Data Structures & Algorithms @ Furpocalypse 2026

Rocq and ROL Nightmare

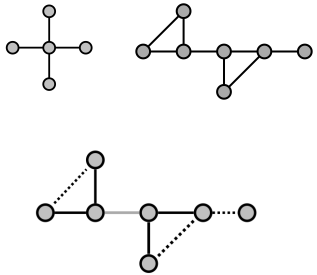
Sunny, Unsigned Long

November 1, 2026



Planar Graphs

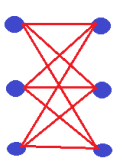
- Deleting vertices or contracting/deleting edges can't turn a planar graph into a non-planar graph.

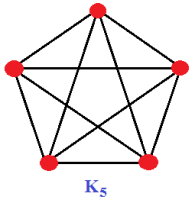


diagrams from Wikimedia

Planar Graphs

- Deleting vertices or contracting/deleting edges can't turn a planar graph into a non-planar graph.
- **Wagner's Theorem**:
  A graph is planar if and only if you cannot form a K_5 or $K_{3,3}$ using deletions and contractions.
- K_5 are $K_{3,3}$ the **forbidden graph minors** of the family of planar graphs.





diagrams from Wikimedia

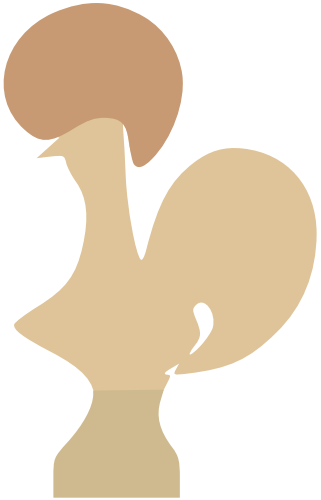
Forbidden Minors

- H is a **graph minor** of G if H can be obtained from G by deleting edges, isolated vertices **or** contracting edges.
- A family $\mathcal{F}$ of graphs is **closed under taking minors** if every minor of a graph in $\mathcal{F}$ also belongs to $\mathcal{F}$.
- **Robertson–Seymour theorem**:
  Every family of graphs that is closed under taking minors can be defined by a finite set of forbidden minors.

Animal computing mascots: Rocq



Animal computing mascots: Coq?

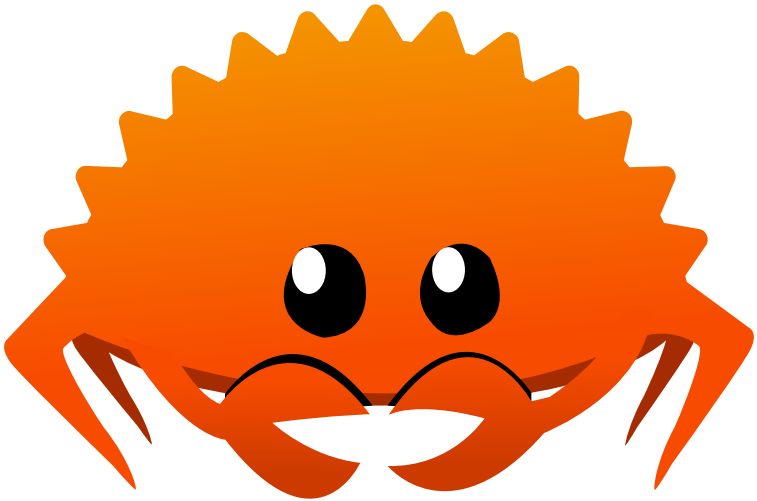


Animal computing mascots: Coq?

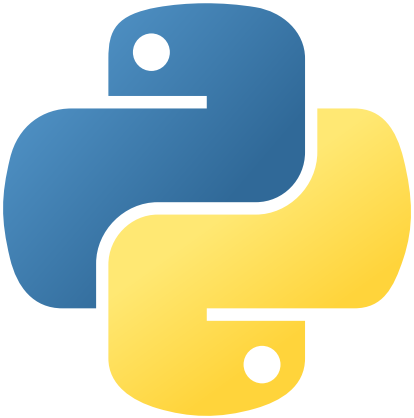
We ran many regressions to learn which demographic characteristics were associated with a different opinion. The only characteristic that was statistically significant (at the 0.05 level) was whether the respondent is from an English-speaking or a non-English-speaking location.

https://discourse.rocq-prover.org/t/coq-community-survey-2022-results-part-iv-and-itp-paper-announcement/2001#p-5444-renaming-coq-8

