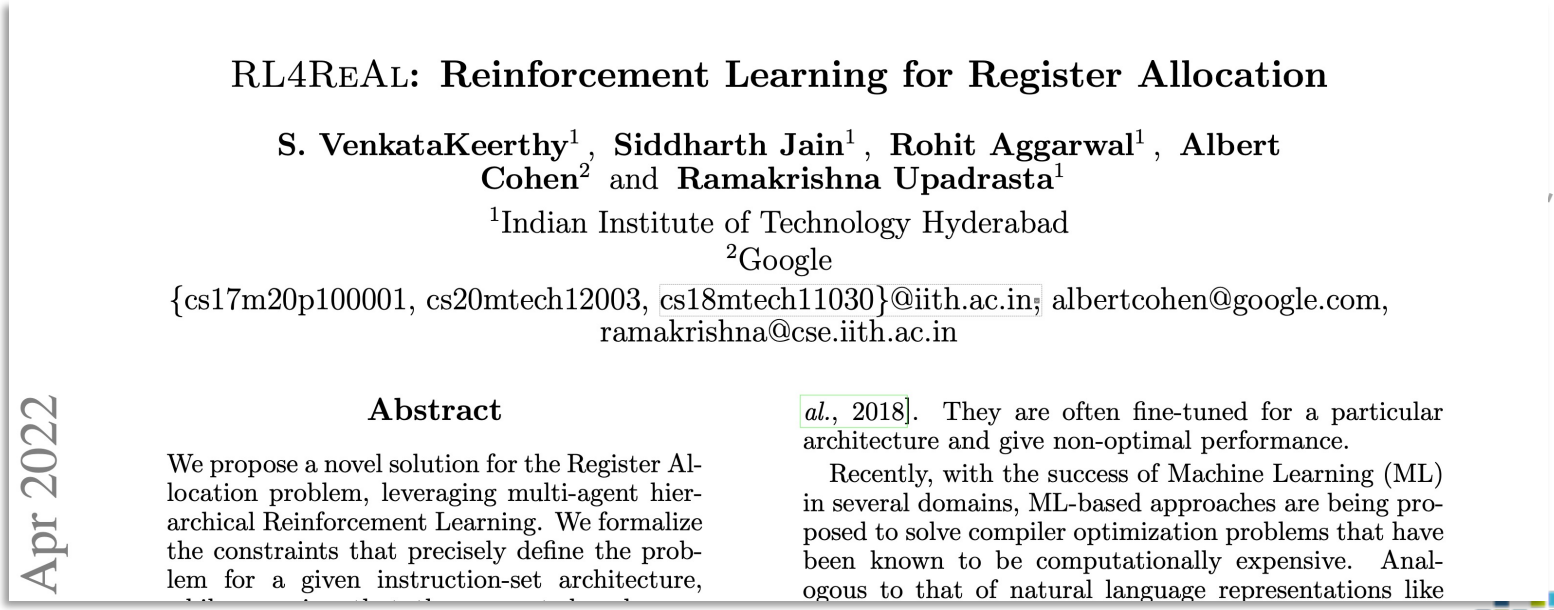
Other approaches to RA

- Bin-packing algorithm
  - Aggressive version of linear scan
  - Uses live ranges instead of live intervals

[Traub, Omri, Glenn Holloway, and Michael D. Smith. *Quality and speed in linear-scan register allocation*. Vol. 33. No. 5. ACM, 1998]

- hierarchical

, 1991]



RL4REAL: Reinforcement Learning for Register Allocation

S. VenkataKeerthy[1], Siddharth Jain[1], Rohit Aggarwal[1], Albert Cohen[2] and Ramakrishna Upadrasta[1]

[1]Indian Institute of Technology Hyderabad

[2]Google

{cs17m20p100001, cs20mtech12003, cs18mtech11030}@iith.ac.in; albertcohen@google.com, ramakrishna@cse.iith.ac.in

Abstract

We propose a novel solution for the Register Allocation problem, leveraging multi-agent hierarchical Reinforcement Learning. We formalize the constraints that precisely define the problem for a given instruction-set architecture,

al., 2018]. They are often fine-tuned for a particular architecture and give non-optimal performance.

Recently, with the success of Machine Learning (ML) in several domains, ML-based approaches are being proposed to solve compiler optimization problems that have been known to be computationally expensive. Analogous to that of natural language representations like

Where are we?

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

11. Scheduling

- Introduction
- Local scheduling
- Global scheduling

Backend: Recall

- Code selection: Uses pattern matching
  - Map IR code to assembly instructions (assume fixed code shape, i.e., IR is “optimal”)
  - ➔ For complexity reasons, assume infinite registers
- Register allocation
  - Assign registers to variables (change storage pattern, i.e., inserts load/stores)
- **Scheduling**
  - **Reorder operations to hide latencies, assume a fixed program (set of operations)**

Scheduling: Definition

Def. **Scheduling** is the reordering of instructions for a given target architecture (constraints) to increase performance

- After code-selection: Need to know which instructions to schedule
- After register allocation: Dependencies due to register assignments & potentially new code due to spilling
  - Phase ordering problem: Register allocation may introduce false dependencies!
  - Solution: Pre-sched (with virtual registers), register allocation, post-sched.
- Complexity: with resource constraints NP complete even for easy cases
  - Need heuristics

Static (compile-time) vs. Dynamic (run-time)

- Static (compile-time)
  - Avoids extra HW, e.g., pipeline-interlocks or out-of-order: VLIW vs. Superscalar
  - Can observe the entire program
  - Has more time to optimize the schedule

- Dynamic (run-time)
  - More precise: Some dependencies only known at run-time

```
A[i] = x;
y    = A[k];
```

Static: i = k?

```
ST [r1],r2
LD r3,[r4]
```

Dynamic: r1 = r4?

Data dependencies: Recall

- A **dependency** between two statements s_1 and s_2 indicates that s_2 cannot be executed before s_1
  - **Read after write** (**RAW**, true dependence): s_1 defines a variable, then s_2 reads it (see reachable defintions)

```
s1: a = b + c;
s2: d = a * x;
```

  - **Write after write** (**WAW**, output dependence): s_1 defines a variable, then s_2 defines it again

```
s1: a = b - c;
s2: a = b * x;
```

  - **Write after read** (**WAR**, anti-dependence): s_1 uses a variable, then s_2 defines it

```
s1: x = a - c;
s2: a = b * y;
```

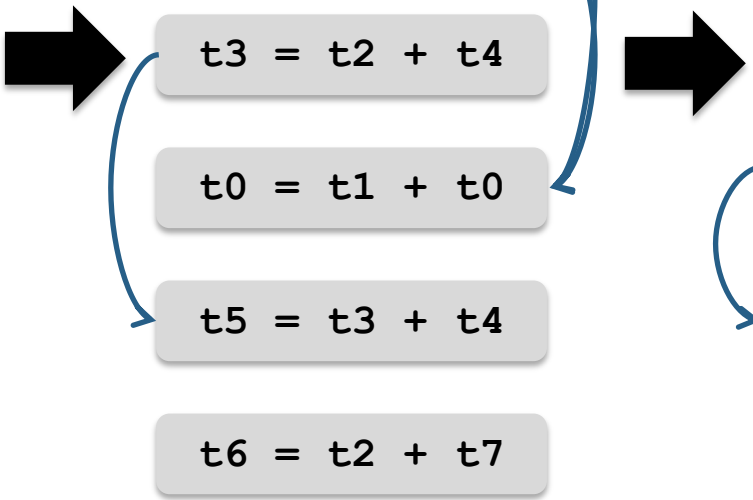
Data dependency graph

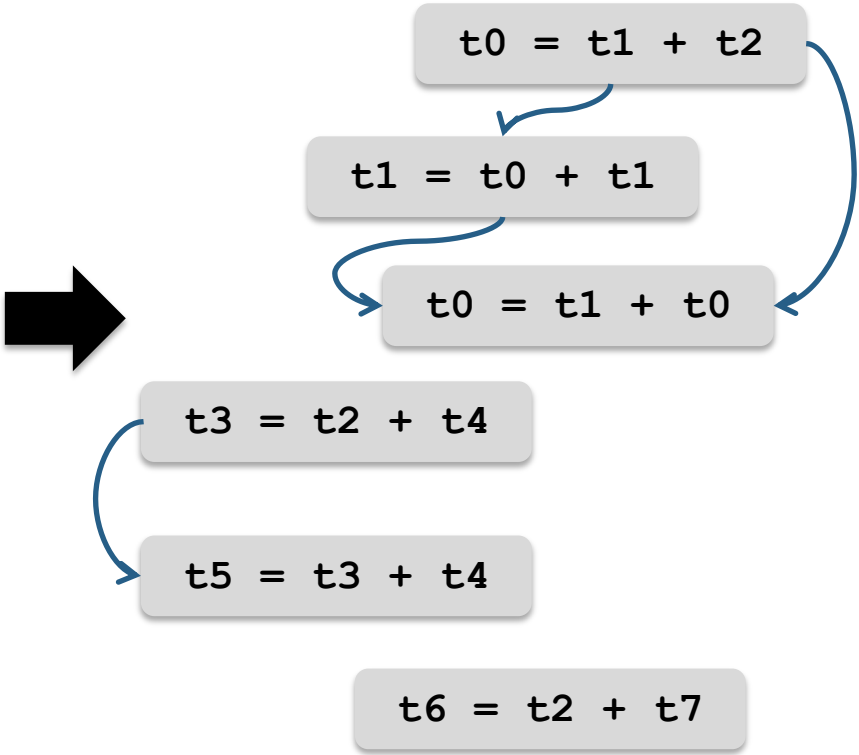
Def. Given a basic-block BB, its **dependency graph** is an edge-weighted directed acyclic graph $G = (V,E,W)$ where nodes represent the instructions of BB and there is an edge $e = (u,v) \in E$ if due to dependencies, **u has to execute before v**. The edge-weight $w_e \in W$ models the latency of the architecture.

