Register Allocation, ii

Liveness analysis & Graph coloring & their interplay

Liveness analysis

Fixed point computation:

- Define which variables are set and which are used, for each instruction: **kill** & **gen** functions
- Specify how nearby instructions transmit live values around the program: **in** & **out** functions
- Iterate until nothing changes

kill, gen : i → register set

kill(s) = {all assigned registers in stm s}

gen(s) = {all referenced registers in stm s}

Gen & Kill

		gen	kill
1:	`:f`	()	()
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2)	(2x2)
6:	`(3x <- eax)`	(eax)	(3x)
7:	`(3x *= 3)`	(3x)	(3x)
8:	`(eax <- 2x2)`	(2x2)	(eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(eax)	()

in, out : Nat → register set

$$in(n) = gen(n\text{-}th\text{-}inst) \cup (out(n) - kill(n\text{-}th\text{-}inst))$$

$$out(n) = \cup\{in(m) \mid m \in succ(n)\}$$

Intuition behind the definitions: if $x \in$ **in**(n) then we know that x is live between instructions n-1 and n. Similarly, if $x \in$ **out**(n) then we know that x is live between instructions n and n+1.

in, out : Nat → register set

$$in(n) = gen(n\text{-}th\text{-}inst) \cup (out(n) - kill(n\text{-}th\text{-}inst))$$

$$out(n) = \cup\{in(m) \mid m \in succ(n)\}$$

One possible solution: every variable is in **in**(n) and in **out**(n) for every n.

But that isn't super helpful... since we want to know what values don't need to be in registers at each point.

in, out : Nat → register set

$in(n) = gen(n\text{-}th\text{-}inst) \cup (out(n) - kill(n\text{-}th\text{-}inst))$

$out(n) = \cup\{in(m) \mid m \in succ(n)\}$

Instead, compute **in** and **out** by iterating the equations until we reach a fixed point. Starting out with the assumption that they map everything to the empty set. Then repeatedly apply the right hand sides until nothing changes.

That result will satisfy the definitions of **in** and **out** but will it be the smallest solution?

Liveness

		in	out
1:	`:f`	()	()
2:	`(x2 <- eax)`	()	()
3:	`(x2 *= x2)`	()	()
4:	`(2x2 <- x2)`	()	()
5:	`(2x2 *= 2)`	()	()
6:	`(3x <- eax)`	()	()
7:	`(3x *= 3)`	()	()
8:	`(eax <- 2x2)`	()	()
9:	`(eax += 3x)`	()	()
10:	`(eax += 4)`	()	()
11:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	()	()
2:	`(x2 <- eax)`	(eax)	()
3:	`(x2 *= x2)`	(x2)	()
4:	`(2x2 <- x2)`	(x2)	()
5:	`(2x2 *= 2)`	(2x2)	()
6:	`(3x <- eax)`	(eax)	()
7:	`(3x *= 3)`	(3x)	()
8:	`(eax <- 2x2)`	(2x2)	()
9:	`(eax += 3x)`	(3x eax)	()
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	(eax)	()

Liveness

		in	out
1:	`:f`	()	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2)	(eax)
6:	`(3x <- eax)`	(eax)	(3x)
7:	`(3x *= 3)`	(3x)	(2x2)
8:	`(eax <- 2x2)`	(2x2)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(eax)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2 eax)	(eax)
6:	`(3x <- eax)`	(eax)	(3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(eax)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(eax)
6:	`(3x <- eax)`	(eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(eax)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(eax)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(eax)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(eax)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(eax)	()

Graph coloring

Reduce register allocation to the *graph coloring* problem

- Nodes represent variables
- Edges connect nodes that cannot share a register
- Colors represent registers

If we can color the graph (where no adjacent nodes share a color) in N colors, then we can compile this program to use N registers

Graph coloring

Build interference graph from the liveness information

- For each instruction:
  - Two variables live at the same time interfere with each other
  - Killed variables interfere with variables in the out set
  - Except that the variables **`x`** and **`y`** do not interfere if the instruction was **`(x <- y)`**
- All real registers interfere with each other

Graph coloring

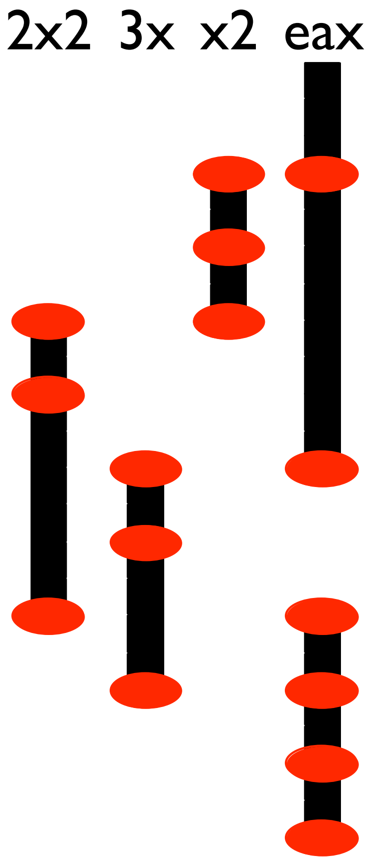
What counts as live "at the same time"?

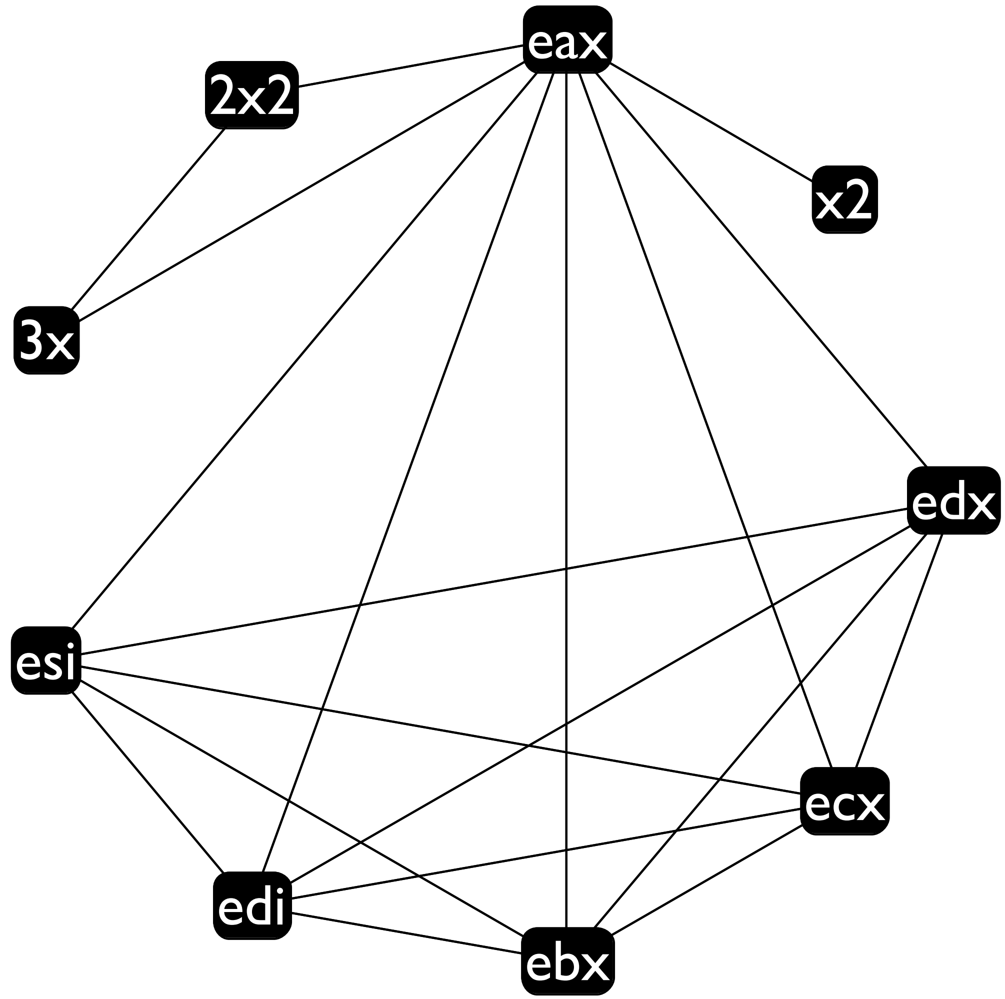
- We could say that two variables are live at the same time if they appear in either the 'in' or the 'out' set together for any instruction
- But, not counting instructions that have no predeccessors, every variable pair that is in an 'in' set, is also in an 'out' set
- Also, the only instruction that has no precedessor that actually gets run is the first one

So, live at the same time means appearing together in an 'out' set, or appearing together in the first instruction's 'in' set

Liveness ⇒ Interference graph

```
:f
(x2 <- eax)
(x2 *= x2)
(2x2 <- x2)
(2x2 *= 2)
(3x <- eax)
(3x *= 3)
(eax <- 2x2)
(eax += 3x)
(eax += 4)
(return)
```





Check yourself

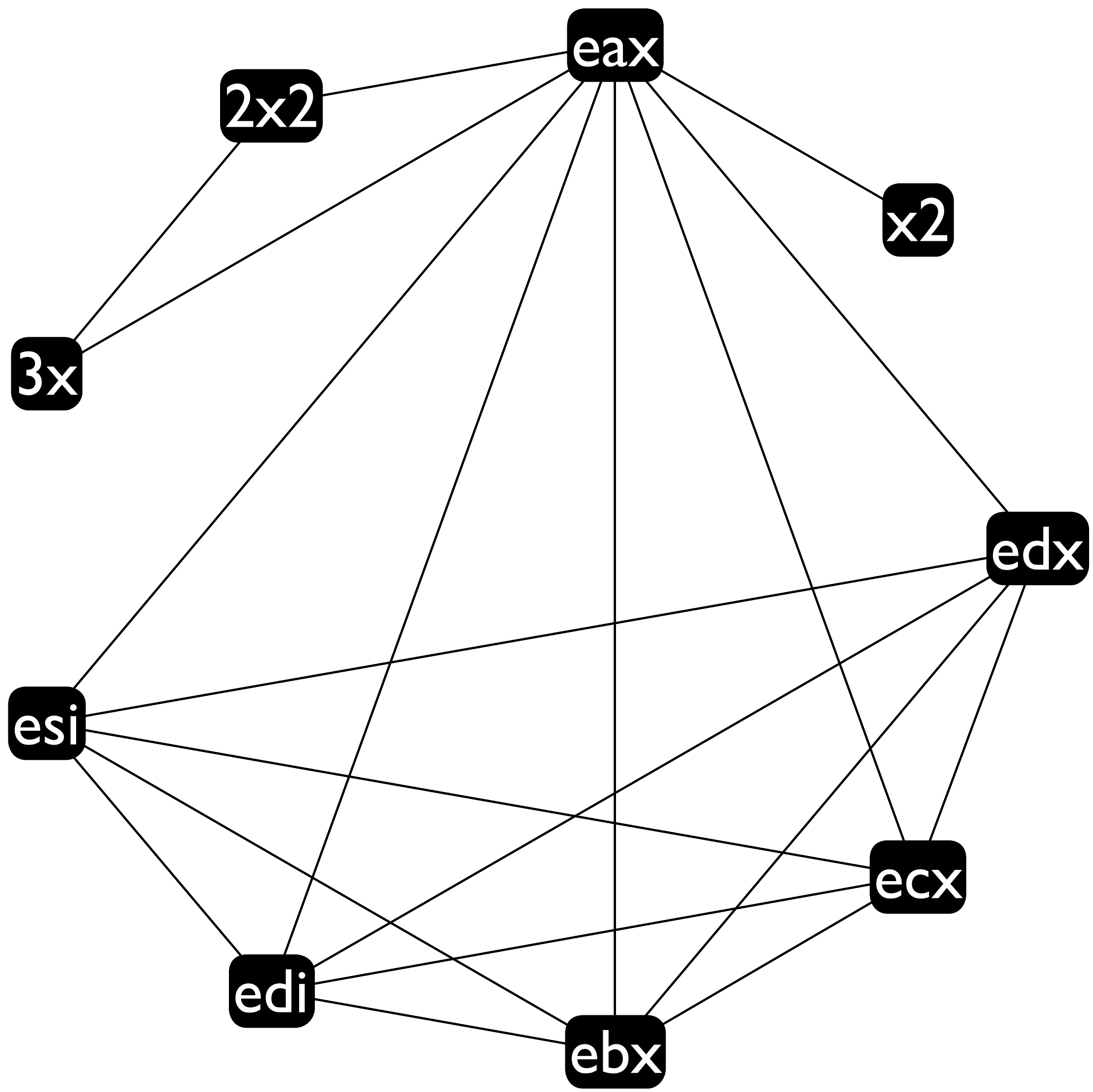
For each pair of x2, 2x2, 3x, and eax:

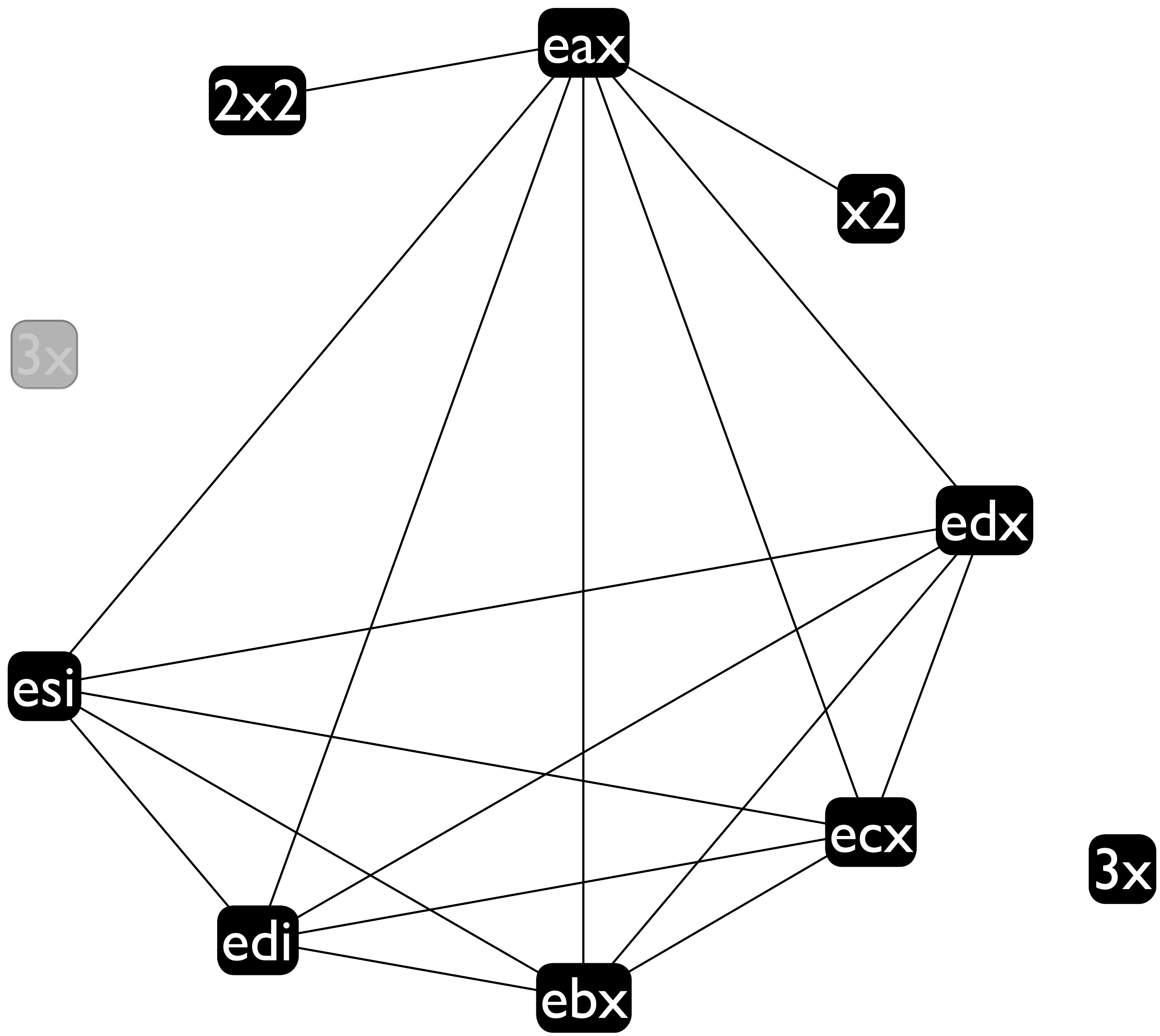
- do they interfere?;
- should they?

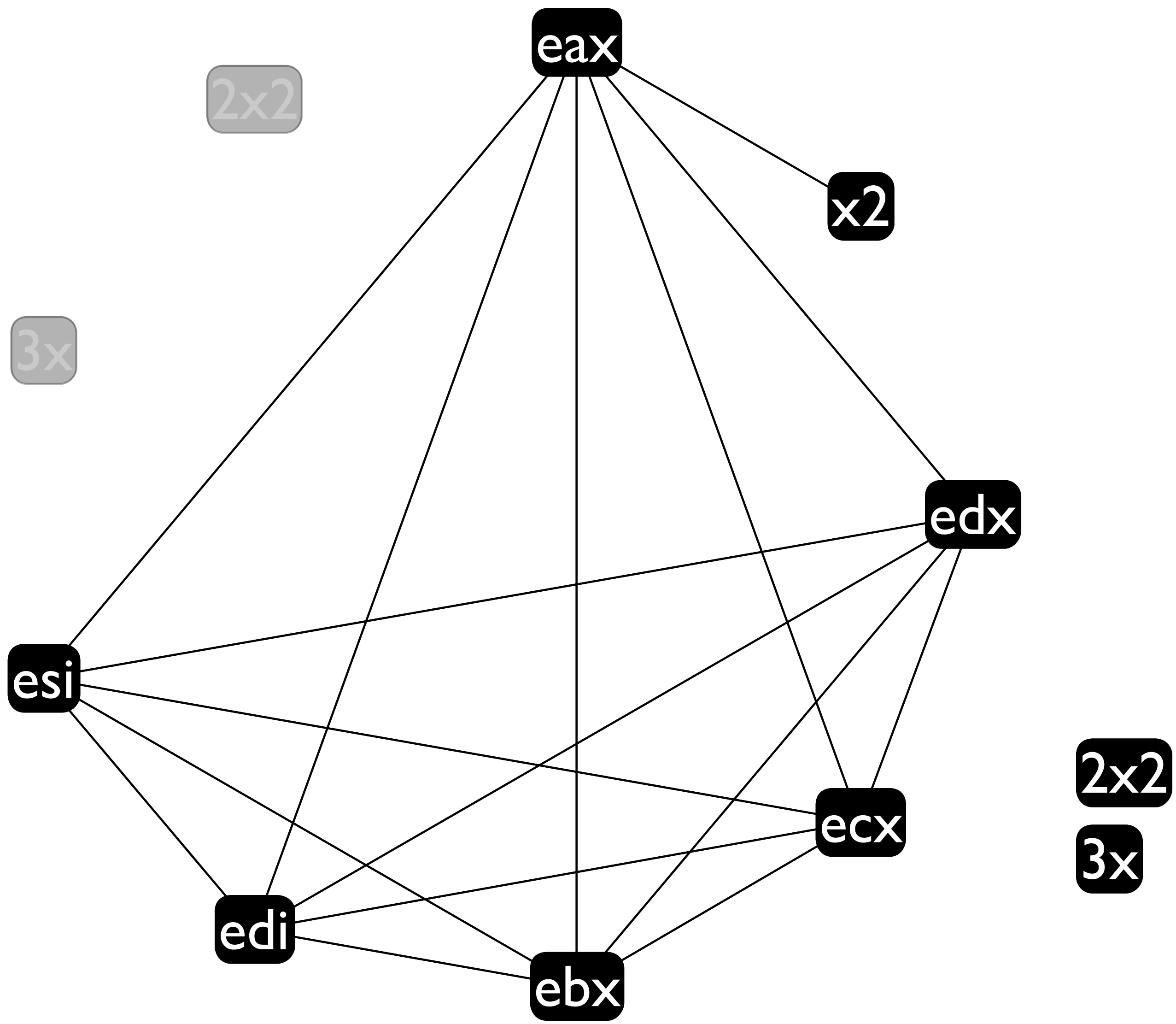
Graph coloring

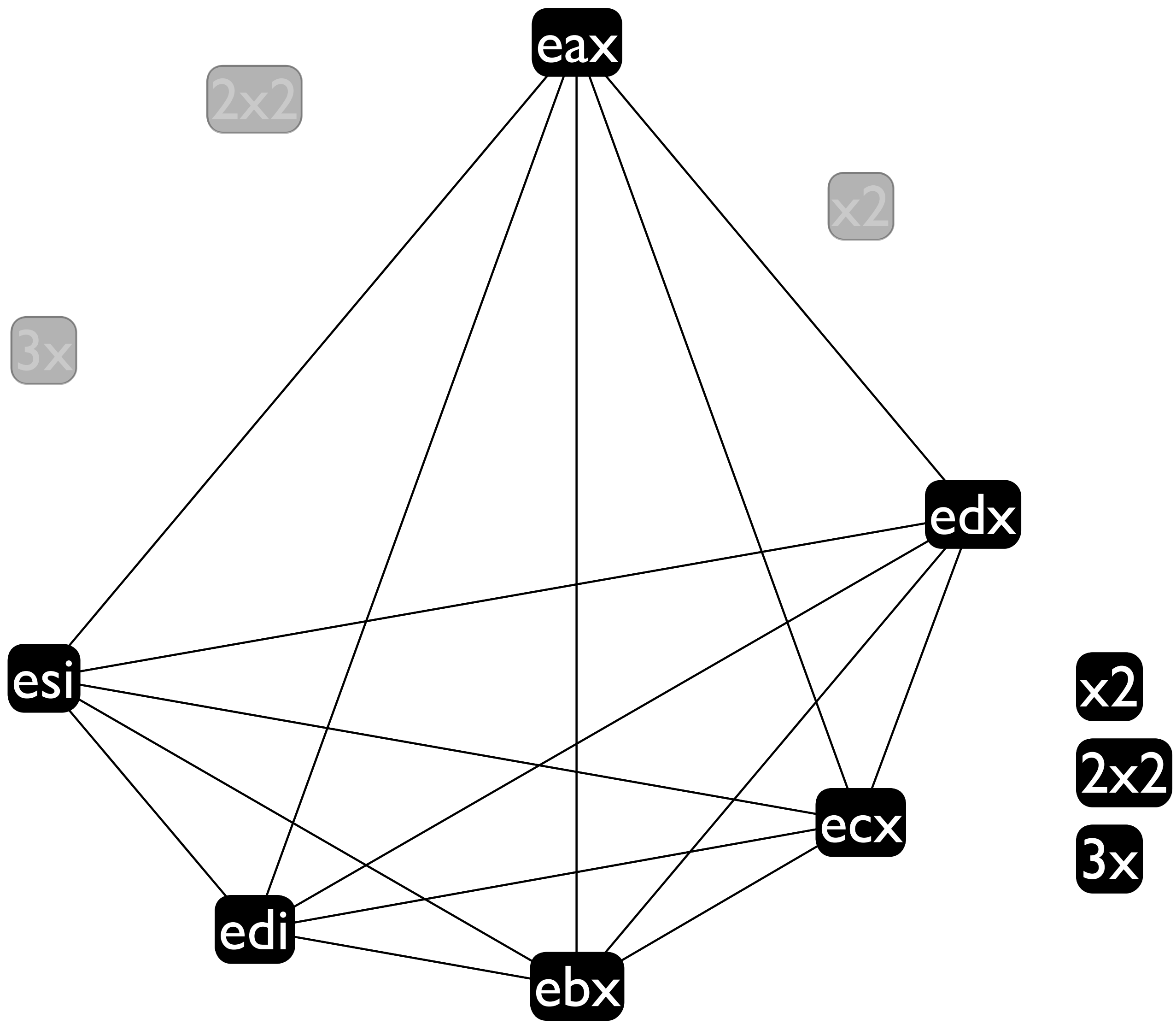
Algorithm:

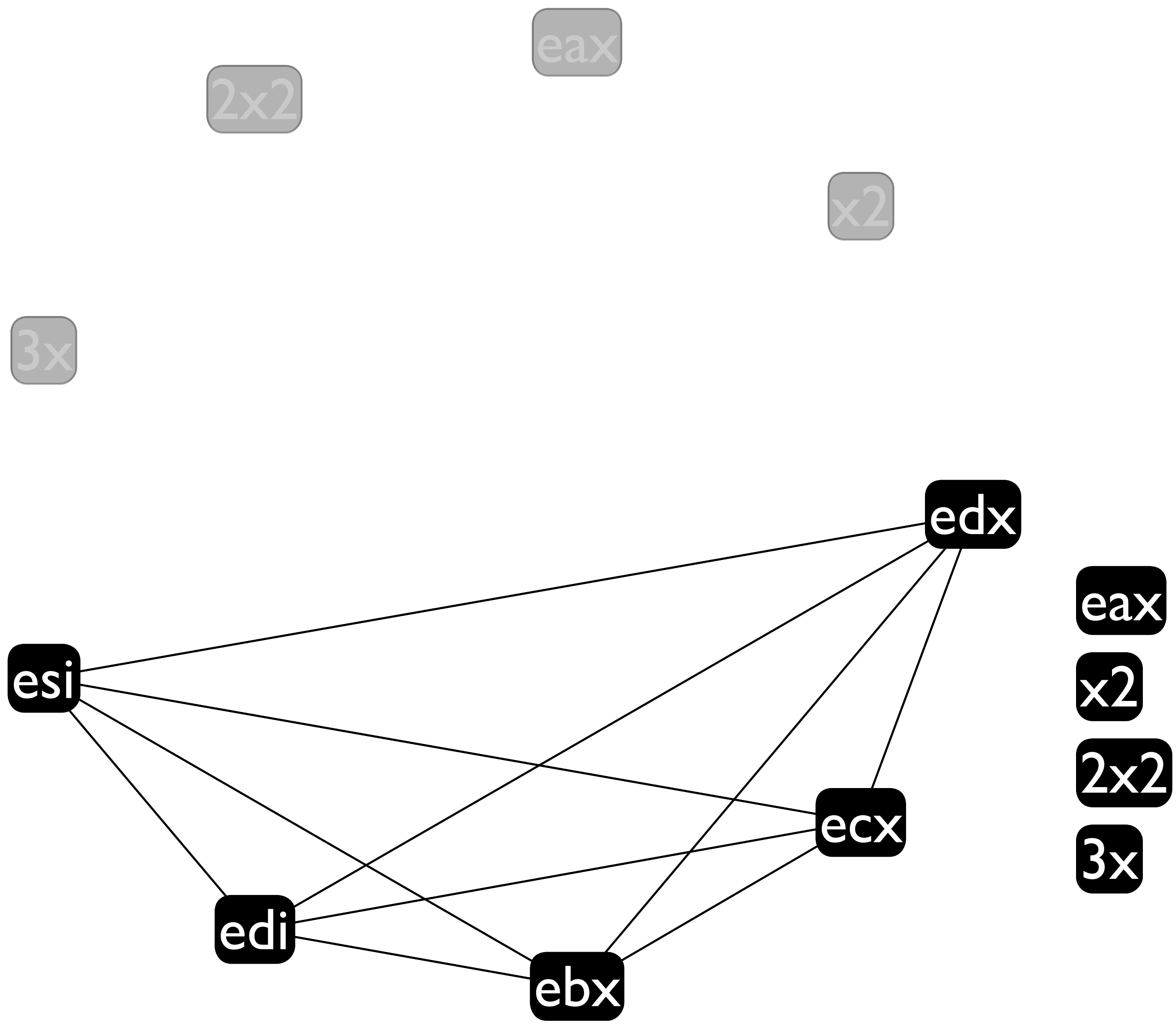
- Repeatedly select a node and remove it from the graph, putting it onto a stack
- When the graph is empty, rebuild it, putting a new color on each node as it comes back into the graph, making sure no adjacent nodes have the same color
- If there are not enough colors, the algorithm fails (spilling comes in here)
- Make sure real registers come out of the graph last so they get the first 6 colors