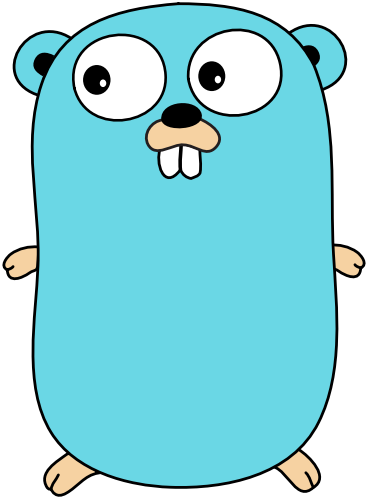
Animal computing mascots: Rust



Animal computing mascots: Python



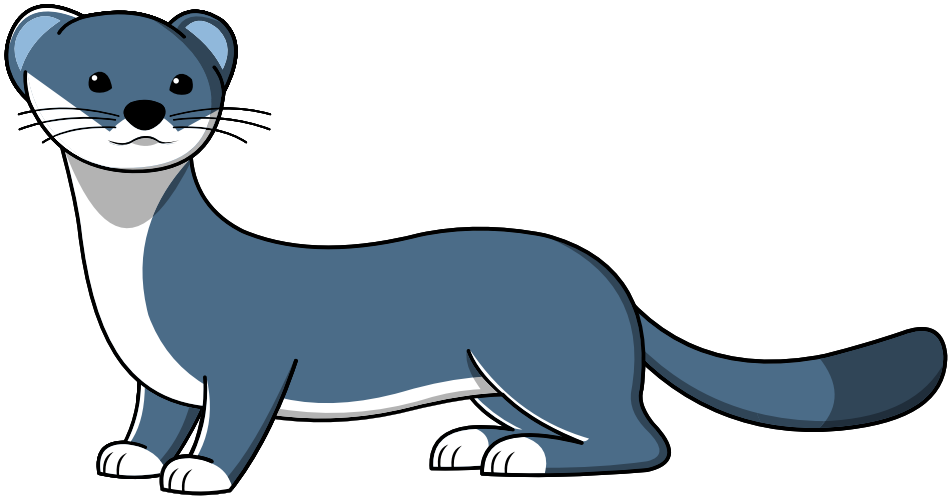
Animal computing mascots: Go



Animal computing mascots: Scratch



Animal computing mascots: V



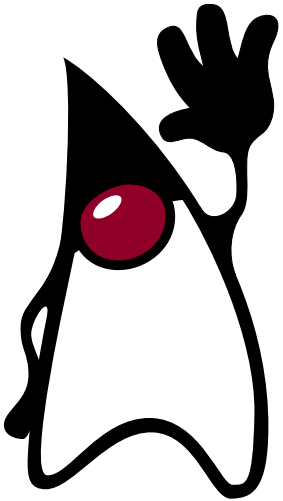
Animal computing mascots: Zig



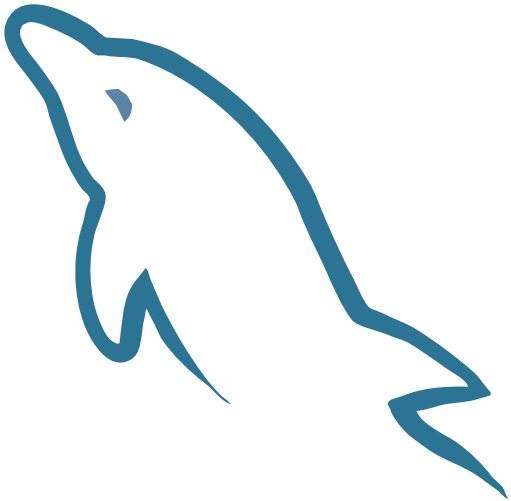
Animal computing mascots: C++



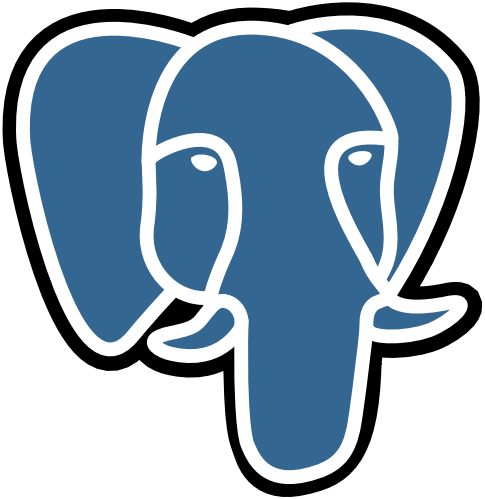
Animal computing mascots: Java



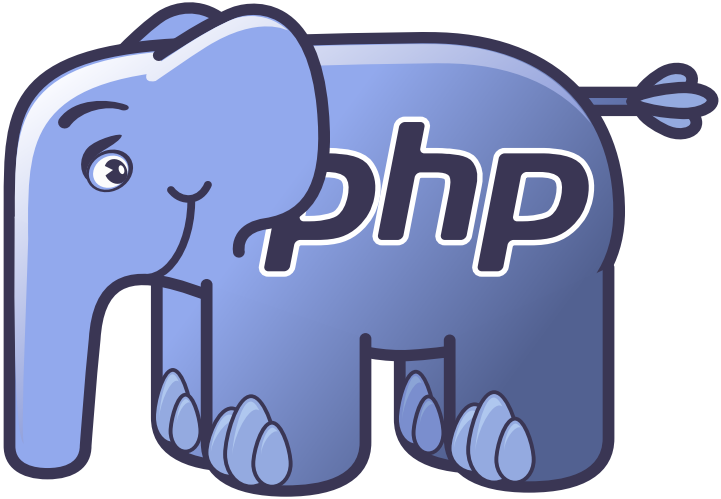
Animal computing mascots: MySQL



Animal computing mascots: Postgres