- Possible schedule: **Any** topological sort of the graph
- WAR dependency: Depending on target architecture, usually means "**u cannot execute after v**"

Data dependency graph: Example

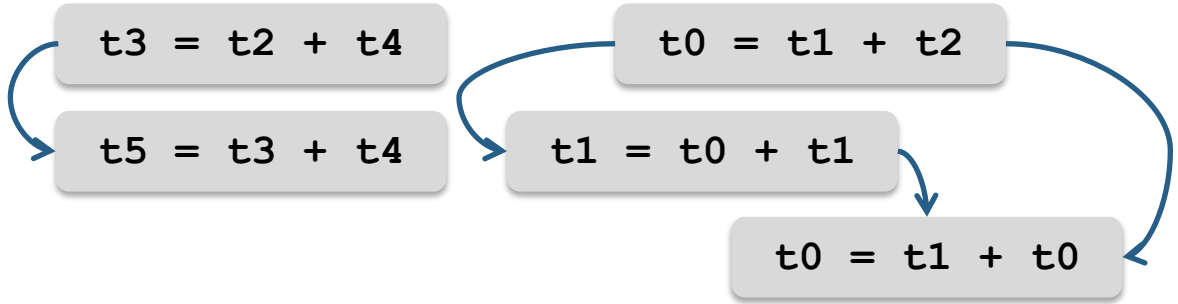
```
t0 = t1 + t2
t1 = t0 + t1
t3 = t2 + t4
t0 = t1 + t0
t5 = t3 + t4
t6 = t2 + t7
```





Adapted from: http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/

Possible schedules: Topological sort



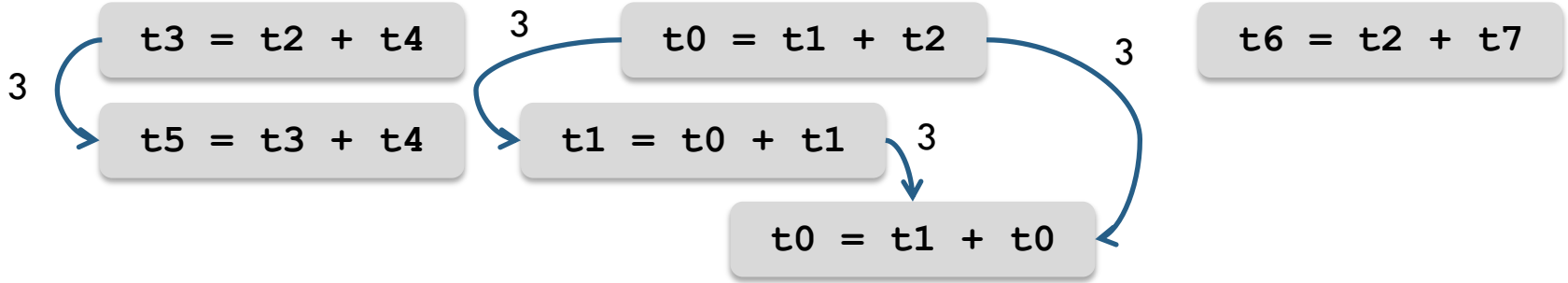

```
t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2
t1 = t0 + t1
t0 = t1 + t0
t6 = t2 + t7
```

```
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4
t0 = t1 + t0
```

Which one is better?

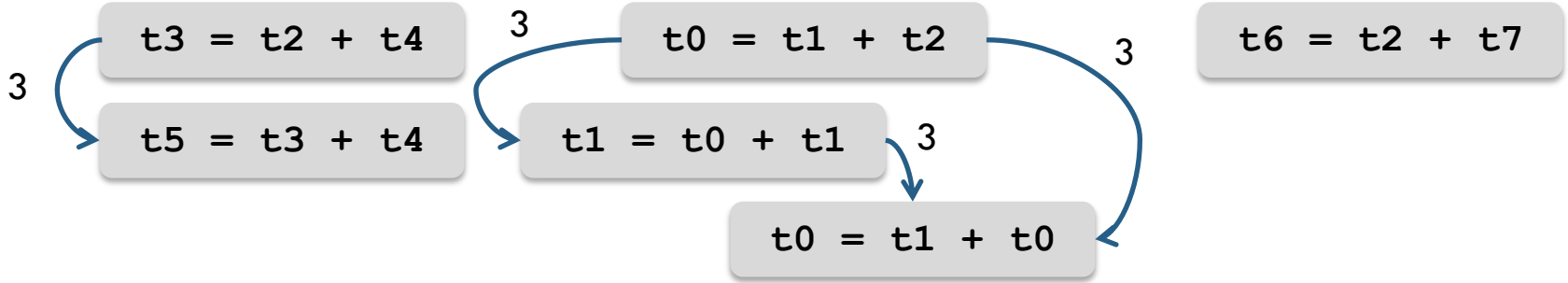
Heuristics, e.g., schedule first instructions with many outgoing edges

Adding edge weights: RISC processor



Option 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
t3 = t2 + t4	IF	ID	EX	MM	WB											
t5 = t3 + t4				IF	ID	EX	MM	WB								
t0 = t1 + t2					IF	ID	EX	MM	WB							
t1 = t0 + t1								IF	ID	EX	MM	WB				
t0 = t1 + t0											IF	ID	EX	MM	WB	
t6 = t2 + t7												IF	ID	EX	MM	WB

Adding edge weights: RISC processor



Option 2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
t0 = t1 + t2	IF	ID	EX	MM	WB											
t3 = t2 + t4		IF	ID	EX	MM	WB										
t6 = t2 + t7			IF	ID	EX	MM	WB									
t1 = t0 + t1				IF	ID	EX	MM	WB								
t5 = t3 + t4					IF	ID	EX	MM	WB							
t0 = t1 + t0							IF	ID	EX	MM	WB					

Saved cycles

Basic algorithms: ASAP

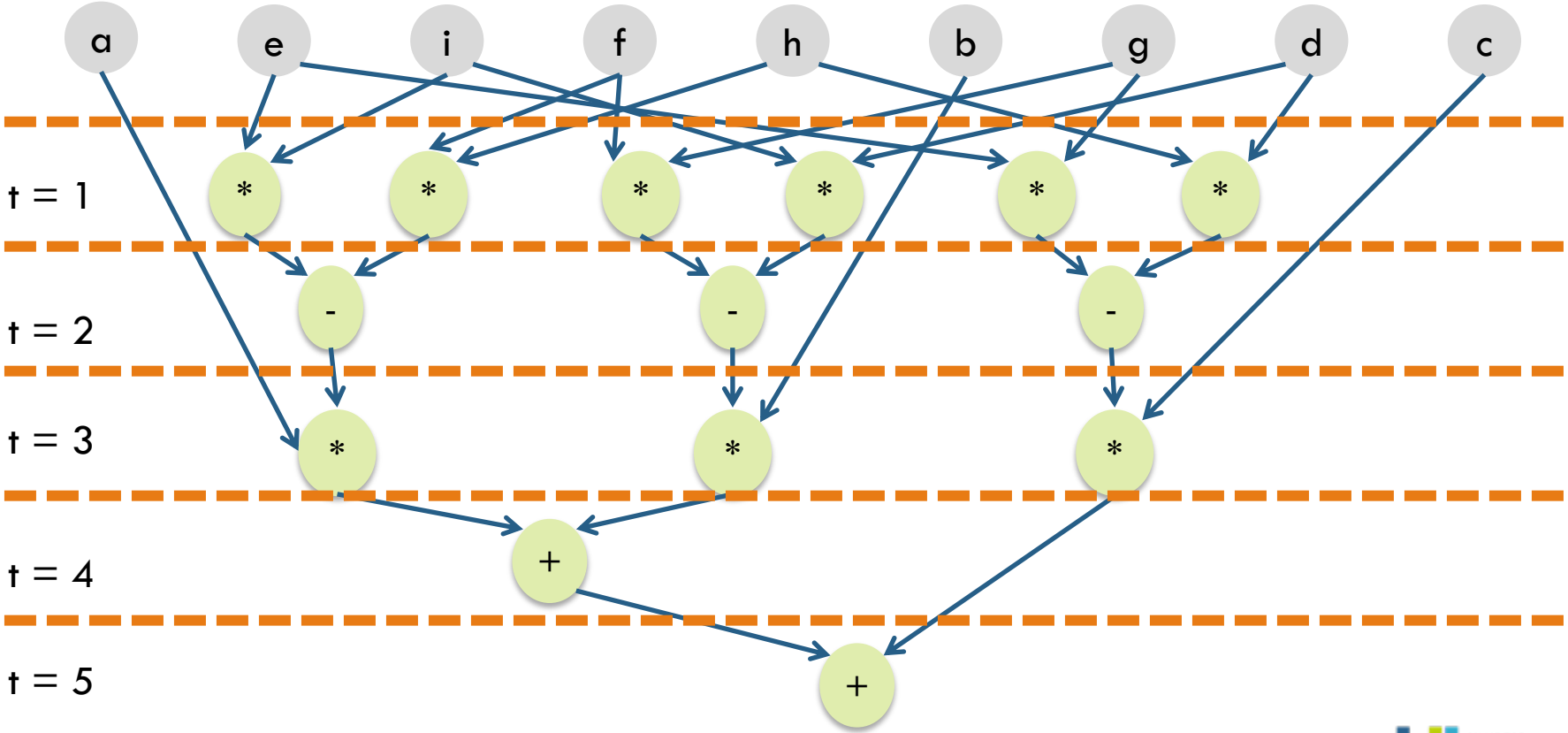
- ASAP: As soon as possible – schedule the earliest ignoring resource constraints
- Intuition: The earliest time depends on the predecessors and the latencies