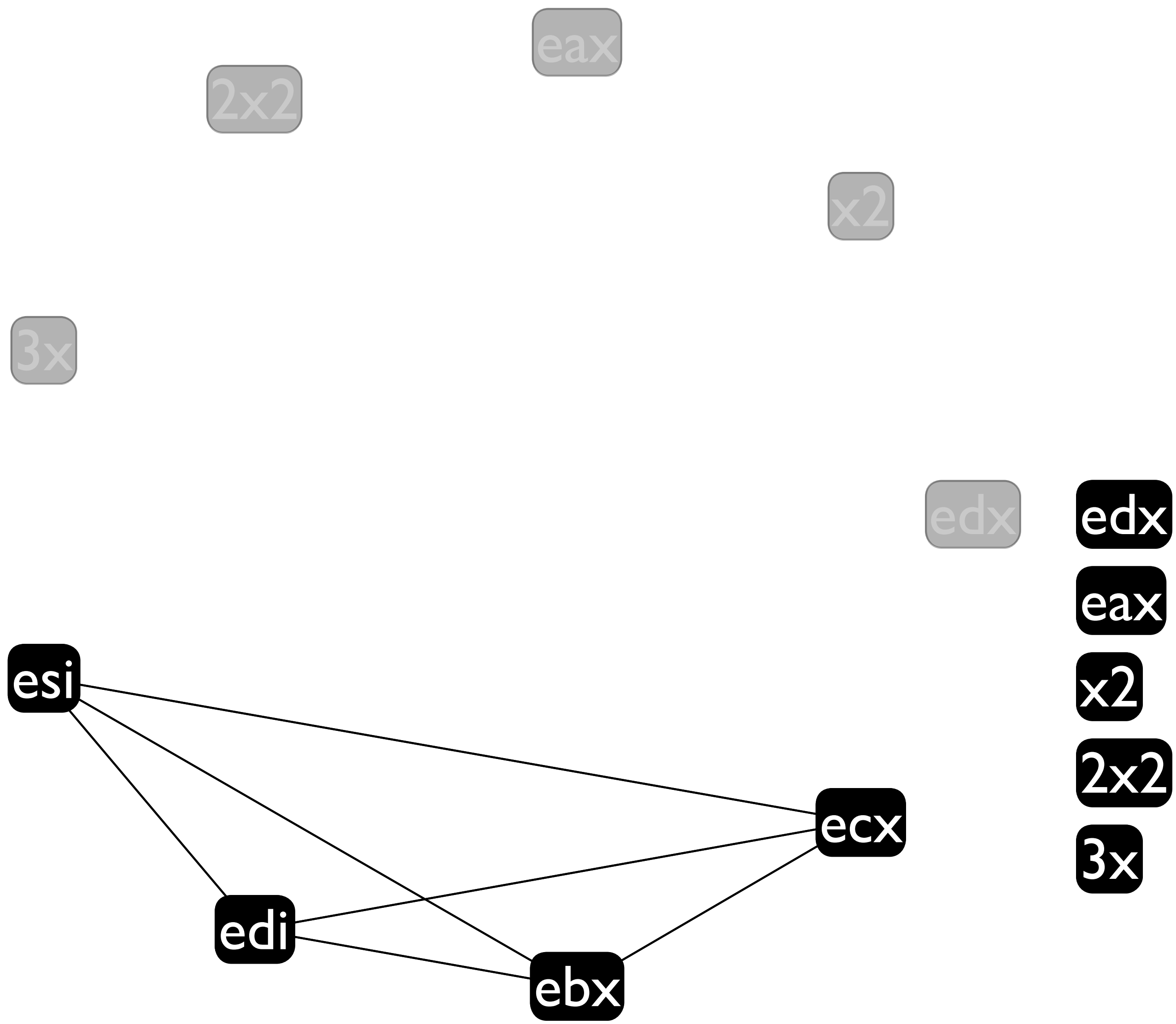

eax
2x2
x2
3x
edx
esi
ecx
edi
ebx

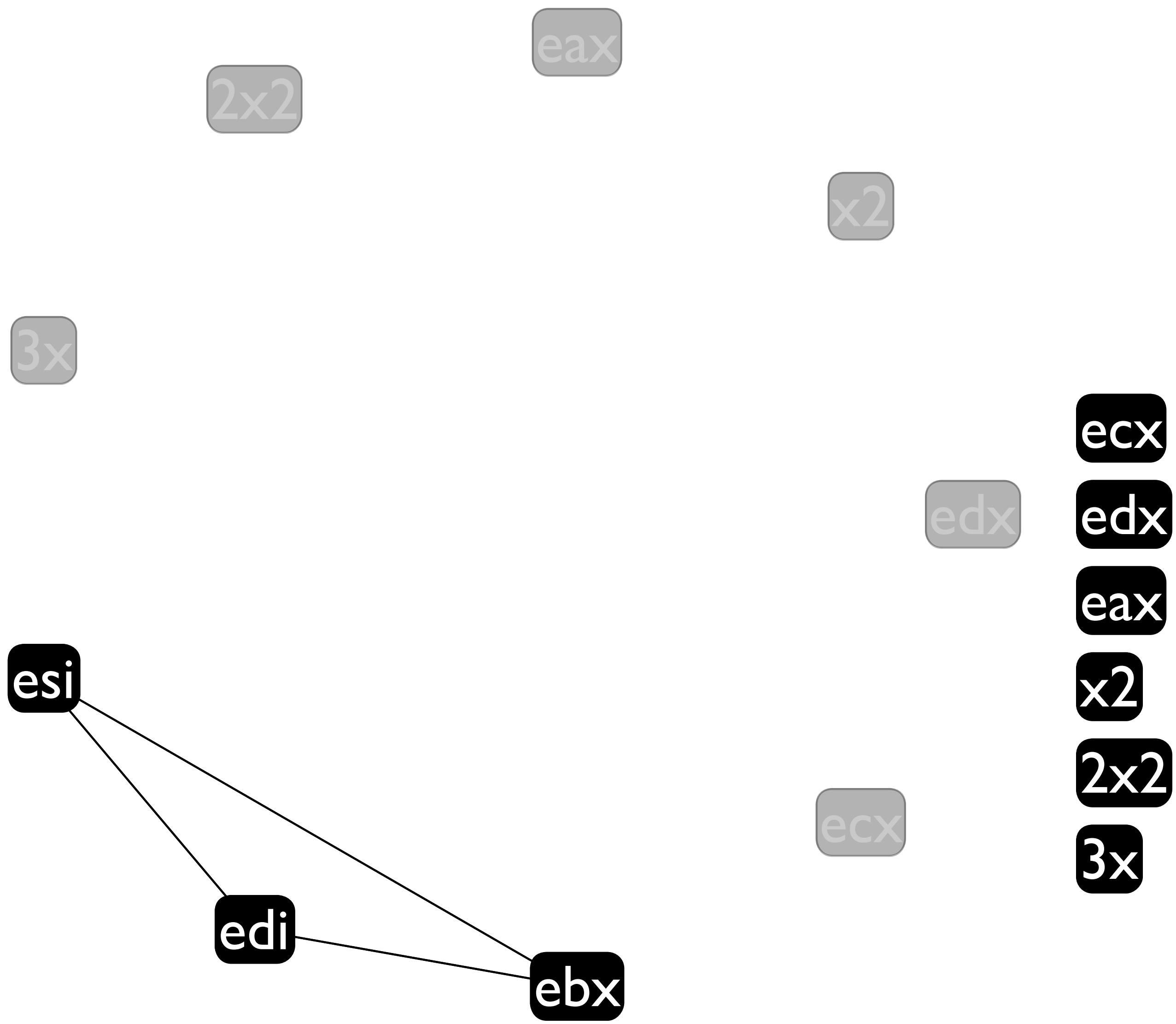

eax
2x2
x2
3x
edx
esi
ecx
3x
edi
ebx

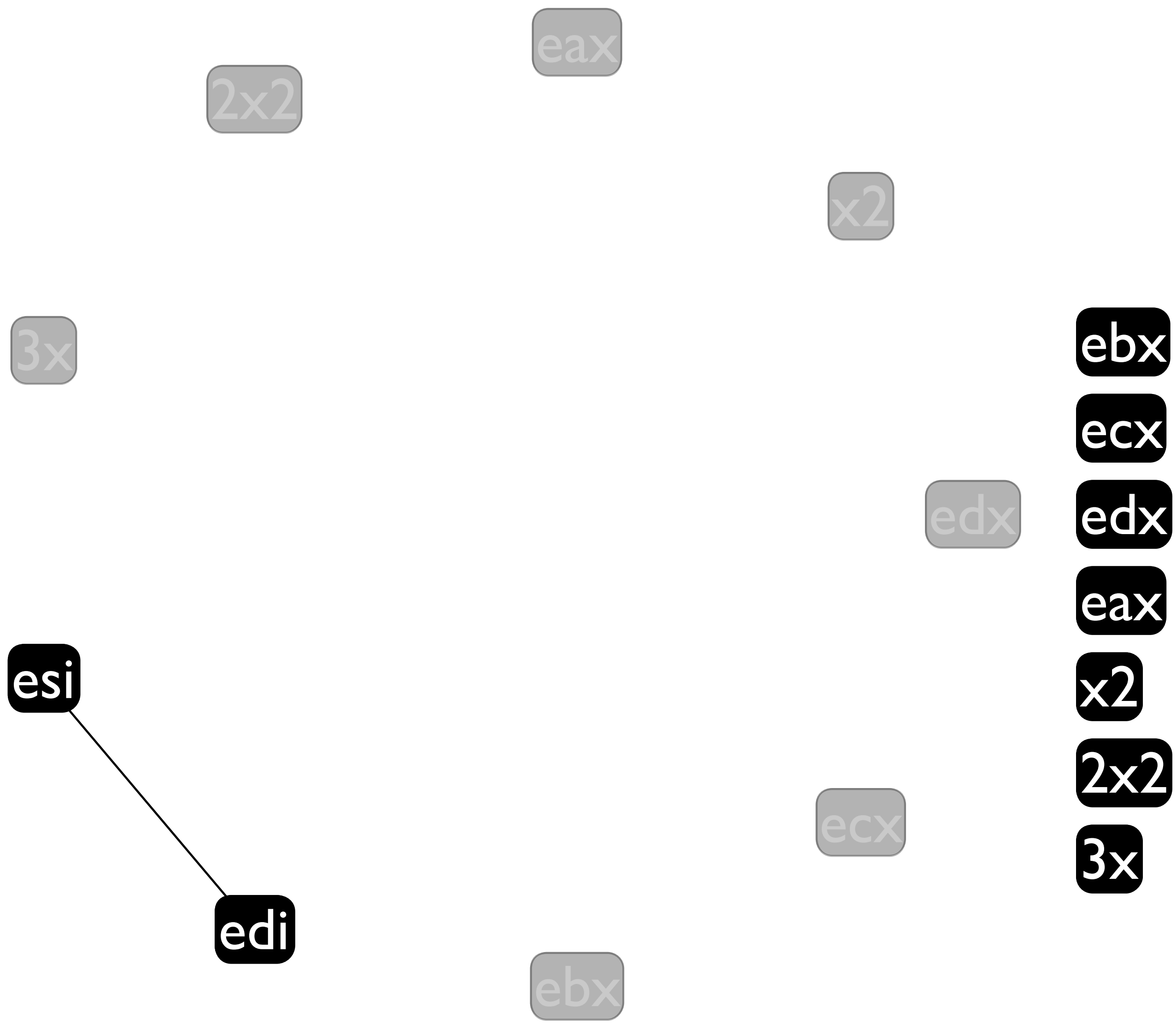

eax
2x2
x2
3x
edx
esi
2x2
ecx
3x
edi
ebx

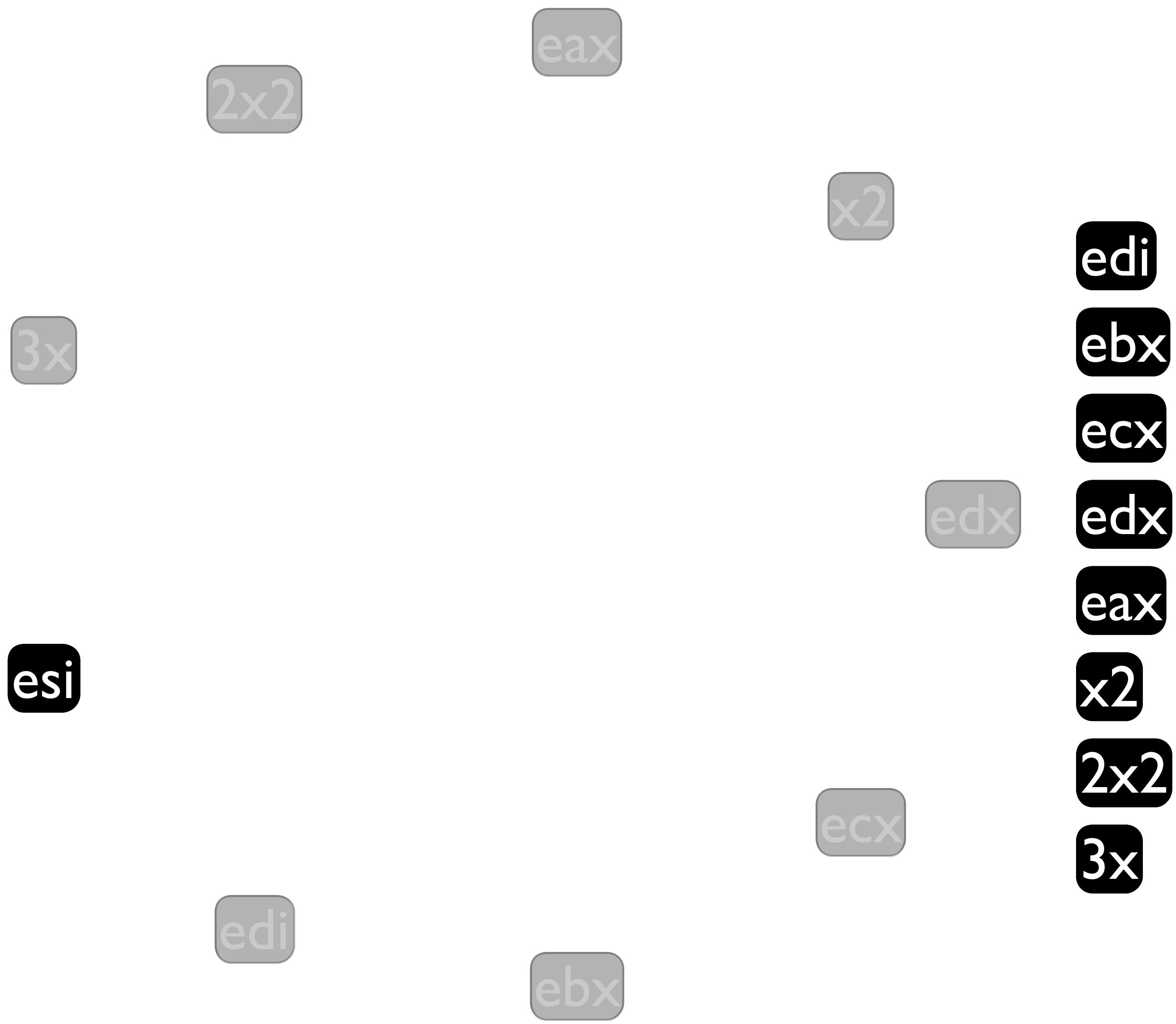

eax
2x2
x2
3x
edx
esi
x2
2x2
ecx
3x
edi