Animal computing mascots: PHP



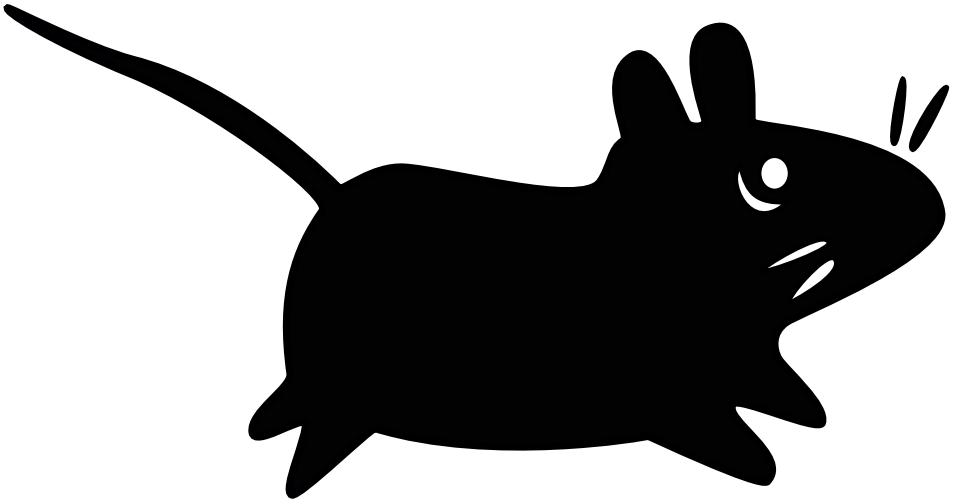
Animal computing mascots: LLVM



Animal computing mascots: KDE



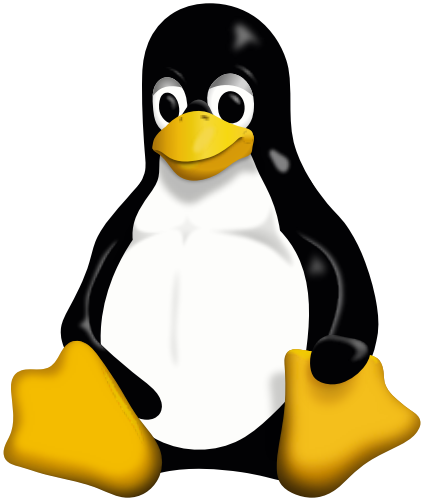
Animal computing mascots: XFCE



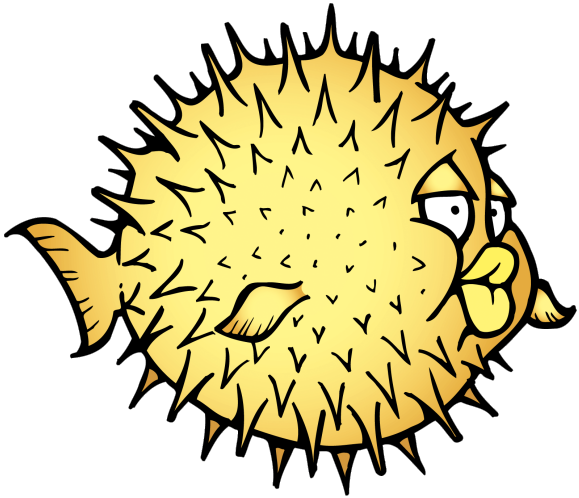
Animal computing mascots: GNU



Animal computing mascots: Linux



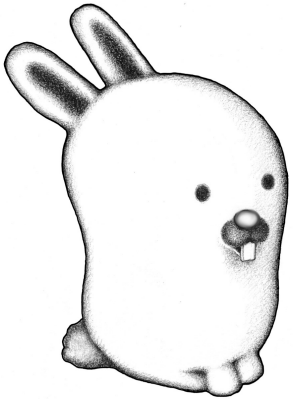
Animal computing mascots: OpenSSH



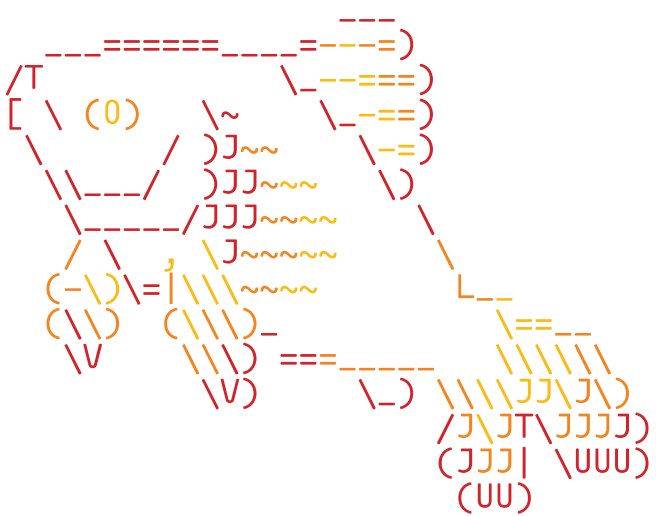
Animal computing mascots: SeaMonkey



Animal computing mascots: Plan9



Animal computing mascots: Fish



Animal computing mascots: Powershell...