Def. Given a dependency graph G = (V,E,W), **ASAP(v)** denotes the earliest scheduling time for every v ∈ V and is defined as:

- ASAP(v) = 1 if pred(v) = ∅
- $ASAP(v) = \max_{u \in pred(v)}(ASAP(u) + w_{(u,v)})$

- Relevance: The ASAP time represents an absolute lower bound

ASAP: Example (edge weights: 1)



Basic algorithms: ALAP

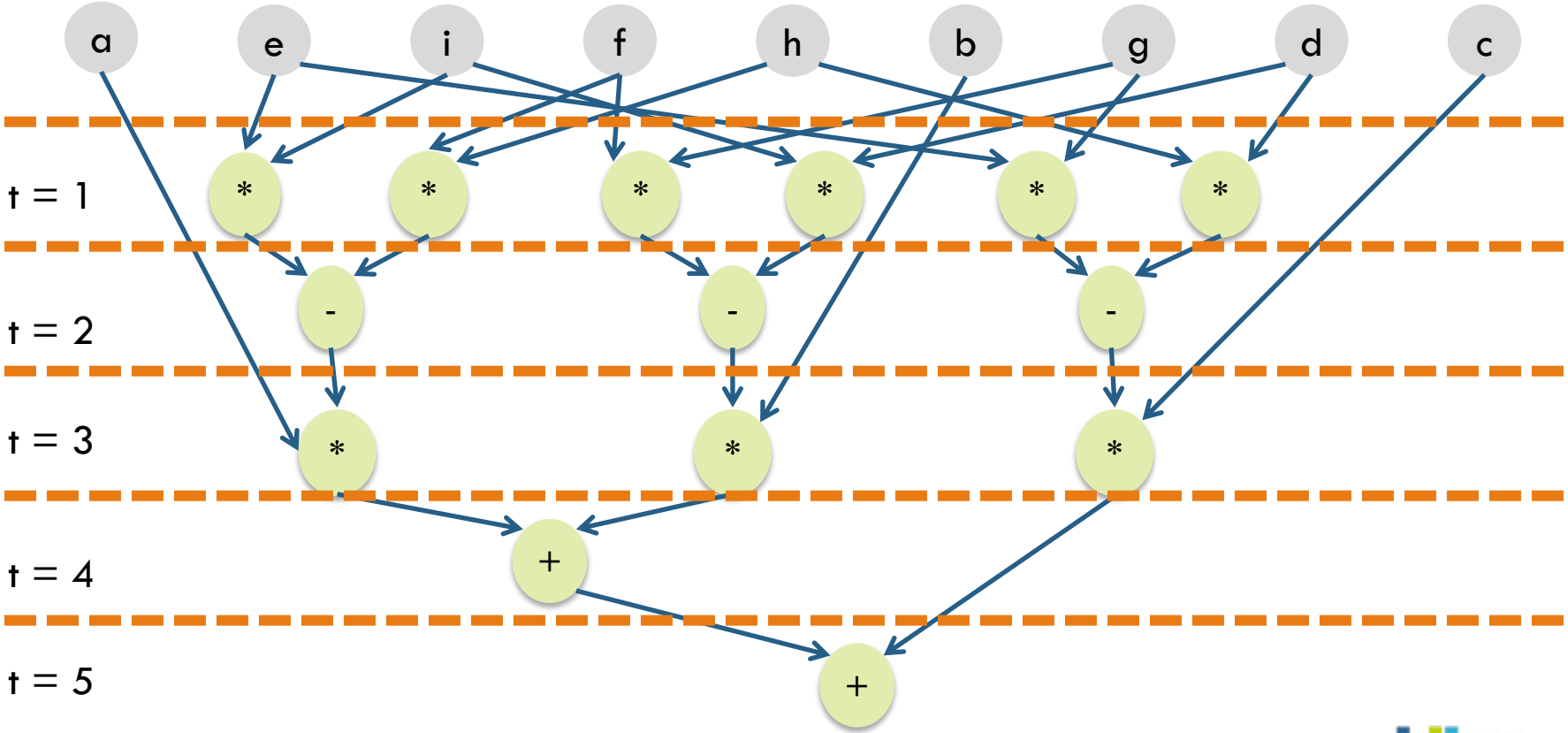
- ALAP: As late as possible – schedule the latest ignoring resource constraints while keeping same duration as ASAP

Def. Given a dependency graph G = (V,E,W), the **critical path length** (L_c) is the length of the paths in the graph with maximum edge weights

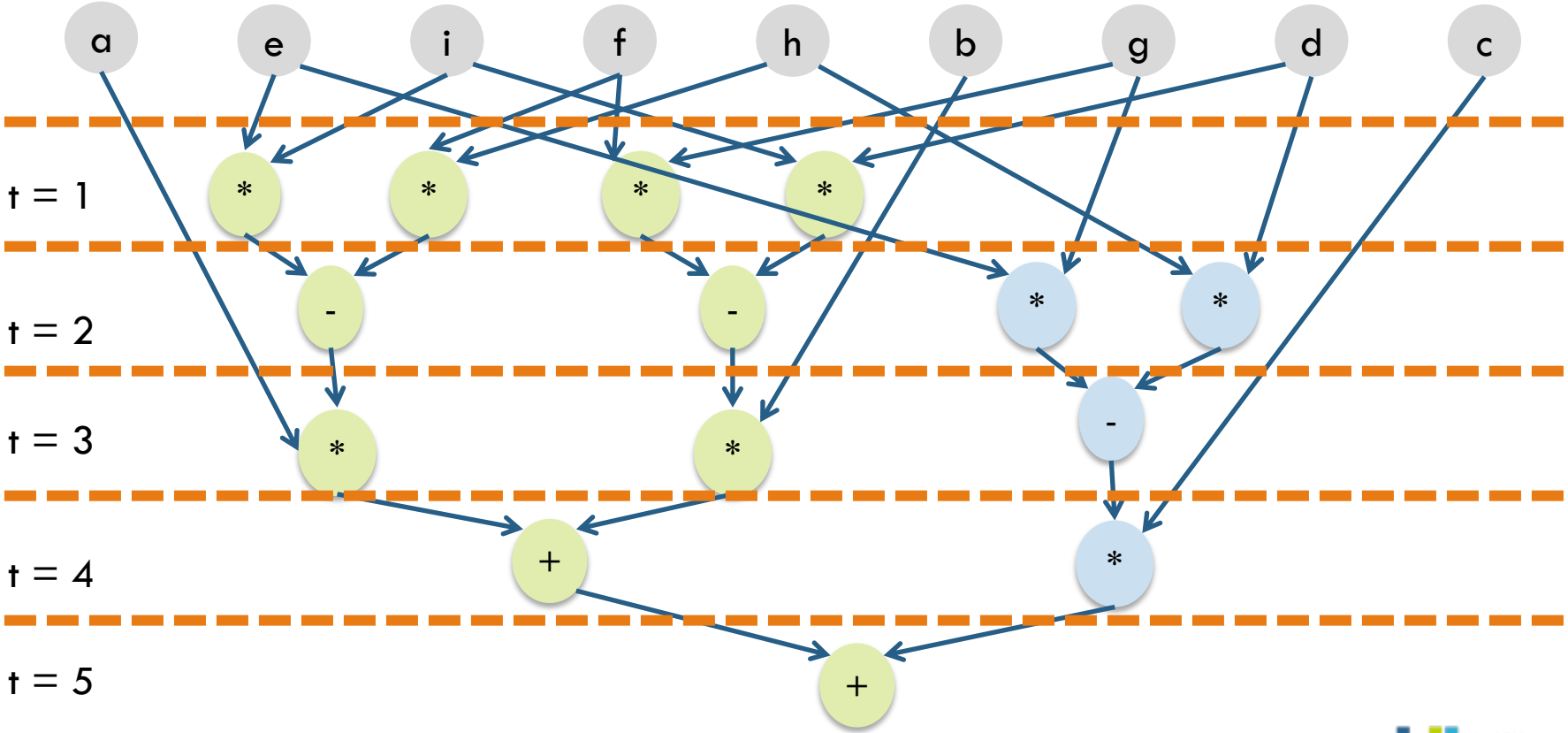
Def. Given a dependency graph G = (V,E,W), **ALAP(v)** for every v ∈ V is defined as:

- $ALAP(v) = L_c$ if $succ(v) = \emptyset$
- $ALAP(v) = \min_{u \in succ(v)}(ALAP(u) - w_{(v,u)})$

ASAP: Example (edge weights: 1)



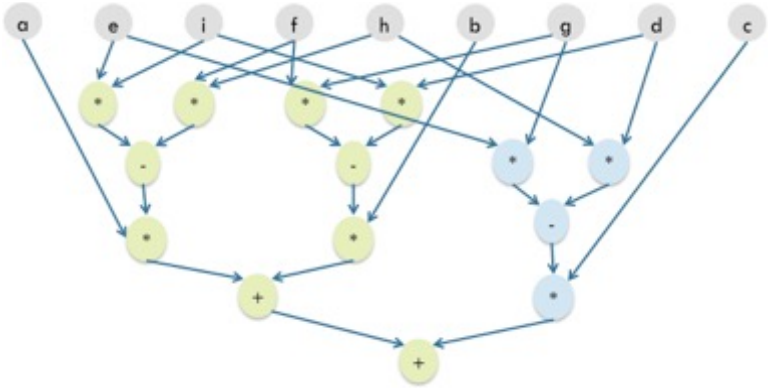
ALAP: Example (edge weights: 1)



ASAP & ALAP

- Theoretical optimum: Serves to restrict search
- Critical path: All nodes for which **ASAP(v) = ALAP(v)**
- Mobility: ALAP(v) – ASAP(v)
- If every node v is scheduled within [ASAP(v), ALAP(v)] ➔ Optimal schedule

- ASAP & ALAP: Ignore resource constraints
  - Requires 4 ALUs!