ebx

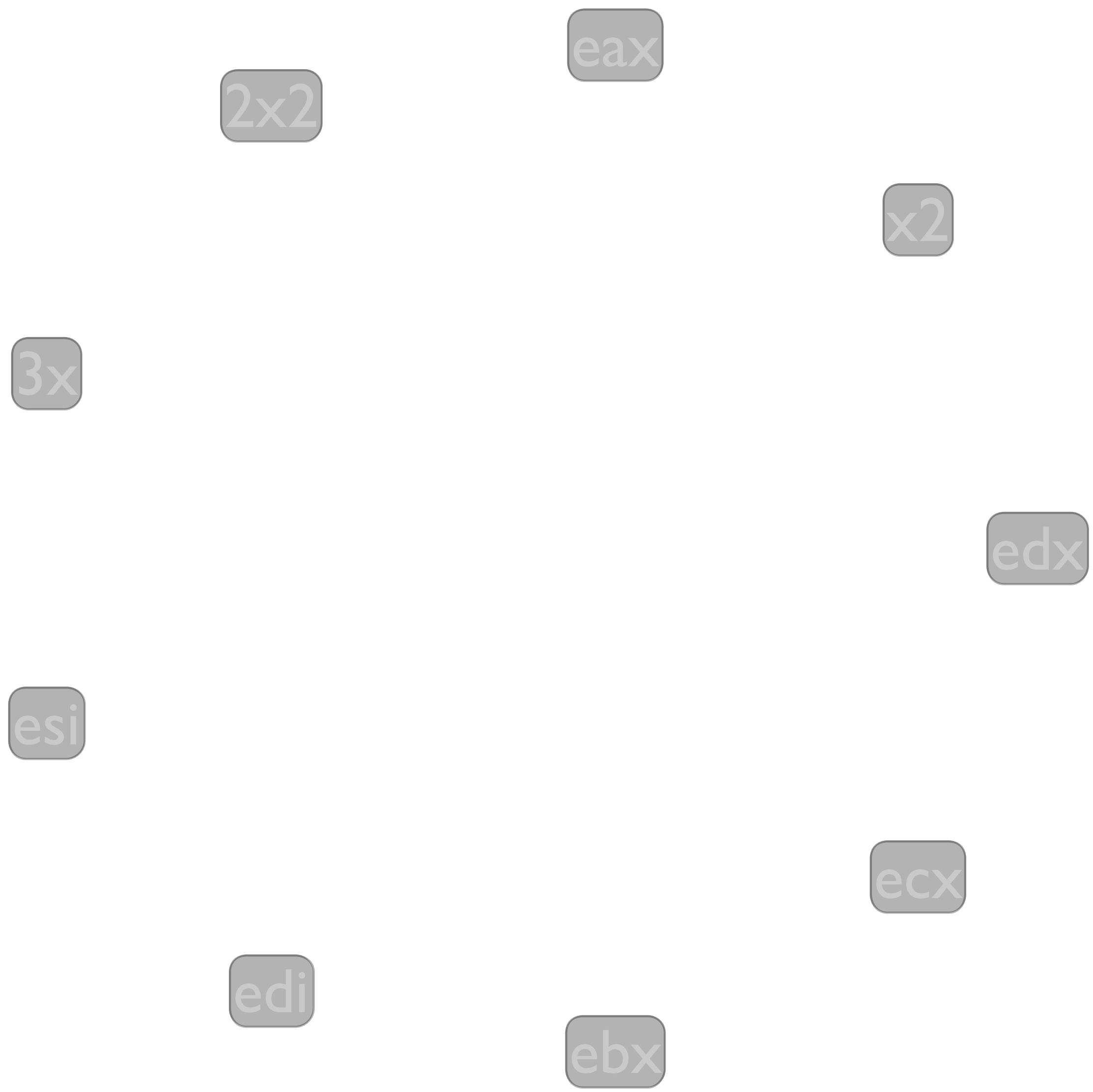

eax
2x2
x2
3x
edx
eax
esi
x2
2x2
ecx
3x
edi
ebx


eax
2x2
x2
3x
edx
edx
eax
x2
2x2
3x
esi
ecx
edi
ebx

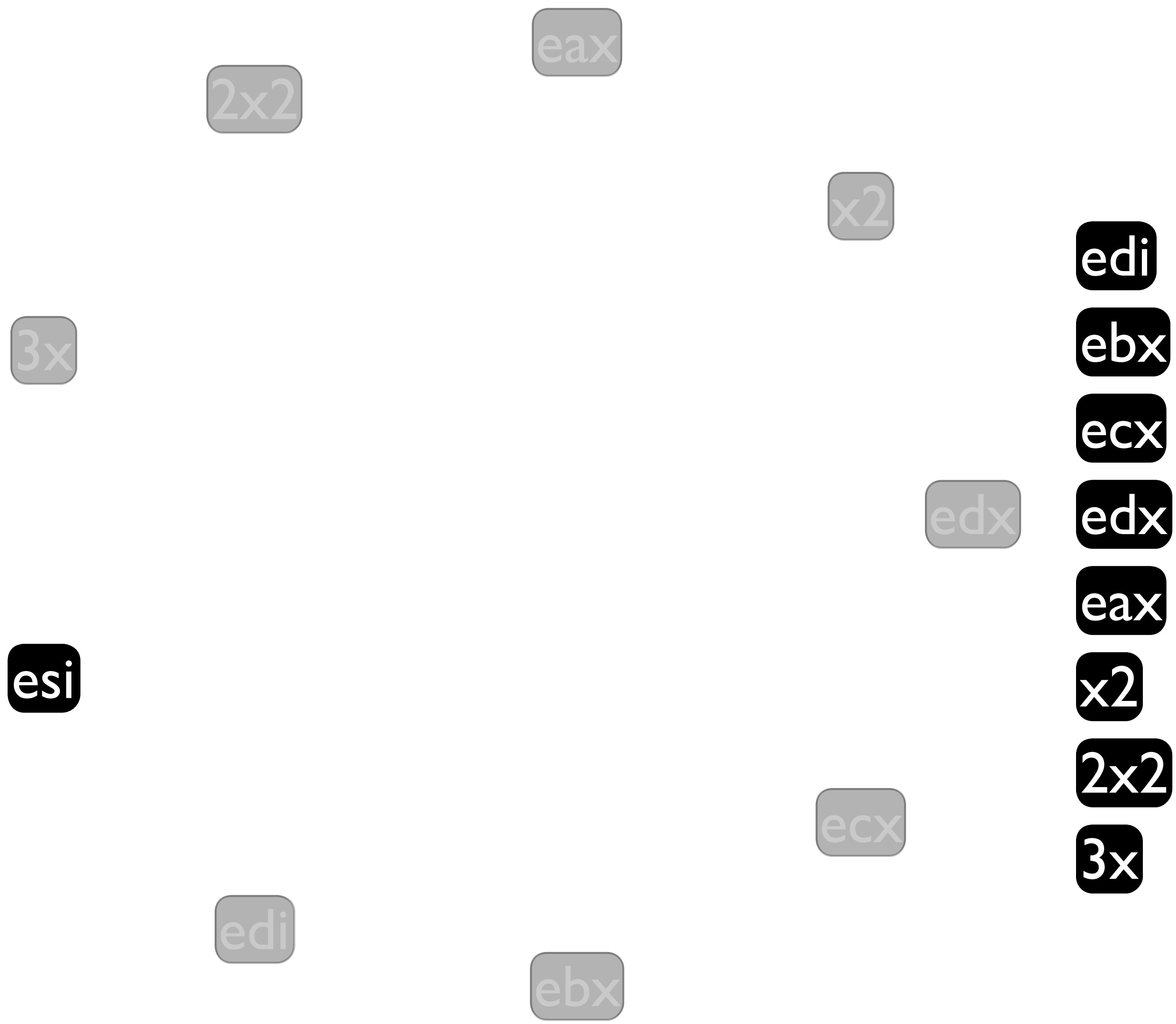

eax
2x2
x2
3x
ecx
edx
edx
eax
x2
2x2
esi
ecx
3x
edi
ebx

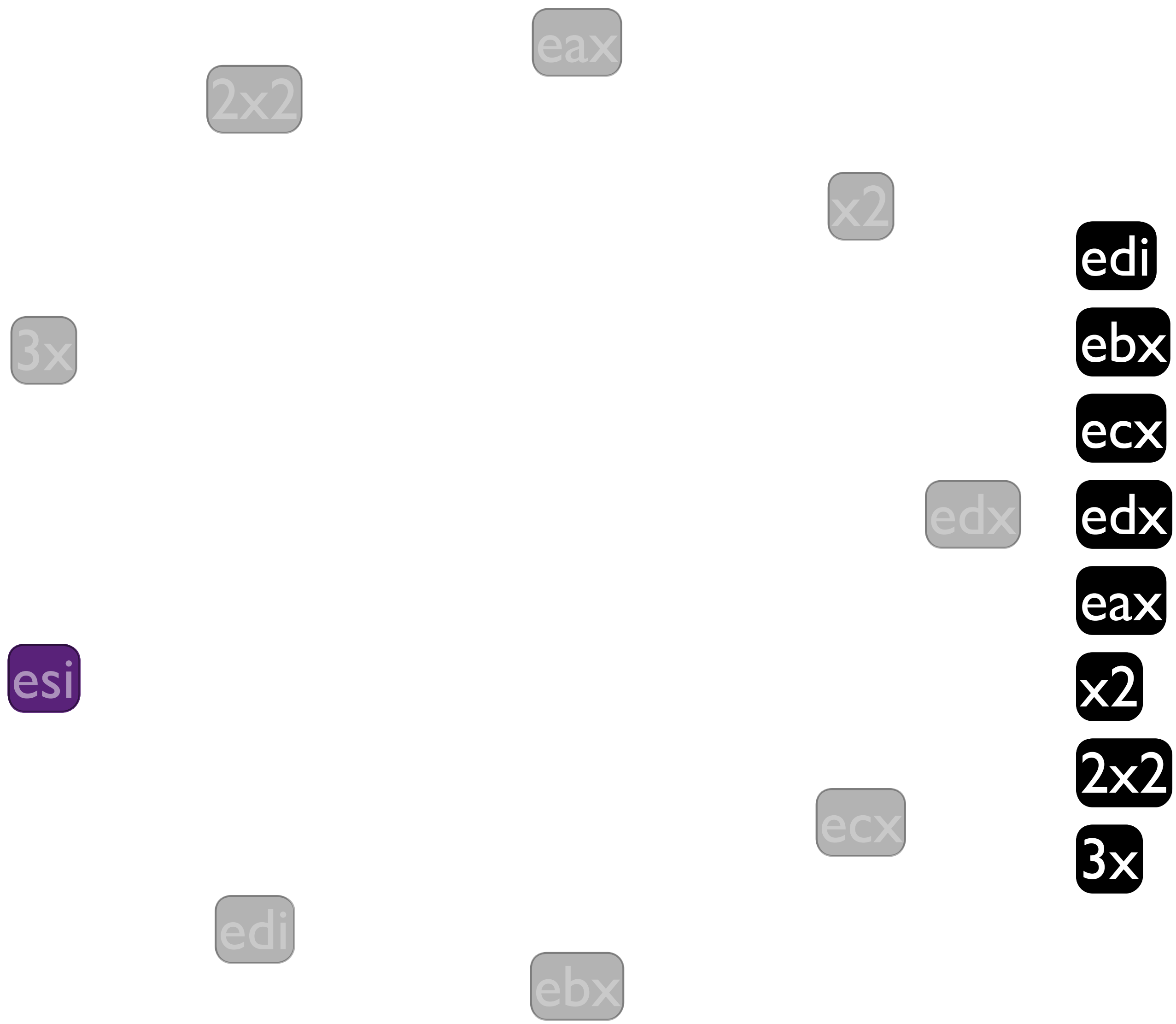

eax
2x2
x2
3x
ebx
ecx
edx
edx