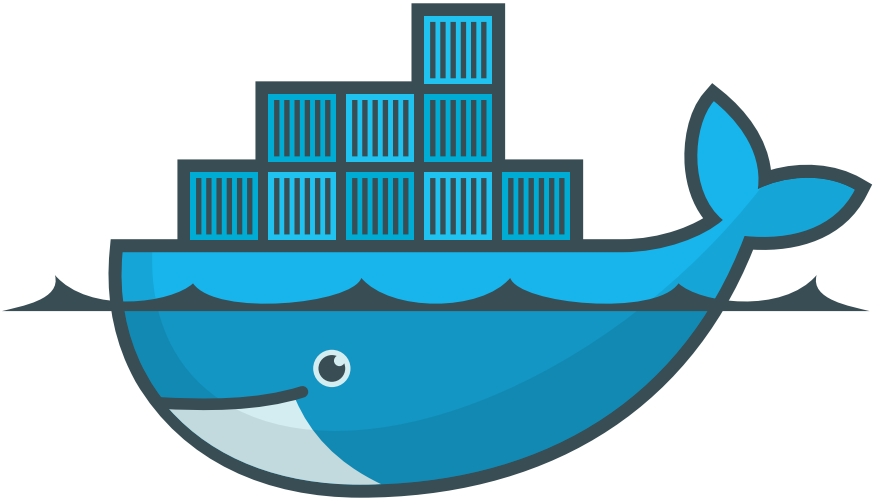
Animal computing mascots: MSI



Animal computing mascots: Ghidra



Animal computing mascots: Docker



Animal computing mascots: GitHub



Animal computing mascots: GitLab



Animal computing mascots: Trans Rights!



Instruction Set Architecture

- CPU Programs = lots of instructions that the CPU steps through to execute
- ISA is how the CPU interprets those instructions.
- Examples: x86, RISC-V etc.
- There are **a lot** of instructions.

x86 ISA: ADD

ADD — Add

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
04 ib	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
05 iw	ADD AX, imm16	I	Valid	Valid	Add imm16 to AX.
05 id	ADD EAX, imm32	I	Valid	Valid	Add imm32 to EAX.
REX.W + 05 id	ADD RAX, imm32	I	Valid	N.E.	Add imm32 sign-extended to 64-bits to RAX.
80 /0 ib	ADD r/m8, imm8	MI	Valid	Valid	Add imm8 to r/m8.
REX + 80 /0 ib	ADD r/m8*, imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m8.
81 /0 iw	ADD r/m16, imm16	MI	Valid	Valid	Add imm16 to r/m16.
81 /0 id	ADD r/m32, imm32	MI	Valid	Valid	Add imm32 to r/m32.
REX.W + 81 /0 id	ADD r/m64, imm32	MI	Valid	N.E.	Add imm32 sign-extended to 64-bits to r/m64.
83 /0 ib	ADD r/m16, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m16.
83 /0 ib	ADD r/m32, imm8	MI	Valid	Valid	Add sign-extended imm8 to r/m32.
REX.W + 83 /0 ib	ADD r/m64, imm8	MI	Valid	N.E.	Add sign-extended imm8 to r/m64.
00 /r	ADD r/m8, r8	MR	Valid	Valid	Add r8 to r/m8.
REX + 00 /r	ADD r/m8*, r8*	MR	Valid	N.E.	Add r8 to r/m8.
01 /r	ADD r/m16, r16	MR	Valid	Valid	Add r16 to r/m16.
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.
02 /r	ADD r8, r/m8	RM	Valid	Valid	Add r/m8 to r8.
REX + 02 /r	ADD r8*, r/m8*	RM	Valid	N.E.	Add r/m8 to r8.

x86 ISA: FADD

FADD/FADDP/FIADD — Add

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /0	FADD m32fp	Valid	Valid	Add m32fp to ST(0) and store result in ST(0).
DC /0	FADD m64fp	Valid	Valid	Add m64fp to ST(0) and store result in ST(0).
D8 C0+i	FADD ST(0), ST(i)	Valid	Valid	Add ST(0) to ST(i) and store result in ST(0).
DC C0+i	FADD ST(i), ST(0)	Valid	Valid	Add ST(i) to ST(0) and store result in ST(i).
DE C0+i	FADDP ST(i), ST(0)	Valid	Valid	Add ST(0) to ST(i), store result in ST(i), and pop the register stack.
DE C1	FADDP	Valid	Valid	Add ST(0) to ST(1), store result in ST(1), and pop the register stack.
DA /0	FIADD m32int	Valid	Valid	Add m32int to ST(0) and store result in ST(0).
DE /0	FIADD m16int	Valid	Valid	Add m16int to ST(0) and store result in ST(0).