11. Scheduling

- Introduction
- Local scheduling
- Global scheduling

List scheduling

- Widespread heuristic algorithm for local scheduling, i.e, within basic-blocks
  - Simple and efficient

- Idea: Build dependency graph and repeatedly
  - **Get ordered ready set** of statements, i.e., statements whose predecessors are already scheduled
  - **Select** a node from the ready set (according to **priority**)
  - **Insert** the node in the **partial** schedule (in the best possible slot)

Algorithm

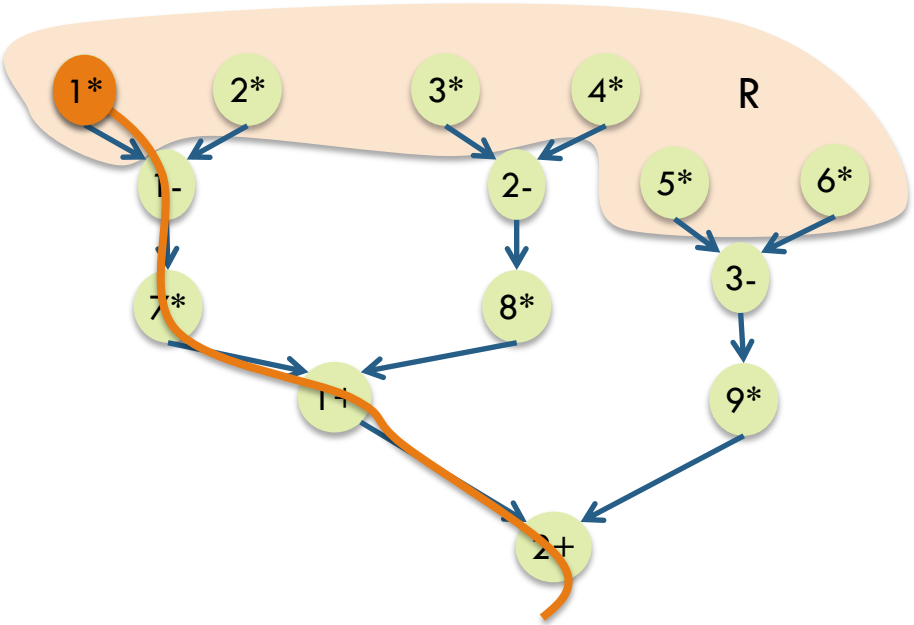
```
Proc  ListSched(Graph G)
S = empty schedule
O = ∅                          // Ordered nodes
while O ≠ V
    R = ReadySet(G)            // R = {v ∈ V; v ∉ O, pred(v)⊆O}
    v = SelectNode(R)          // Heuristic selection
    S = Insert(S,v)            // Insert v in the earliest possible time
    O = O ∪{v}
return S
```

List scheduling: Select heuristics

- In case $|R| > 1$: Good selection of a node to schedule is crucial
- Most famous ones
  - Select node with the most successors (increases options)
  - Select node on the **dynamic** critical path
  - Many others, although less used

- Complexity: Dominated by the construction of the dependency graph $O(|V|^2)$

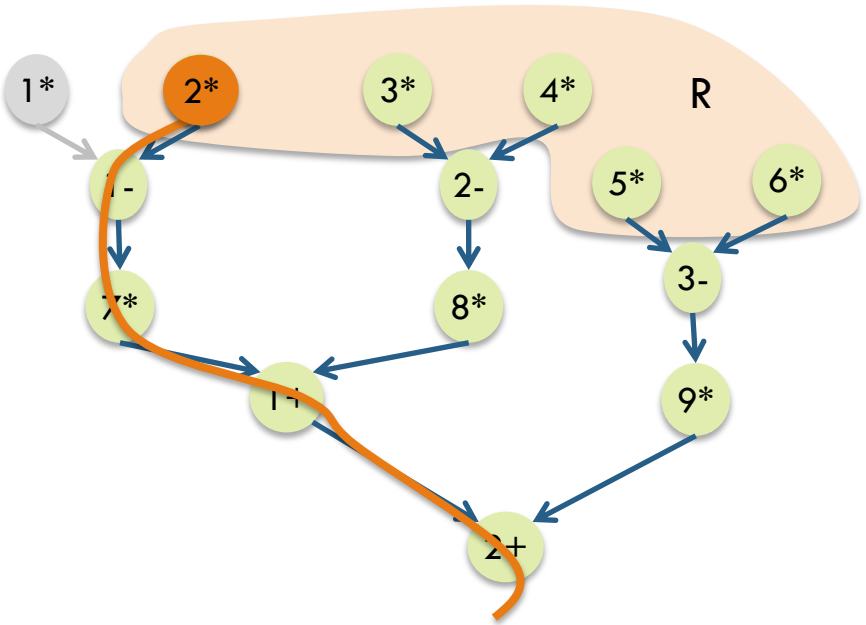
Example



- Assumptions
  - ISA with 2 multipliers and 2 ALU (+/-)
  - Instructions take 1 cycle
  - Select heuristic: Dynamic critical path

Time	*	*	+/-	+/-
t = 1	1*			
t = 2				
t = 3				
t = 4				
t = 5				
t = 6				

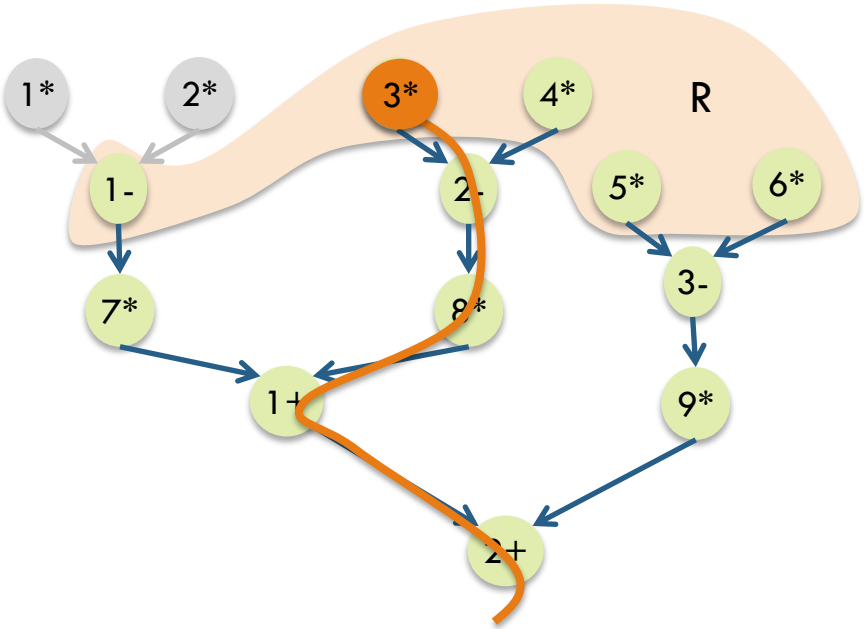
Example



- Assumptions
  - ISA with 2 multipliers and 2 ALU (+/-)
  - Instructions take 1 cycle
  - Select heuristic: Dynamic critical path

Time	*	*	+/-	+/-
t = 1	1*	2*		
t = 2				
t = 3				
t = 4				
t = 5				
t = 6				

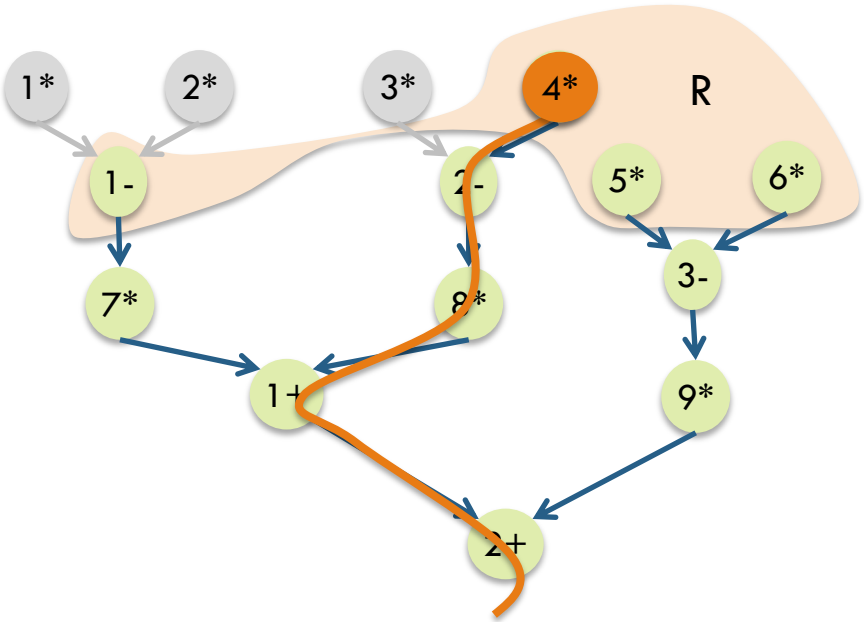
Example



- Assumptions
  - ISA with 2 multipliers and 2 ALU (+/-)
  - Instructions take 1 cycle
  - Select heuristic: Dynamic critical path