eax
esi
x2
2x2
ecx
3x
edi
ebx

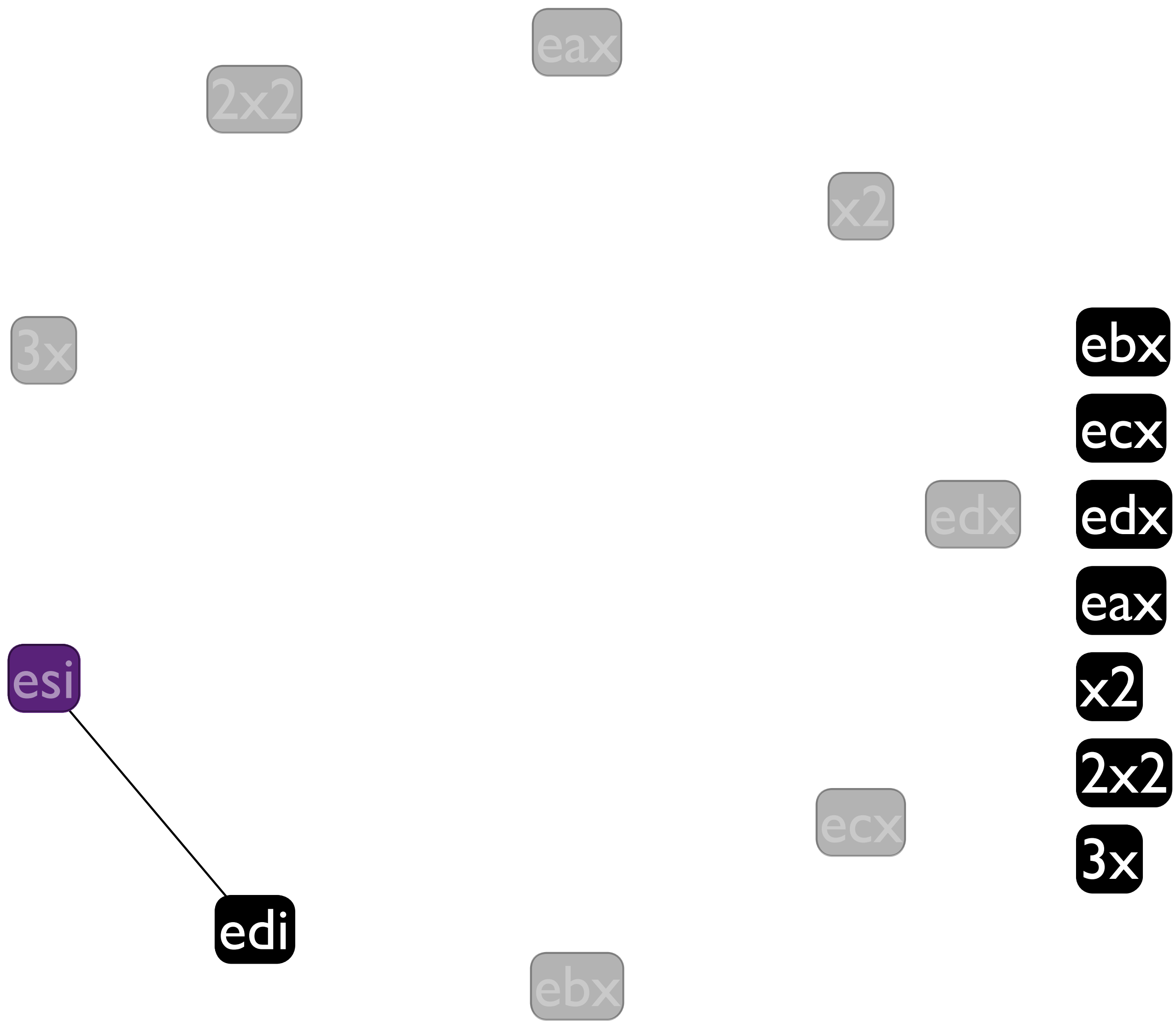

eax
2x2
x2
edi
ebx
ecx
3x
edx
edx
eax
esi
x2
2x2
ecx
3x
edi
ebx

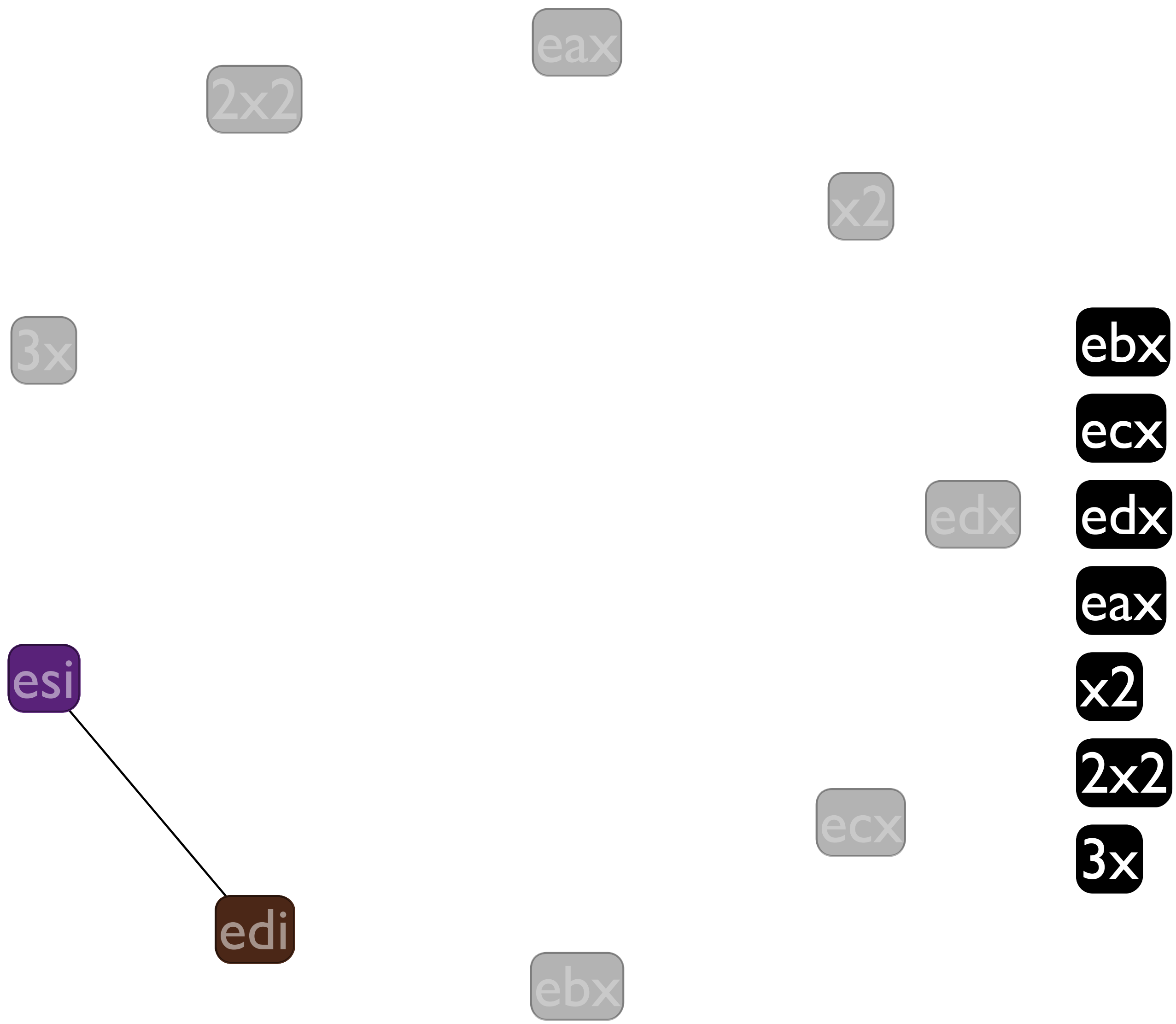


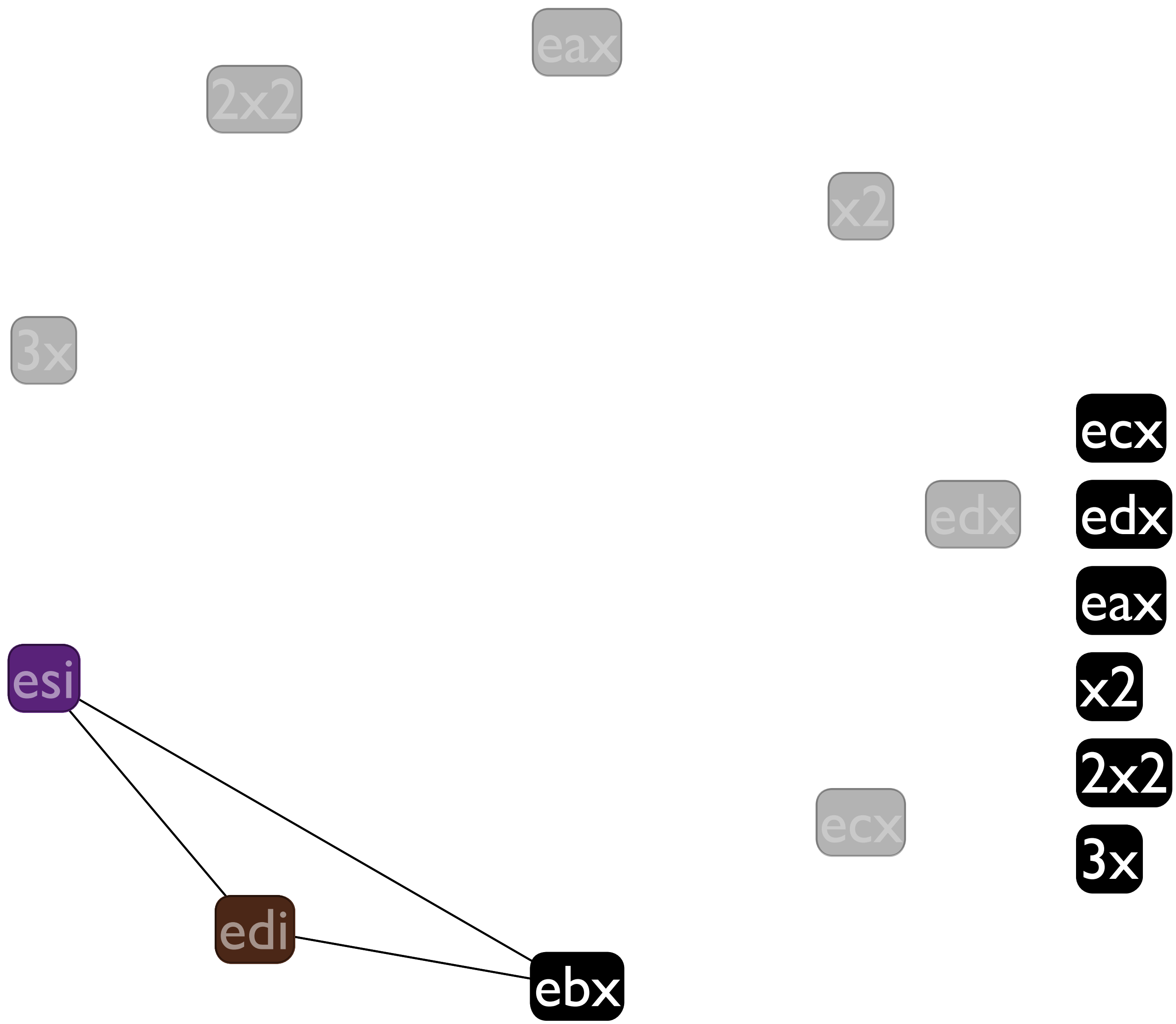
esi
edi
ebx
ecx
edx
eax
x2
2x2
3x

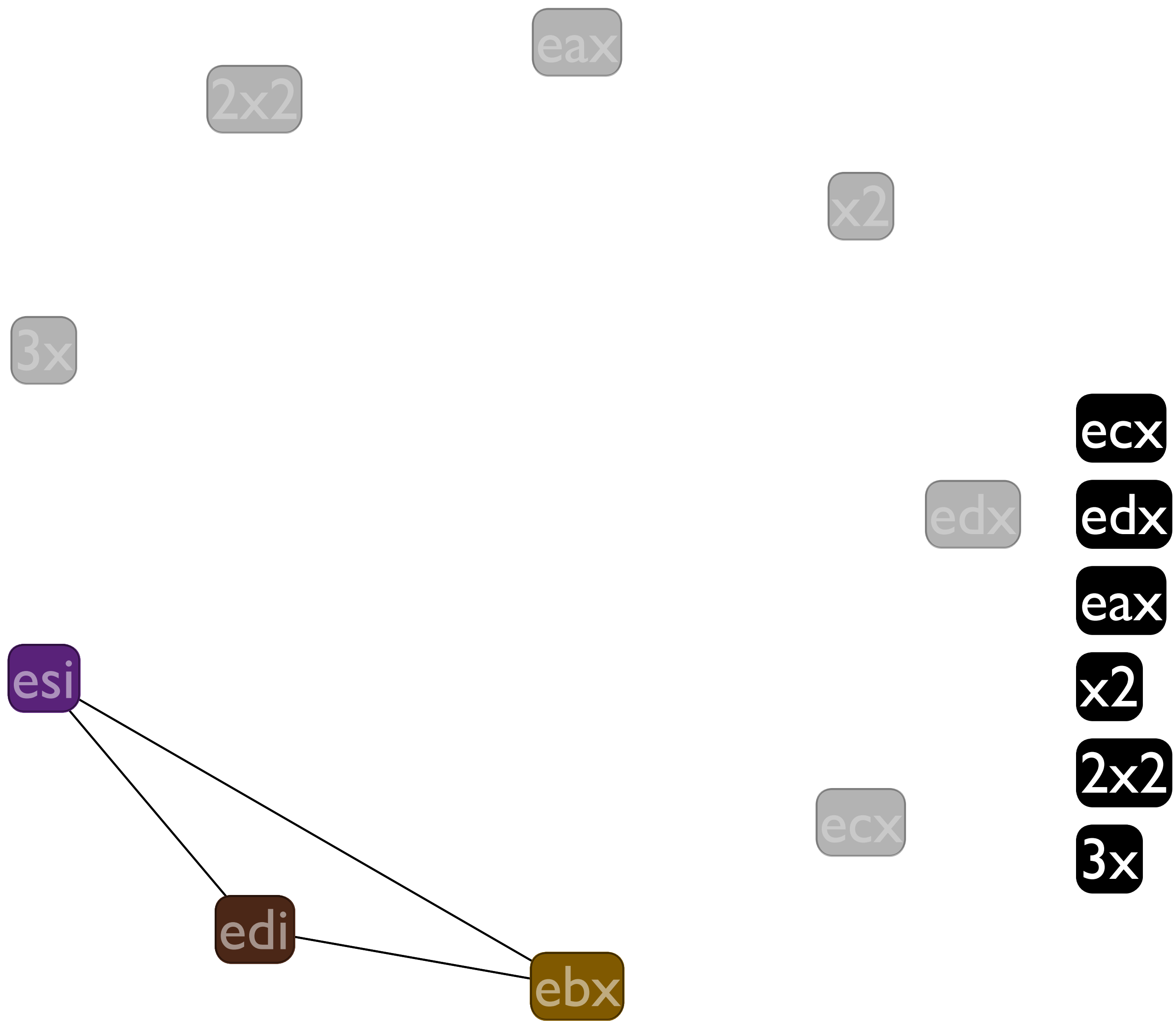

eax
2x2
x2
3x
edx
esi
ecx