FMUL/FMULP/FIMUL — Multiply

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /1	FMUL m32fp	Valid	Valid	Multiply ST(0) by m32fp and store result in ST(0).
DC /1	FMUL m64fp	Valid	Valid	Multiply ST(0) by m64fp and store result in ST(0).
D8 C8+i	FMUL ST(0), ST(i)	Valid	Valid	Multiply ST(0) by ST(i) and store result in ST(0).
DC C8+i	FMUL ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0) and store result in ST(i).
DE C8+i	FMULP ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack
DE C9	FMULP	Valid	Valid	Multiply ST(1) by ST(0), store result in ST(1), and pop the register stac
DA /1	FIMUL m32int	Valid	Valid	Multiply ST(0) by m32int and store result in ST(0).
DE /1	FIMUL m16int	Valid	Valid	Multiply ST(0) by m16int and store result in ST(0).

x86 ISA: VADDPH/VMULPH

VADDPH — Add Packed FP16 Values

Instruction En Bit Mode Flag Support Instruction En Bit Mode Flag Support 64/32 CPUID Feature Instruction En Bit Mode Flag CPUID Feature Instruction En Bit Mode Flag Op/ 64/32 CPUID Feature Instruction En Bit Mode Flag 64/32 CPUID Feature Instruction En Bit Mode Flag CPUID Feature Instruction En Bit Mode Flag Op/ 64/32 CPUID Feature		Support		Description
EVEX.128.NP.MAP5.W0 58 /r VADDPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Add packed FP16 value from xmm3/m128/m16bcst to xmm2, and store result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 58 /r VADDPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Add packed FP16 value from ymm3/m256/m16bcst to ymm2, and store result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 58 /r VADDPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Add packed FP16 value from zmm3/m512/m16bcst to zmm2, and store result in zmm1 subject to writemask k1.

VMULPH — Multiply Packed FP16 Values

Instruction En bit Mode Flag Support Instruction En bit Mode Flag Support 64/32 CPUID Feature Instruction En bit Mode Flag CPUID Feature Instruction En bit Mode Flag Op/ 64/32 CPUID Feature Instruction En bit Mode Flag 64/32 CPUID Feature Instruction En bit Mode Flag CPUID Feature Instruction En bit Mode Flag Op/ 64/32 CPUID Feature		Support		Description
EVEX.128.NP.MAP5.W0 59 /r VMULPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm3/m128/m16bcst to xmm2 and store the result in xmm1 subject to writemask k1.
EVEX.256.NP.MAP5.W0 59 /r VMULPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm3/m256/m16bcst to ymm2 and store the result in ymm1 subject to writemask k1.
EVEX.512.NP.MAP5.W0 59 /r VMULPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values in zmm3/m512/m16bcst with zmm2 and store the result in zmm1 subject to writemask k1.

x86 ISA: VFMADD132PH

VFMADD132PH/VFNMADD132PH/VFMADD213PH/VFNMADD213PH/VFMADD231PH/VFNMADD231PH — Fused Multiply-Add of Packed FP16 Values

Instruction En Bit Mode Flag Support Instruction En Bit Mode Flag Support 64/32 CPUID Feature Instruction En Bit Mode Flag CPUID Feature Instruction En Bit Mode Flag Op/ 64/32 CPUID Feature Instruction En Bit Mode Flag 64/32 CPUID Feature Instruction En Bit Mode Flag CPUID Feature Instruction En Bit Mode Flag Op/ 64/32 CPUID Feature		Support		Description
EVEX.128.66.MAP6.W0 98 /r VFMADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add to xmm2, and store the result in xmm1.
EVEX.256.66.MAP6.W0 98 /r VFMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add to ymm2, and store the result in ymm1.
EVEX.512.66.MAP6.W0 98 /r VFMADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add to zmm2, and store the result in zmm1.
EVEX.128.66.MAP6.W0 A8 /r VFMADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm1 and xmm2, add to xmm3/m128/ m16bcst, and store the result in xmm1.
EVEX.256.66.MAP6.W0 A8 /r VFMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm1 and ymm2, add to ymm3/m256/ m16bcst, and store the result in ymm1.
EVEX.512.66.MAP6.W0 A8 /r VFMADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values from zmm1 and zmm2, add to zmm3/m512/ m16bcst, and store the result in zmm1.
EVEX.128.66.MAP6.W0 B8 /r VFMADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, add to xmm1, and store the result in xmm1.
EVEX.256.66.MAP6.W0 B8 /r VFMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, add to ymm1, and store the result in ymm1.
EVEX.512.66.MAP6.W0 B8 /r VFMADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, add to zmm1, and store the result in zmm1.
EVEX.128.66.MAP6.W0 9C /r VFNMADD132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm2, and store the result in xmm1.
EVEX.256.66.MAP6.W0 9C /r VFNMADD132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm2, and store the result in ymm1.
EVEX.512.66.MAP6.W0 9C /r VFNMADD132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm2, and store the result in zmm1.
EVEX.128.66.MAP6.W0 AC /r VFNMADD213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm1 and xmm2, and negate the value. Add this value to xmm3/m128/m16bcst, and store the result in xmm1.
EVEX.256.66.MAP6.W0 AC /r VFNMADD213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm1 and ymm2, and negate the value. Add this value to ymm3/m256/m16bcst, and store the result in ymm1.
EVEX.512.66.MAP6.W0 AC /r VFNMADD213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values from zmm1 and zmm2, and negate the value. Add this value to zmm3/m512/m16bcst, and store the result in zmm1.
EVEX.128.66.MAP6.W0 BC /r VFNMADD231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm1, and store the result in xmm1.
EVEX.256.66.MAP6.W0 BC /r VFNMADD231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm1, and store the result in ymm1.
EVEX.512.66.MAP6.W0 BC /r VFNMADD231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm1, and store the result in zmm1.