Time	*	*	+/-	+/-
t = 1	1*	2*		
t = 2	3*			
t = 3				
t = 4				
t = 5				
t = 6				

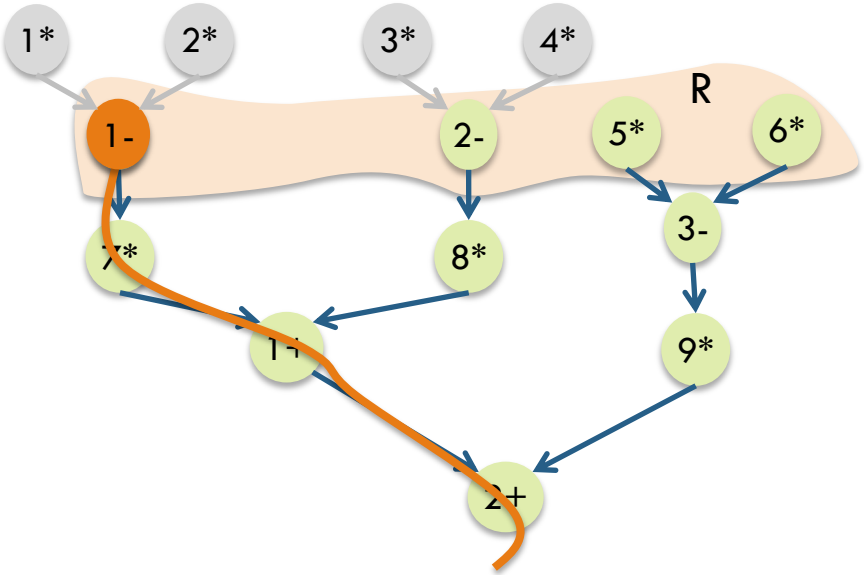
Example



- Assumptions
  - ISA with 2 multipliers and 2 ALU (+/-)
  - Instructions take 1 cycle
  - Select heuristic: Dynamic critical path

Time	*	*	+/-	+/-
t = 1	1*	2*		
t = 2	3*	4*		
t = 3				
t = 4				
t = 5				
t = 6				

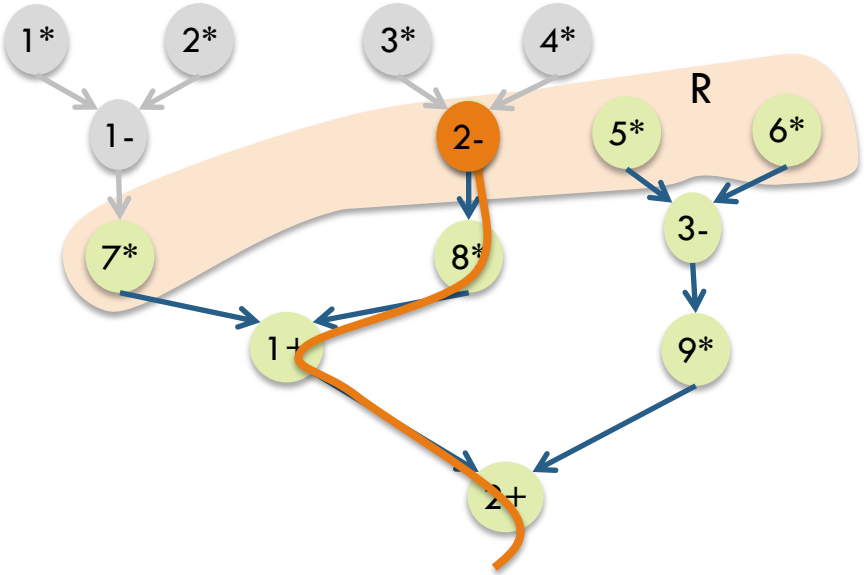
Example



- Assumptions
  - ISA with 2 multipliers and 2 ALU (+/-)
  - Instructions take 1 cycle
  - Select heuristic: Dynamic critical path

Time	*	*	+/-	+/-
t = 1	1*	2*		
t = 2	3*	4*	1-	
t = 3				
t = 4				
t = 5				
t = 6				

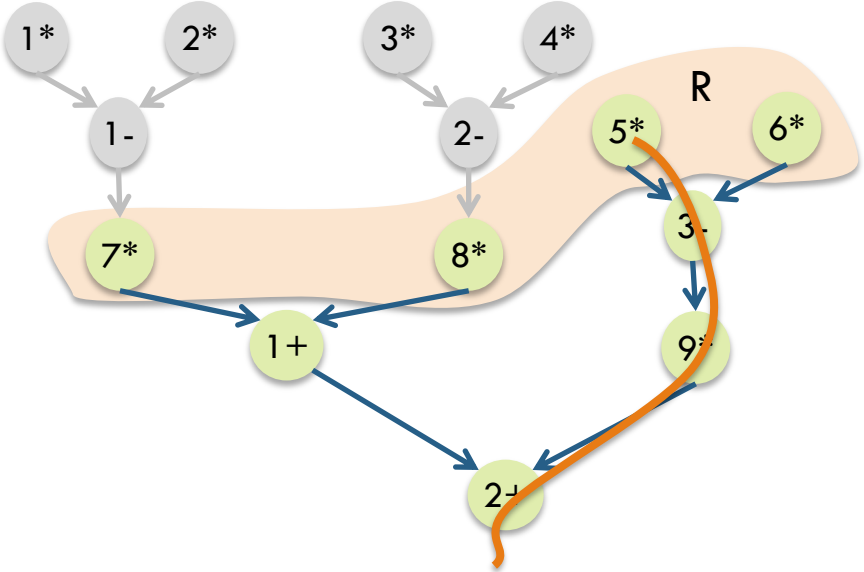
Example



- Assumptions
  - ISA with 2 multipliers and 2 ALU (+/-)
  - Instructions take 1 cycle
  - Select heuristic: Dynamic critical path

Time	*	*	+/-	+/-
t = 1	1*	2*		
t = 2	3*	4*	1-	
t = 3			2-	
t = 4				
t = 5				
t = 6				

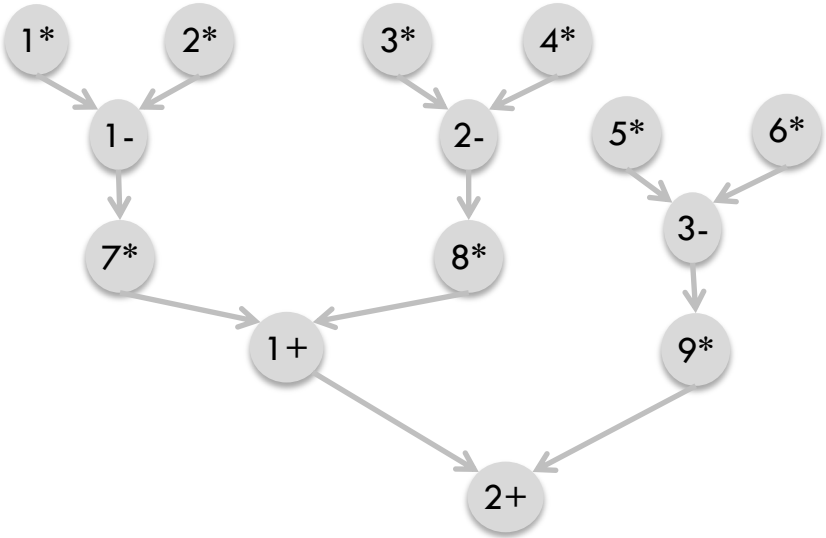
Example



- Assumptions
  - ISA with 2 multipliers and 2 ALU (+/-)
  - Instructions take 1 cycle
  - Select heuristic: Dynamic critical path

Time	*	*	+/-	+/-
t = 1	1*	2*		
t = 2	3*	4*	1-	
t = 3	5*	6*	2-	
t = 4				
t = 5				
t = 6				