edi
ebx
edi
ebx
ecx
edx
eax
x2
2x2
3x

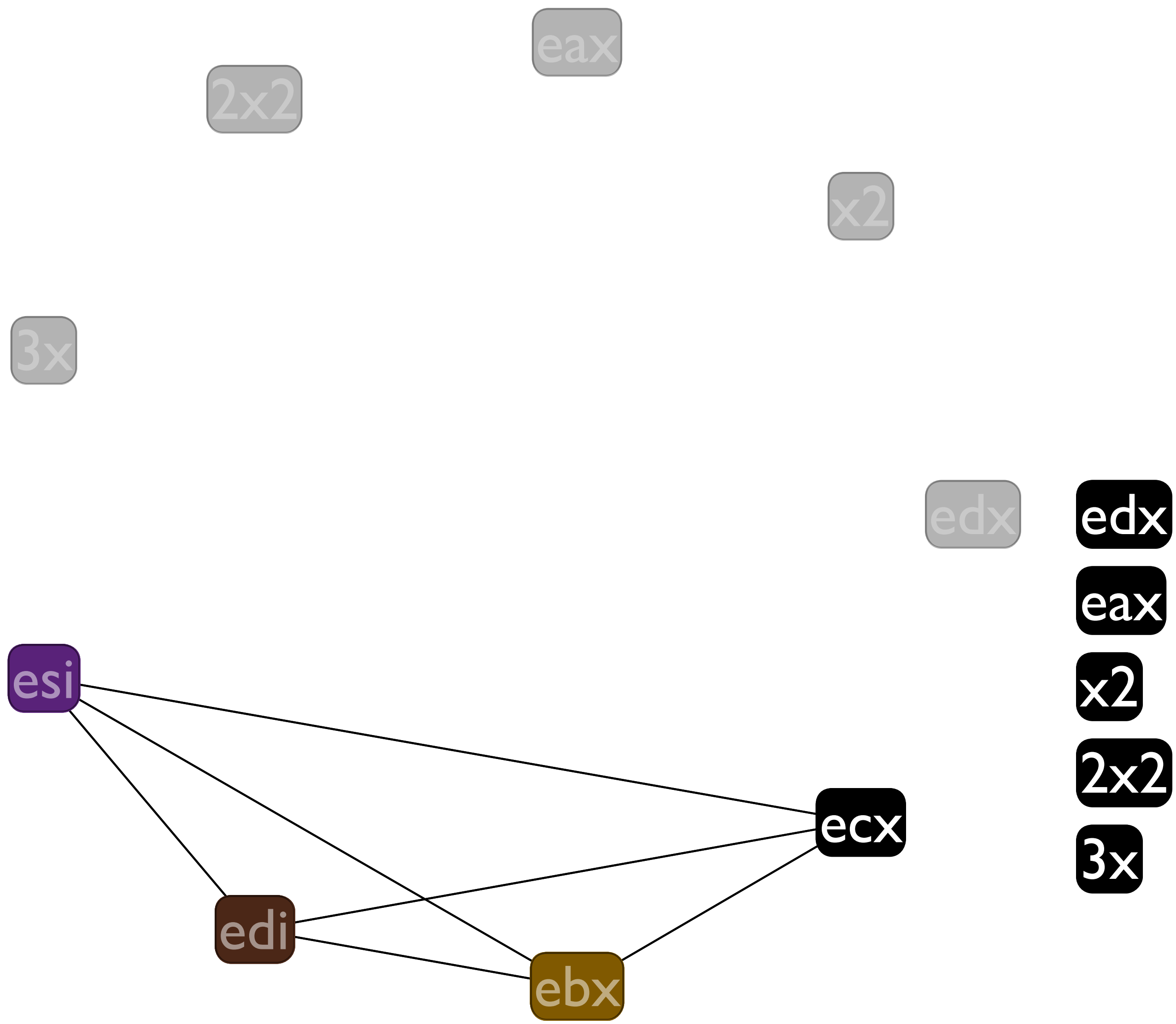

eax
2x2
x2
3x
edx
esi
ecx
edi
ebx
edi
ebx
ecx
edx
eax
x2
2x2
3x

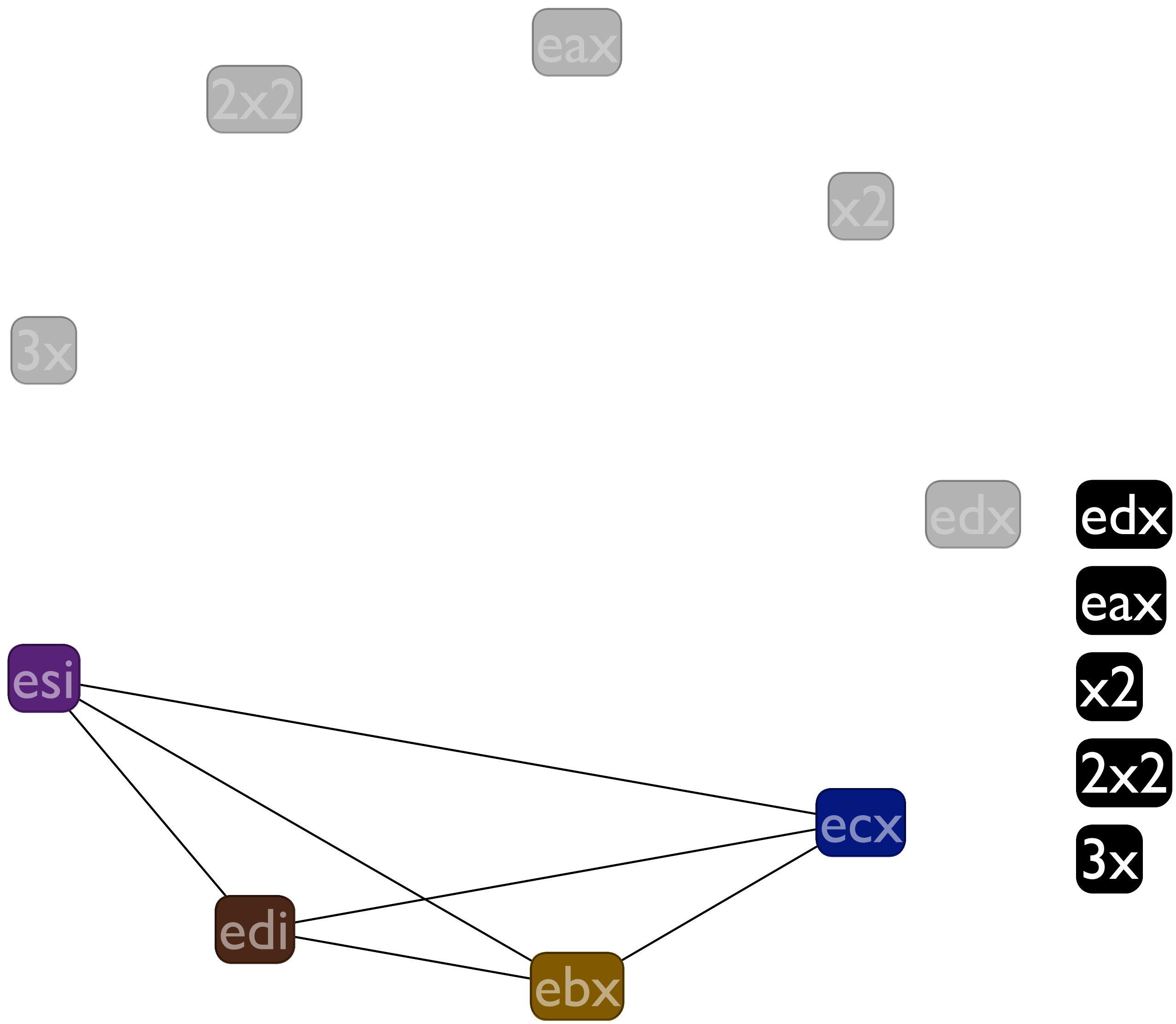

eax
2x2
x2
3x
ebx
ecx
edx
eax
x2
2x2
3x
edx
esi
ecx
edi
ebx

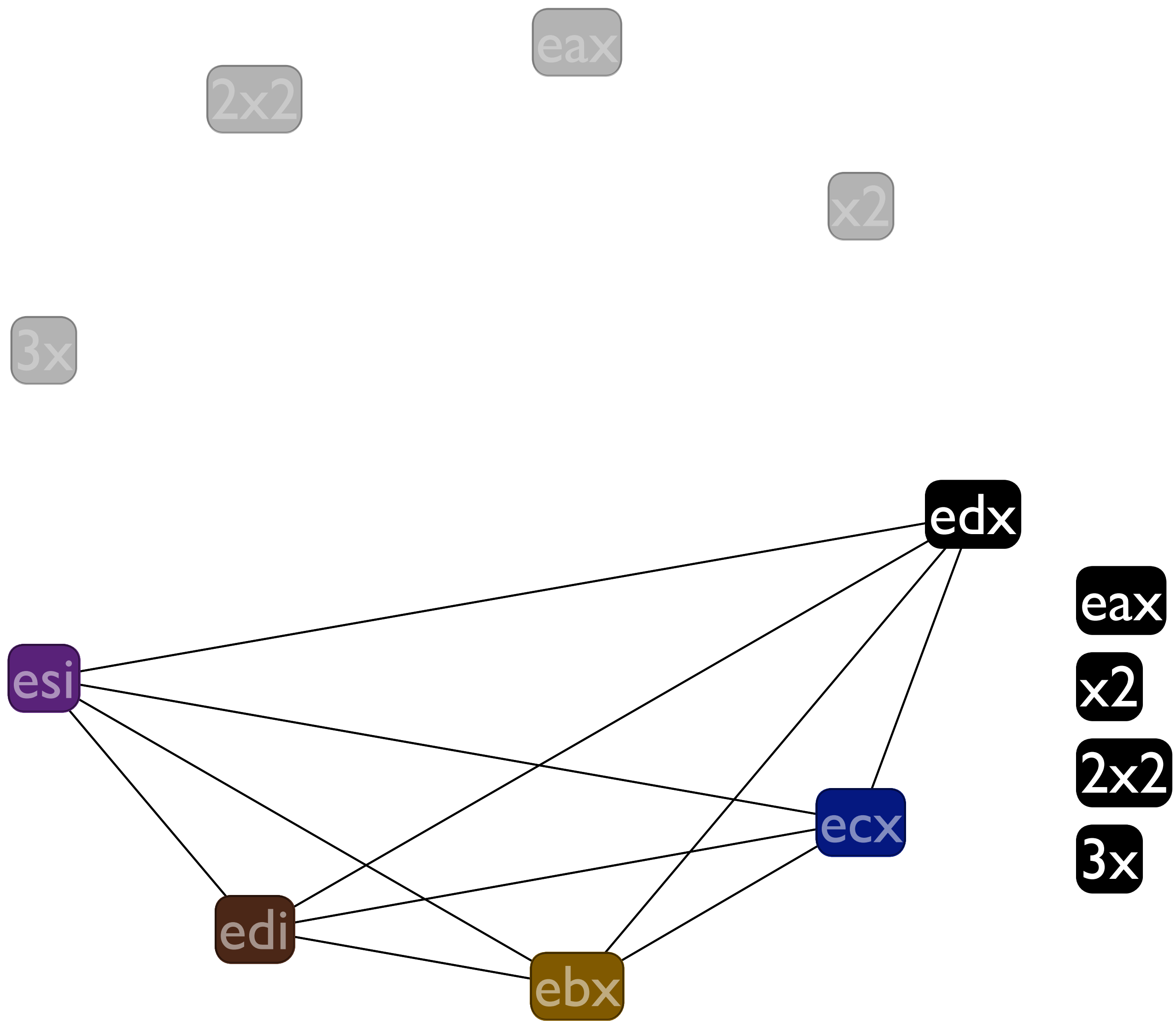

eax
2x2
x2
3x
ebx
ecx
edx
eax