x86 ISA: VFMADDSUB132PH

VFMADDSUB132PH/VFMADDSUB213PH/VFMADDSUB231PH — Fused Multiply-AlternatingAdd/Subtract of Packed FP16 Values

Instruction En Bit Mode Flag Support Instruction En Bit Mode Flag Support 64/32 CPUID Feature Instruction En Bit Mode Flag CPUID Feature Instruction En Bit Mode Flag Op/ 64/32 CPUID Feature Instruction En Bit Mode Flag 64/32 CPUID Feature Instruction En Bit Mode Flag CPUID Feature Instruction En Bit Mode Flag Op/ 64/32 CPUID Feature		Support		Description
EVEX.128.66.MAP6.W0 96 /r VFMADDSUB132PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add/subtract elements in xmm2, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 96 /r VFMADDSUB132PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add/subtract elements in ymm2, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 96 /r VFMADDSUB132PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add/subtract elements in zmm2, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 A6 /r VFMADDSUB213PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 A6 /r VFMADDSUB213PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 A6 /r VFMADDSUB213PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.
EVEX.128.66.MAP6.W0 B6 /r VFMADDSUB231PH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, add/subtract elements in xmm1, and store the result in xmm1 subject to writemask k1.
EVEX.256.66.MAP6.W0 B6 /r VFMADDSUB231PH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst	A	V/V	AVX512-FP16 AVX512VL	Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, add/subtract elements in ymm1, and store the result in ymm1 subject to writemask k1.
EVEX.512.66.MAP6.W0 B6 /r VFMADDSUB231PH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {er}	A	V/V	AVX512-FP16	Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, add/subtract elements in zmm1, and store the result in zmm1 subject to writemask k1.

x86 ISA: VPCLMULQDQ

PCLMULQDQ—Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ *xmm1, xmm2/m128, imm8*	A	V/V	PCLMULQDQ	Carry-less multiplication of one quadword of xmm1 by one quadword of *xmm2/m128*, stores the 128-bit result in *xmm1*. The immediate is used to determine which quadwords of *xmm1* and *xmm2/m128* should be used.
VEX.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ *xmm1, xmm2, xmm3/m128, imm8*	B	V/V	PCLMULQDQ AVX	Carry-less multiplication of one quadword of *xmm2* by one quadword of *xmm3/m128*, stores the 128-bit result in *xmm1*. The immediate is used to determine which quadwords of *xmm2* and *xmm3/m128* should be used.
VEX.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8	B	V/V	VPCLMULQDQ	Carry-less multiplication of one quadword of ymm2 by one quadword of ymm3/m256, stores the 128-bit result in ymm1. The immediate is used to determine which quadwords of ymm2 and ymm3/m256 should be used.

x86 ISA: VPHMINPOSUW

PHMINPOSUW — Packed Horizontal Word Minimum

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 41 /r PHMINPOSUW *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Find the minimum unsigned word in *xmm2/m128* and place its value in the low word of *xmm1* and its index in the second-lowest word of *xmm1*.
VEX.128.66.0F38.WIG 41 /r VPHMINPOSUW *xmm1, xmm2/m128*	RM	V/V	AVX	Find the minimum unsigned word in *xmm2/m128* and place its value in the low word of xmm1 and its index in the second-lowest word of *xmm1*.

x86 ISA: VMASKMOVDQU

MASKMOVDQU—Store Selected Bytes of Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F F7 /r MASKMOVDQU *xmm1, xmm2*	RM	V/V	SSE2	Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:DI/EDI/RDI.
VEX.128.66.0F.WIG F7 /r VMASKMOVDQU *xmm1, xmm2*	RM	V/V	AVX	Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:DI/EDI/RDI.

x86 ISA: VPCMPISTRI

PCMPISTRI — Packed Compare Implicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 63 */r imm8* PCMPISTRI *xmm1, xmm2/m128, imm8*	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A.WIG 63 /r ib VPCMPISTRI *xmm1, xmm2/m128, imm8*	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.

x86 ISA: CMPXCHG16B

CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C7 /1 CMPXCHG8B *m64*	M	Valid	Valid*	Compare EDX:EAX with *m64*. If equal, set ZF and load ECX:EBX into *m64*. Else, clear ZF and load *m64* into EDX:EAX.
REX.W + 0F C7 /1 CMPXCHG16B *m128*	M	Valid	N.E.	Compare RDX:RAX with *m128*. If equal, set ZF and load RCX:RBX into *m128*. Else, clear ZF and load *m128* into RDX:RAX.