Example

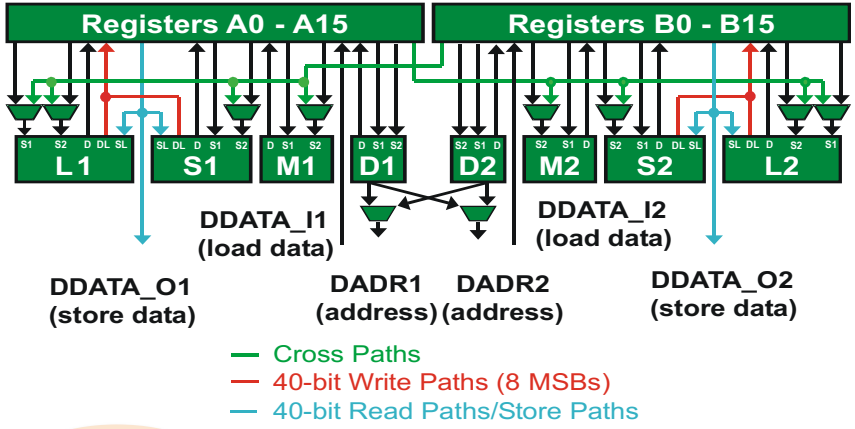


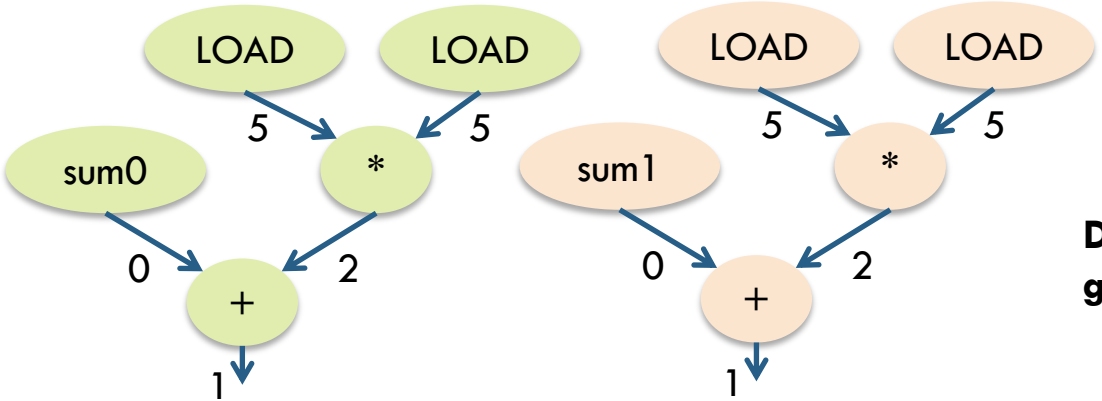
- Assumptions
  - ISA with 2 multipliers and 2 ALU (+/-)
  - Instructions take 1 cycle
  - Select heuristic: Dynamic critical path

Time	*	*	+/-	+/-
t = 1	1*	2*		
t = 2	3*	4*	1-	
t = 3	5*	6*	2-	
t = 4	7*	8*	3-	
t = 5	9*		1+	
t = 6			2+	

VLIW example: TI C6201

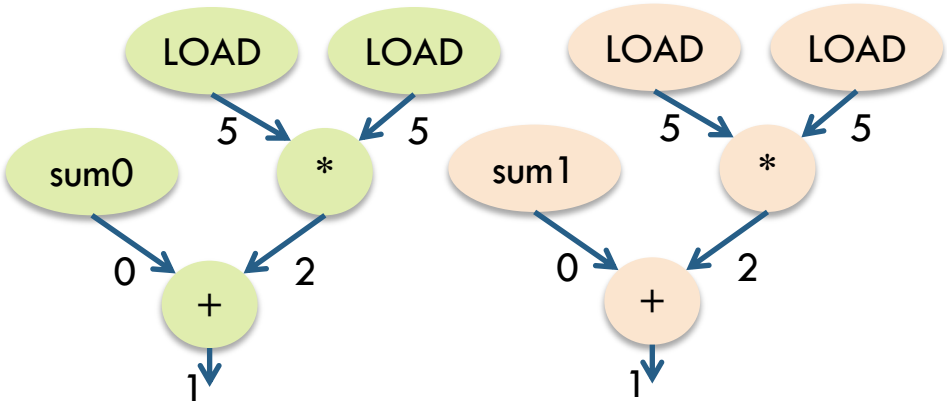
```
int dotp(short a[], short b[])
{ int sum0, sum1, i;
  sum0 = sum1 = 0;
  for (i = 0; i < 100; i += 2) {
    sum0 += a[i] * b[i];
    sum1 += a[i+1] * b[i+1];
  }
  return sum0 + sum1;
}
```

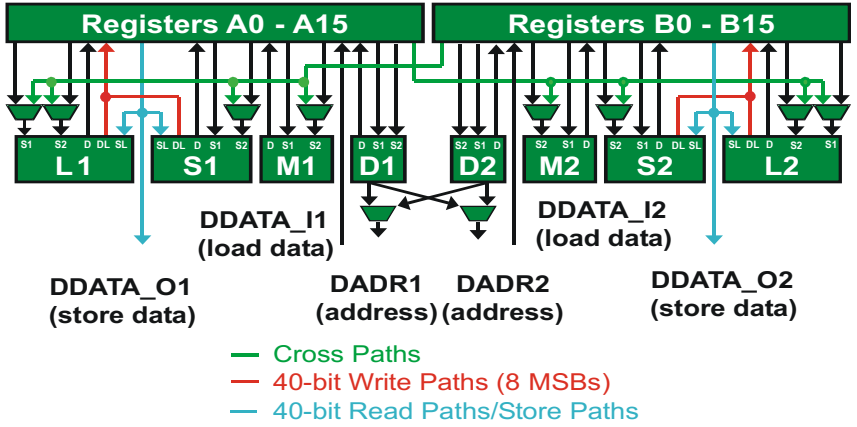




Dependency graph

VLIW example: TI C6201 (2)






```
LDH .D2   *++B4(4),B6 || LDH .D1 *++A4(4),A5
LDH .D2   *+B4(2),B5  || LDH .D1 *+A4(2),A6
NOP       3
MPY .M1X  B6,A5,A5
MPY .M1X  B5,A6,A6
NOP       1
ADD .L1   A6,A0,A0    || ADD .S1 A5,A3,A3
```

Using max. 2 parallel loads

3 more cycles delay before Multiply

11. Scheduling

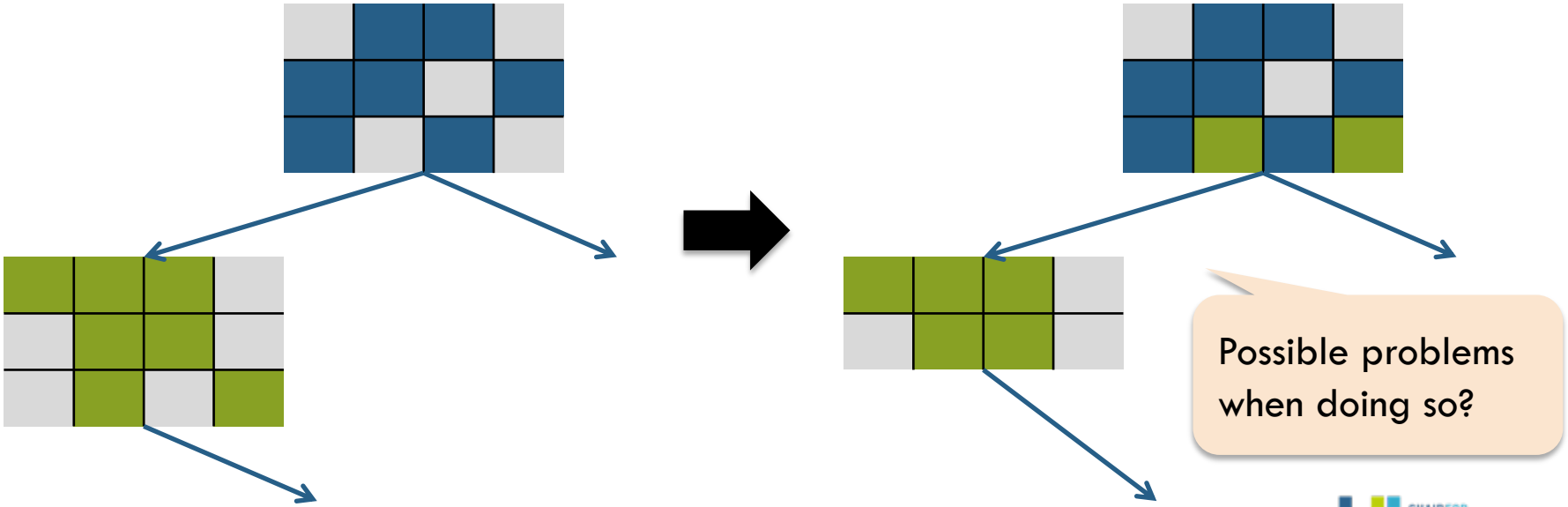
- Introduction
- Local scheduling
- Global scheduling

Global scheduling

- Motivation: Look beyond basic-blocks to exploit more ILP
  - Typical parallelism inside basic-block in C: 2-3 operation per cycle
  - Sample VLIW: TI C6201 with up to 8 operations per cycle
- **Trace-scheduling**
  - Trace: Sequence of basic-blocks
  - Move code to enlarge basic-blocks ➔ More ILP
  - Insert code to fix program (compensation code)
- Others: Superblocks, hyperblocks, tree regions, trace-2, …

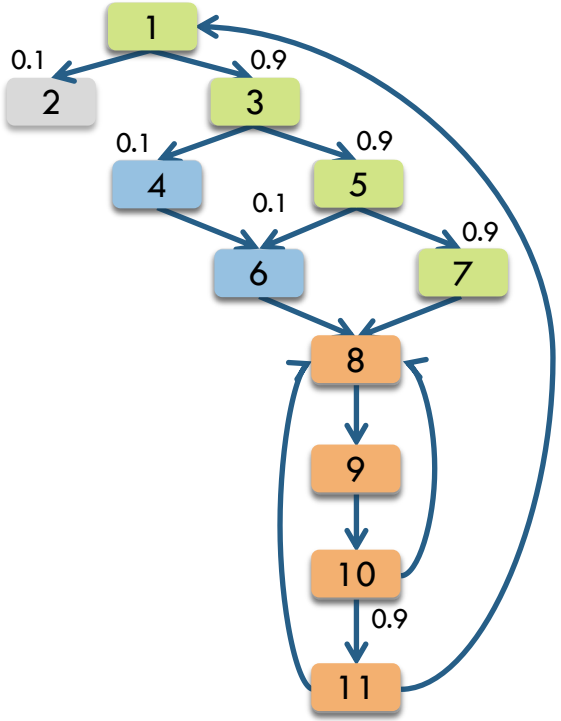
Trace scheduling

- Intuition
  - Use free slots in a basic-block to execute instruction of the following one
  - ➔ Reduce critical path