NOTES:

*See IA-32 Architecture Compatibility section below.

x86 ISA: VGF2P8AFFINEINVQB

GF2P8AFFINEINVQB—Galois Field Affine Transformation Inverse

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CF /r /ib GF2P8AFFINEINVQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.512.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F GFNI	Computes inverse affine transformation in the finite field GF(2^8).

x86 ISA: ROL

SAR	Shift arithmetic right.
SHR	Shift logical right.
SAL/SHL	Shift arithmetic left/Shift logical left.
SHRD	Shift right double.
SHLD	Shift left double.
ROR	Rotate right.
ROL	Rotate left.
RCR	Rotate through carry right.
RCL	Rotate through carry left.

x86 ISA: FIST

FIST/FISTP—Store Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /2	FIST *m16int*	Valid	Valid	Store ST(0) in *m16int*.
DB /2	FIST *m32int*	Valid	Valid	Store ST(0) in *m32int*.
DF /3	FISTP *m16int*	Valid	Valid	Store ST(0) in *m16int* and pop register stack.
DB /3	FISTP *m32int*	Valid	Valid	Store ST(0) in *m32int* and pop register stack.
DF /7	FISTP *m64int*	Valid	Valid	Store ST(0) in *m64int* and pop register stack.

x86 ISA: KNOTD

KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 44 /r KNOTW k1, k2	RR	V/V	AVX512F	Bitwise NOT of 16 bits mask k2.
VEX.L0.66.0F.W0 44 /r KNOTB k1, k2	RR	V/V	AVX512DQ	Bitwise NOT of 8 bits mask k2.
VEX.L0.0F.W1 44 /r KNOTQ k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 64 bits mask k2.
VEX.L0.66.0F.W1 44 /r KNOTD k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 32 bits mask k2.



NeedsMoreGPUs • 4y ago

Two Star Pixel Defender

Of course it's AVX512. AVX512 is the most degenerate instruction set.

7 Share ···

PowerPC: EIEIO

eieio (Enforce In-Order Execution of I/O) Instruction

Purpose

Ensures that cache-inhibited storage accesses are performed in main memory in the order specified by the program.

Note: The **eieio** instruction is supported only in the PowerPC architecture.

Motorolla 6809: SEX

SEX **Sign Extended** **SEX**

Source Form: SEX

Operation: If bit seven of ACCB is set then ACCA′ ← FF_{16}
else ACCA′ ← 00_{16}

Condition Codes:
H — Not affected.
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Not affected.
C — Not affected.

Description: This instruction transforms a twos complement 8-bit value in accumulator B into a twos complement 16-bit value in the D accumulator.

Addressing Mode: Inherent

Overview

- I don't have puns to present TwT
  - I need to graduate
  - grad school got hands
- Here's my research instead

Requirement	Status	Date	Comments
TQE	approved	Fall 2026	**due at end of 5 terms**
SM Proposal	completed	Summer 2026	6.9991@F25: J- 6.9991@S26: U
SM Thesis			6.9992@F26: **due after 4 terms**
RQE			**apply in 5th term**
PhD Proposal			due at end of 8 terms
PhD Thesis			
Minor Requirement			due at end of 9 terms
Teaching	completed	Fall 2026	6.9900@S24: P [EECS TA [100%] assignment: 6.1903 (TA)]
Professional Perspective			6.9940: – 6.9950: – due at end of 10 terms

Intro to programming

- Sequential code is a sequence of instructions.
  - Read numbers from memory.
  - Add these numbers.
  - Store the result back to memory.
  - If this number is nonzero, go to that line.
- Modern processors do not exceed 10 billion sequential instructions per second.
  - Clock frequencies have plateaued (Dennard scaling).
  - In the time it took for light to travel from the projector to your eyes, my laptop executed $> 3\text{GHz}/(c/2\text{m}) \approx 20$ instructions.

Intro to parallel programming

- What if many processors worked on the same problem at the same time?
- Benefit: Better performance!
  - Having p processors means your runtime can be divided by p.
- Downside: Race conditions!
  - Now you need to coordinate p processors. Good luck >:3

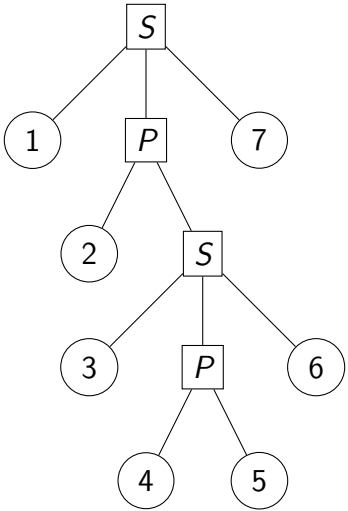
```
a = 0
parallel_for (i in 0,...,10^9) {
    a += 1
}
print(a)
```

Processors	Time	Result
1 Processor	2.504s	1000000000
5 Processor	0.507s	925316000[a]

[a] This is suspiciously a multiple of 1000 because I fudged the implementation for the demo.