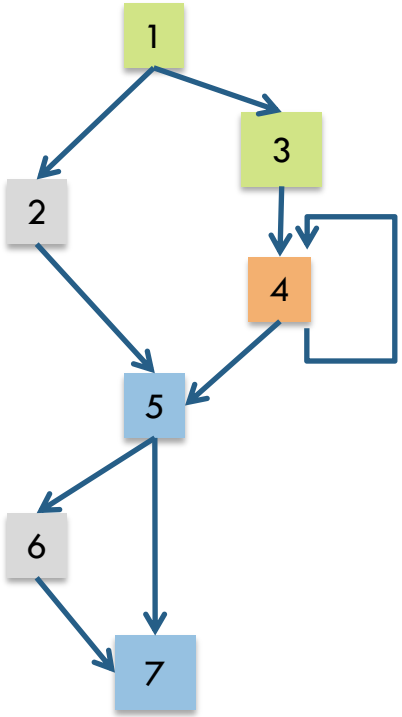
Trace scheduling (2)

- Intuition
  - Remove loops from control-flow graph (recall natural loops)
  - Partition control flow graph into disjoint traces
  - Partition: Base on execution statistics/estimates
    - Try to improve the execution time of the trace executed the most
  - Moving code: Changes the semantic
  - ➔ Need to insert fix code (tradeoff: performance-code size)



Compensation code: Example

```
B1: x = x+1
    y = x - y
    if x<5 goto B3
B2: z = x*z
    x = x+1
    goto B5
B3: y = 2*y
    x = x-2
    ...
B5: a = y+1
    ...
```



5 Traces

```
B1: x = x+1
    if x<5 goto B3
B2: y = x - y
    z = x*z
    x = x+1
    goto B5
B3: y = x - y
    y = 2*y
    x = x-2
    ...
B5: a = y+1
    ...
```

Fix program (if y is live-in in B2!)

Moved from B1 to B3

Where are we?

1. Introduction
2. Lexical analysis
3. Syntax analysis
4. Semantic analysis
5. Intermediate representation
6. Control & data-flow analysis
7. IR optimization
8. Target architectures
9. Code selection
10. Register allocation
11. Scheduling
12. Advanced topics

12. Advanced topics

- Bonus code optimizations
- Research topics

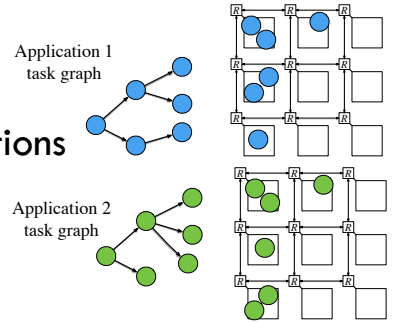
General

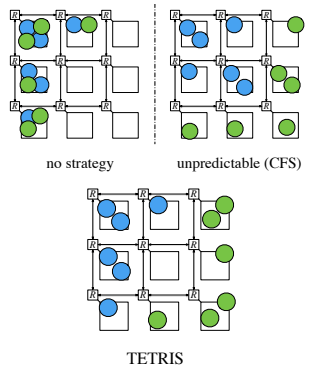
- Topic-wise
  - Formal programming models (dataflow, reactors), e.g., for automotive software
  - Compiler analysis and optimizations for emerging systems (AMD collaboration)
  - Domain-specific languages and optimizations (DSLs)
    - Computational biology
    - Physics simulations
    - Systems programming and big-data
- Students involved in research
  - Co-author research papers (e.g., CC'18, CC'20, CC'23?)
  - In exceptional cases visit conferences overseas

TETRiS: Run-time system for energy-efficient execution

- Context:
  - **TETRiS**: **T**ransitive **E**fficient **T**emplate **R**un-time **S**ystem
  - Exploits HW symmetries for mapping of multiple applications
  - Hybrid compile-time/run-time approach
  - Aims at energy-efficient and predictable execution
- Possible projects:
  - Subgraph isomorphism on heterogeneous platforms
  - Efficient generation of pareto-optimal configurations (e.g. by using evolutionary algorithms)
- Suitable for Großer Beleg, BA, DA or MA

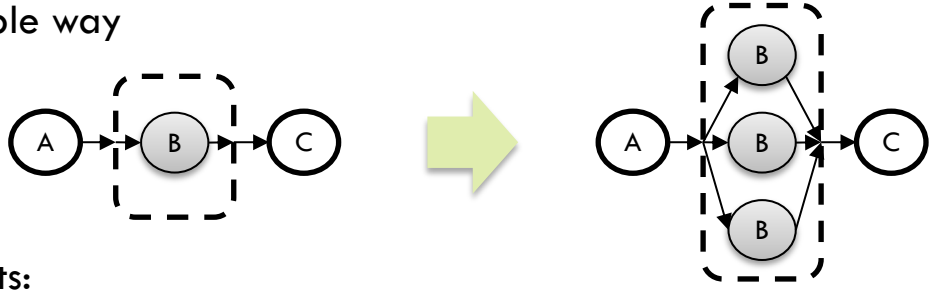
Contact: Robert Khasanov
robert.khasanov@tu-dresden.de





Implicit parallelism in Kahn Process Networks (KPN)

- Context:
  - Kahn Process Networks (KPN): Parallel dataflow programming
  - Extension of implicit parallelism in KPN allows reconfiguration of application in malleable way



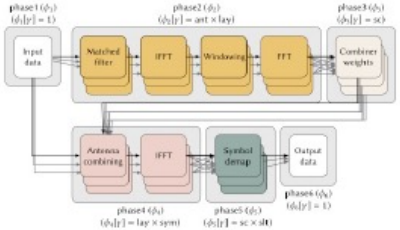
Contact: Robert Khasanov
robert.khasanov@tu-dresden.de

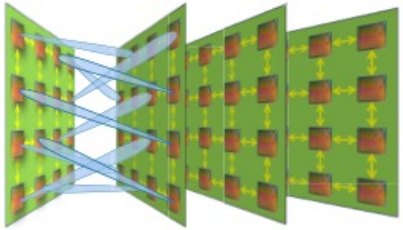
- Possible projects:
  - Code generation of KPN with implicit parallelism
  - Implicit parallelism on a distributed memory model
  - Using SIMD instructions for KPN with implicit parallelism
  - Adaptivity at run-time (integration with TETRiS)
- Suitable for Großer Beleg, BA, DA or MA

Adaptive methodologies for 5G and beyond

Context:

- High workload heterogeneity due to the variety of new use cases in 5G
- High flexibility required in baseband systems to support the fast evolution of the telecommunication standards





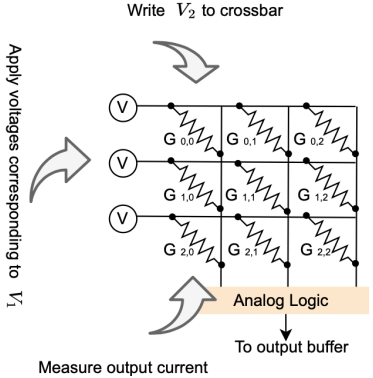
Contact: Julian Robledo-Mejia
julian.robledo@tu-dresden.de

Goal:

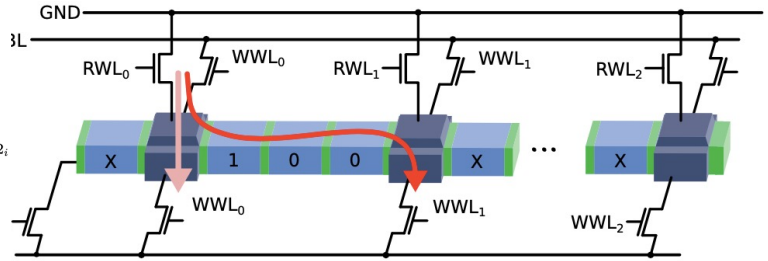
- Adaptive methodologies:
  - Profiling mobile workload parameterization
  - Adaptable models of computation
  - Exploring CRAN network architectures and heterogenous platforms
  - Scheduling algorithms for 5G applications

Tools for Compute-in-memory (CIM) systems

- CIM exploits the physical properties of memory devices to implement logic and arithmetic operations *in-memory*
- Simulation tools are essential to quickly evaluate and understand fundamental trade-offs in these systems



$$V_1 \cdot V_2 = \sum_{i=1}^{i=n} V_{1_i} V_{2_i}$$



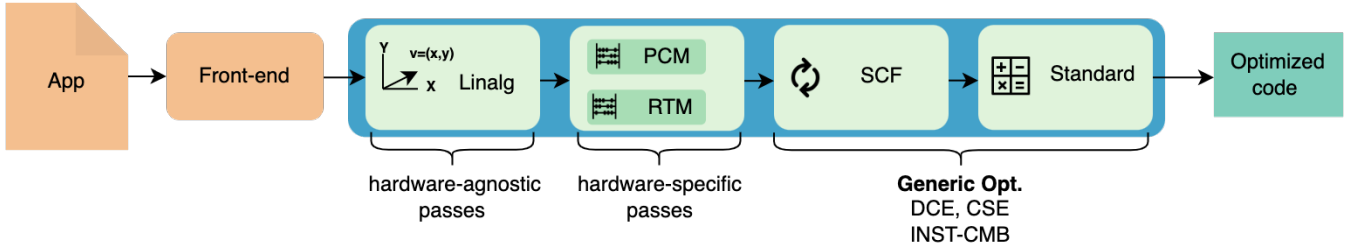
- **Goal:** Developing a simulation infrastructure to model CIM systems using various memory technologies