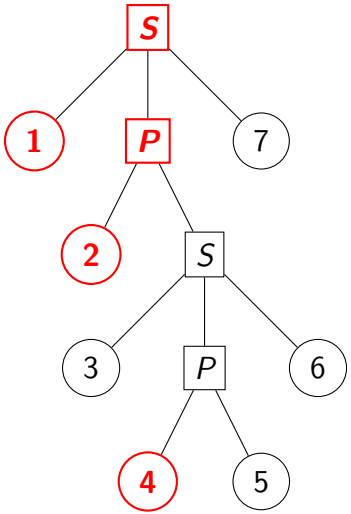
Intro to fork-join parallelism

- Model program execution as an *SP*-**tree**.
  - Individual operations are at leaves.
  - Internal nodes are labeled S or P.
  - Children of S nodes must run sequentially.
    - All operations in child i must finish before any operations in child $i+1$ can start.
  - Children of P nodes may execute in parallel.
    - A scheduler decides which operations actually execute in parallel.
    - *handwave* Blumofe and Leiserson (1999).
- Benefit: Easier analysis than general threading.
- Downside: Less general than general threading.



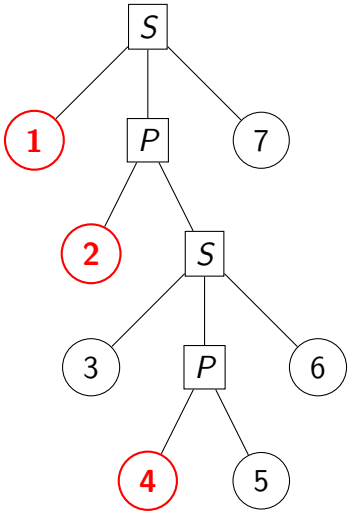
Logical parallelism analysis

- Operation u **logically precedes** v if their least common ancestor (LCA) is a S node, and u is to the left of v.
  - Equivalently, v **logically succeeds** u.
- Two operations are **logically parallel** if their LCA is a P node.
  - We make a distinction about logical parallelism because scheduling decisions might make logically parallel operations not actually execute in parallel.



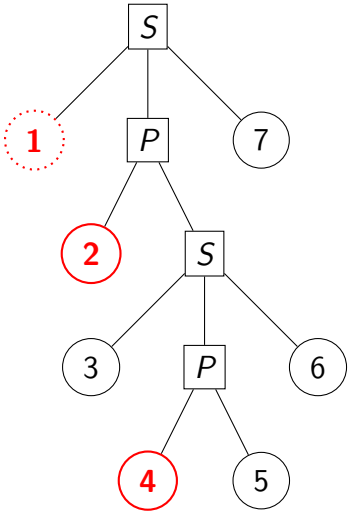
Determinacy-race detection

- Two memory operations are a **determinacy race** if they access the same location, are logically parallel, and at least one is a write.
- A determinacy-race detection algorithm must monitor all memory accesses.
- Each time the program makes a new memory access we must:
  - Query if any previous memory access to the same location is logically parallel with the new access.
  - Update our data structures to account for the new memory access.



Active nodes

- We want to summarize a set of accesses V so that for any access w, we know if there exists some $v \in V$ that is logically parallel with w.
  - Assume w doesn't logically precede any $v \in V$.
- Suppose $u \in V$ logically precedes $v \in V$.
  - If w logically succeeds v, then w logically succeeds u.
  - If w is logically parallel with u, then w is logically parallel with v.
- u is **inactive** and we can forget about it.
  - Any $u \in V$ that isn't logically succeeded by any $v \in V$ is **active**.

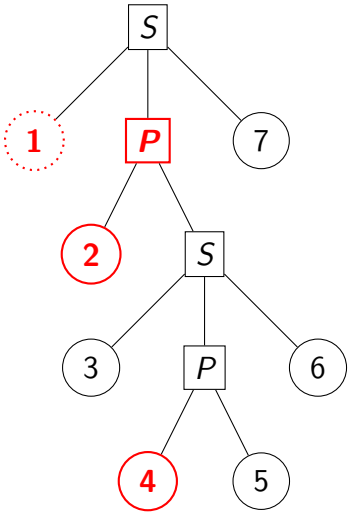


The representative node

- Any $u \in V$ that isn't logically succeeded by any $v \in V$ is **active**.
- Let the **representative** of V be the LCA of the active nodes of V, notated as $\widehat{V}$.
  - By properties of LCA, w is logically parallel with some active node if and only if w is logically parallel with $\widehat{V}$.

Theorem

For any w not logically preceding any $v \in V$, w is logically parallel with some $v \in V$ if and only if w is logically parallel with the representative of V.



Data Structures & Algorithms @ Furpocalypse 2026

Rocq and ROL Nightmare

Sunny, Unsigned Long

November 1, 2026



Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. J. ACM, 46(5):720–748, September 1999. ISSN 0004-5411. doi: 10.1145/324133.324234. URL https://doi.org/10.1145/324133.324234.