Contact: Asif Ali Khan
asif_ali.khan@tu-dresden.de

Skills:
C/C++ and python.
Physics and comp-arch is a plus

Front-ends for MLIR CIM Compiler

- Context:
  - Recent advances in technology have enabled in- and near-memory computing
  - End-to-end compilation flows exist but interfaces to high-level languages are missing (limited)



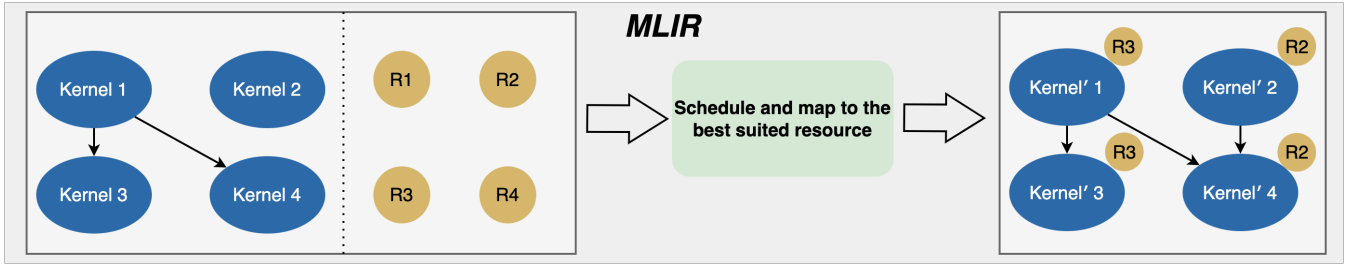
- **Goal**: Developing front-ends to enable lowering high-level languages/descriptions to the CIM compilers

Contact: Hamid/Asif
hamid.farzaneh@tu-dresden.de
asif_ali.khan@tu-dresden.de

Required:
C/C++;
Preferable:
LLVM, MLIR knowledge

Heterogeneous Systems: Mapping and Optimizations

- Context:
  - Future systems are predicted to be highly heterogenous
  - Efficient mapping and optimizations of the application on a heterogenous system is central to system's performance



- Goal: An MLIR based automatic infrastructure to map kernels to the fitting hardware target and optimize for it

Contact: Hamid/Asif
hamid.farzaneh@tu-dresden.de
asif_ali.khan@tu-dresden.de

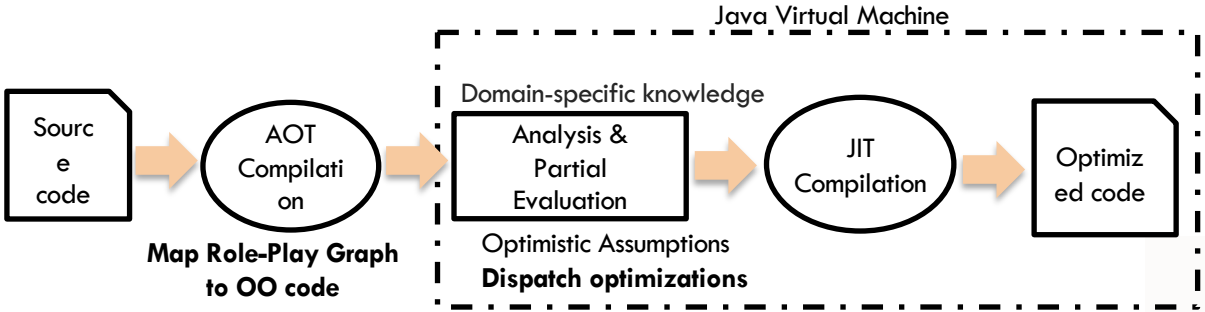
Required:
C/C++;
Preferable:
LLVM, MLIR, Polyhedral Compilation

Partial Evaluation of Role Dispatch in a Virtual Machine

Lars Schütze
lars.schuetze@tu-dresden.de

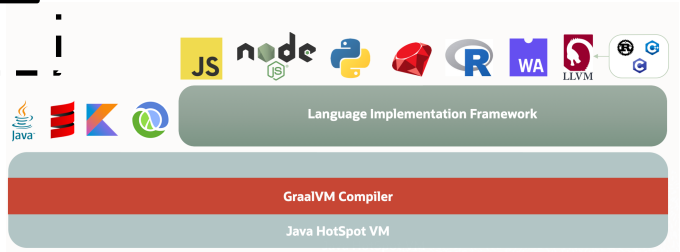
Context

- Adaptive software requires concepts beyond object-orientation
- Goal: Virtual machine aware of role mechanisms and **optimize at runtime**



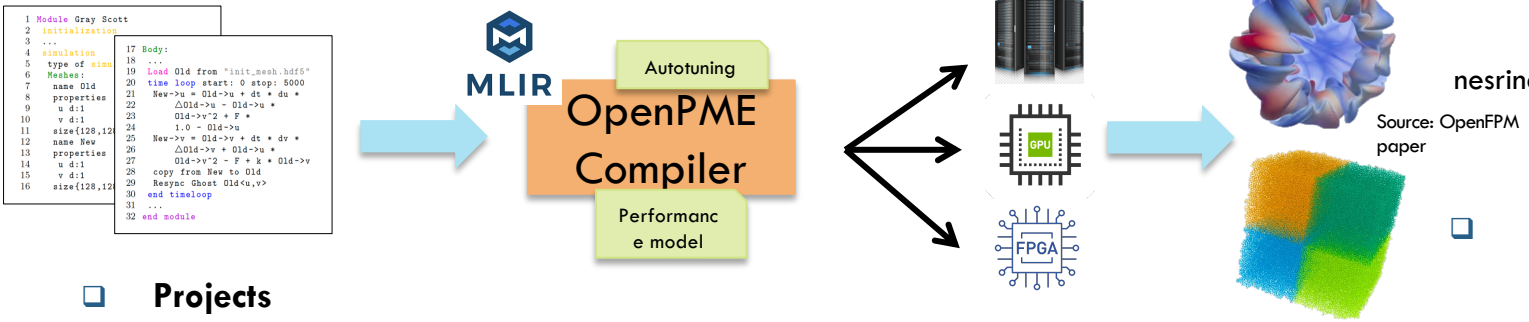
Task

- Extend Espresso, a Java Virtual Machine (JVM) written in Java (requires strong **Java experience**)
- Design and implementation of **optimizations** for role features
- Suitable for SHK/WHK, Practical, Theses (BA, DA, MA)



DSL and Compiler Optimizations for Particle-mesh Simulations

- **Context**
  - OpenPME: A DSL for particle-mesh simulation that generates C++ code against OpenFPM library
  - Model-based autotuning approach for numerical discretization methods
  - Goal: Extend OpenPME DSL and enhance its compiler



Source: OpenFPM paper

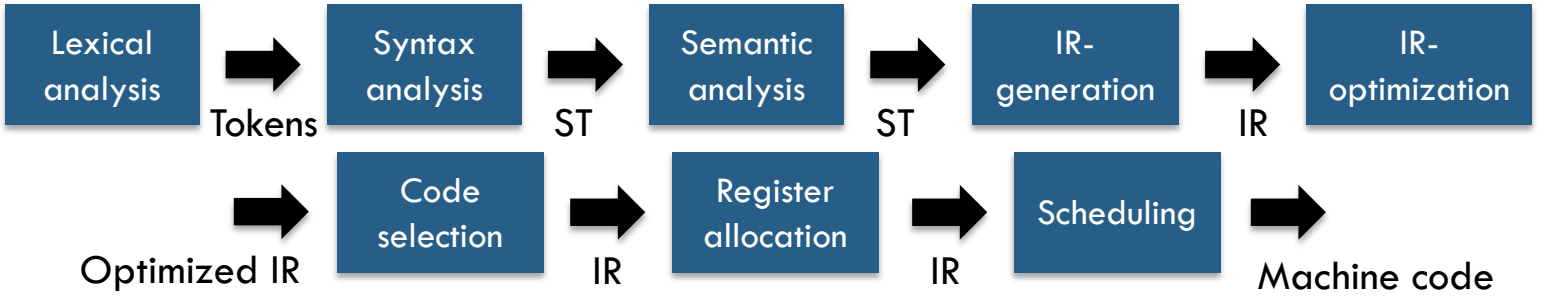
Contact: Nesrine Khouzami
nesrine.khouzami@tu-dresden.de

- **Projects**
  - Develop extensions and enhancements for OpenPME code generator
  - Integration of the autotuning approach in OpenPME compiler
  - Analyse particle-mesh interactions and propose a performance model to optimize memory consumption
  - Implement a particle-mesh MLIR dialect to target different backends

- **Required**
  - C++
  - Compiler construction
- **Useful**
  - MPS Jetbrains
  - MLIR/LLVM

Closing remarks

It's been a long journey



Thanks for you attention!!

&

Good luck in the exam!!!

(… now time to talk about the exam)