x2
2x2
3x
edx
esi
ecx
edi
ebx

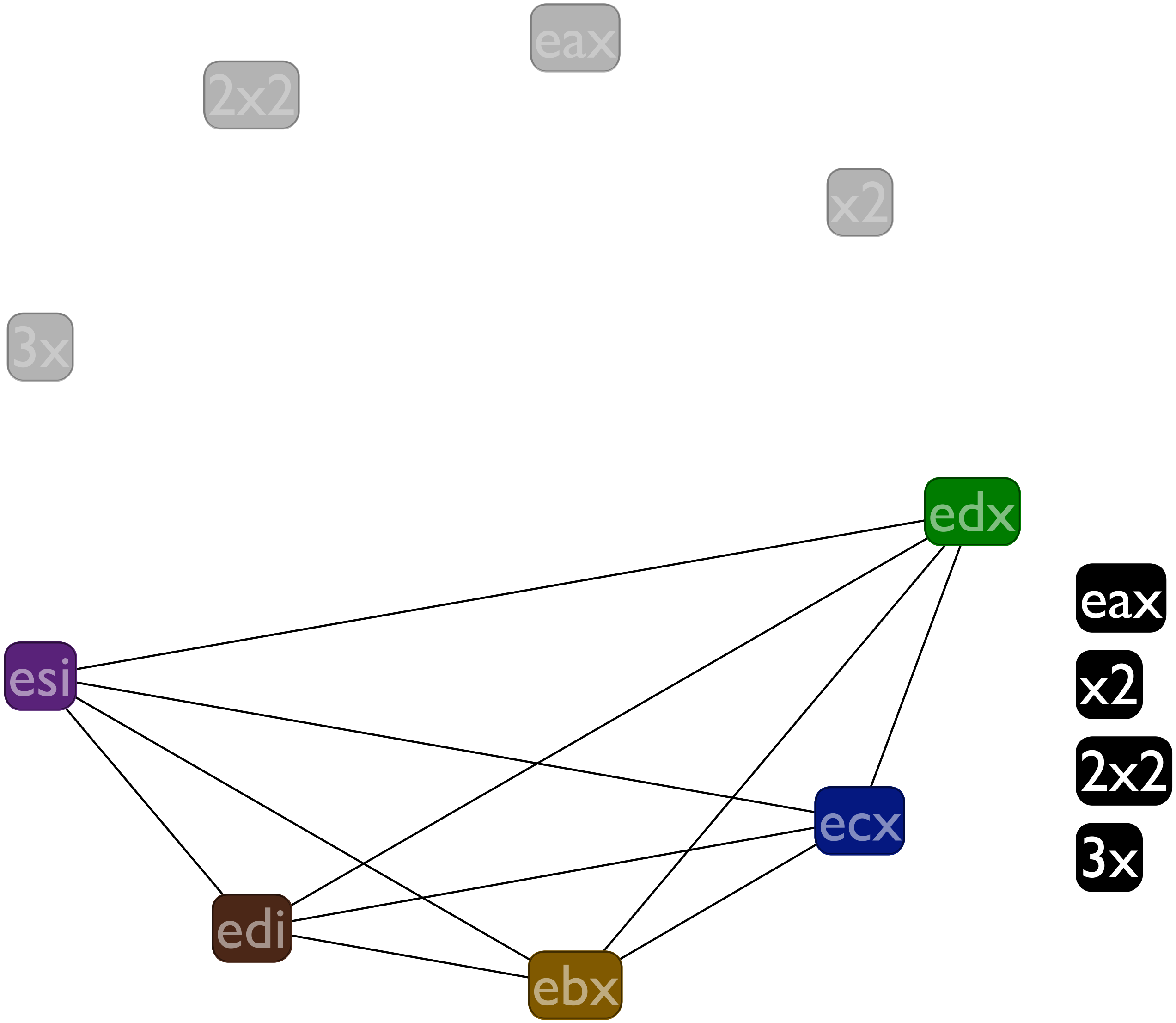

eax
2x2
x2
3x
ecx
edx
edx
eax
x2
2x2
esi
3x
ecx
edi
ebx

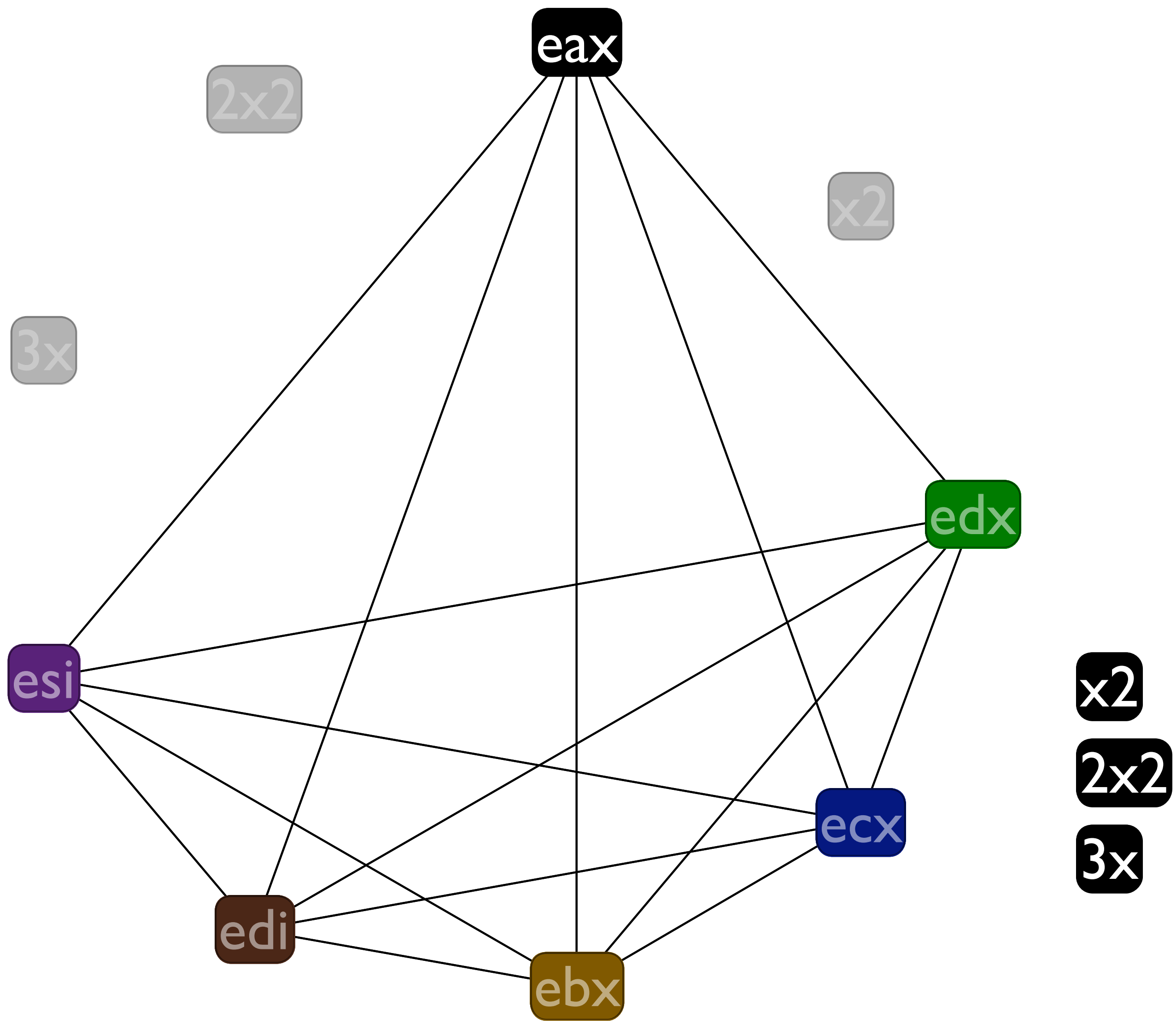

eax
2x2
x2
3x
edx
esi
ecx
edi
ebx
ecx
edx
eax
x2
2x2
3x

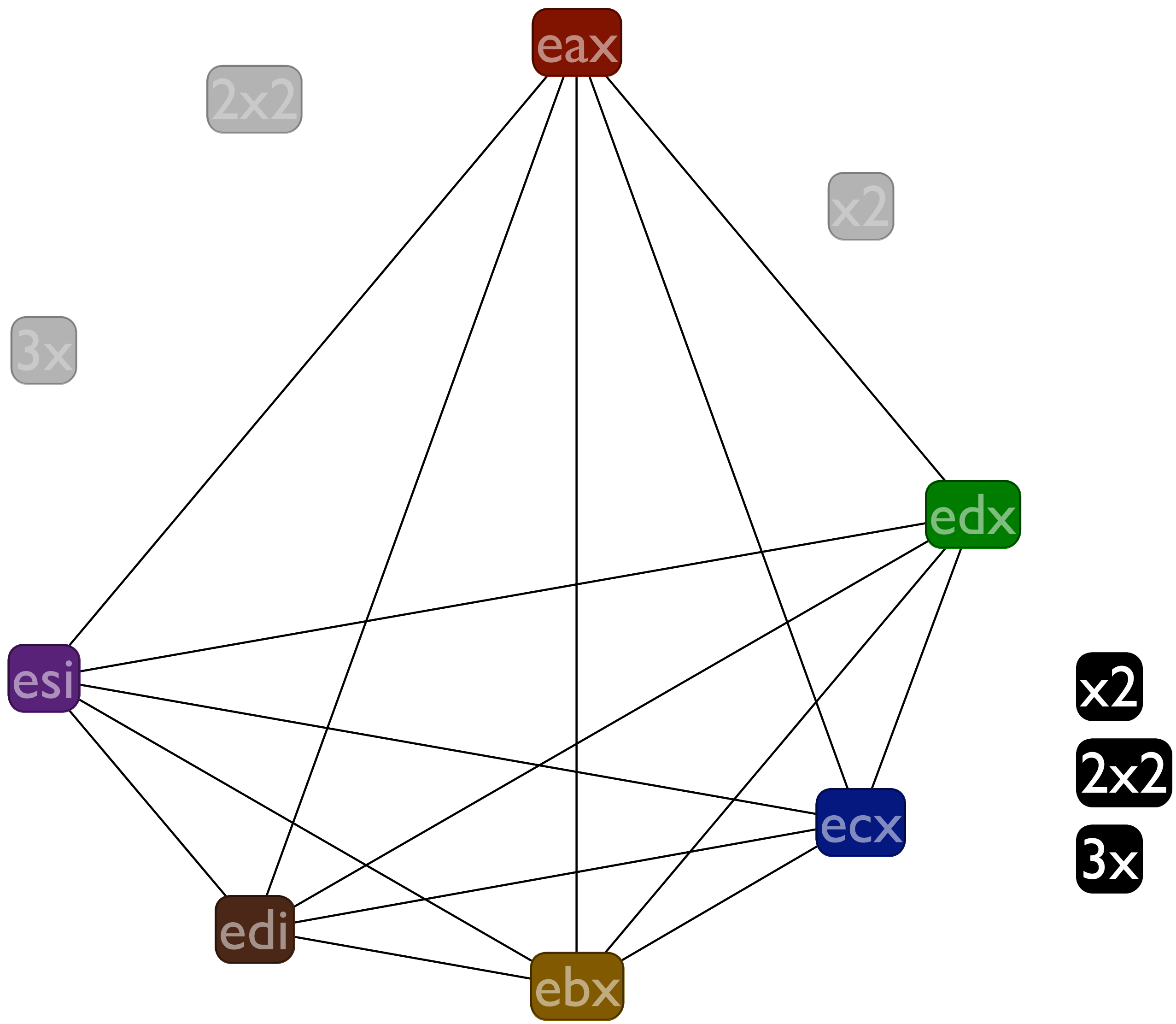

eax
2x2
x2
3x
edx
edx
eax
x2
2x2
3x
esi
ecx
edi
ebx

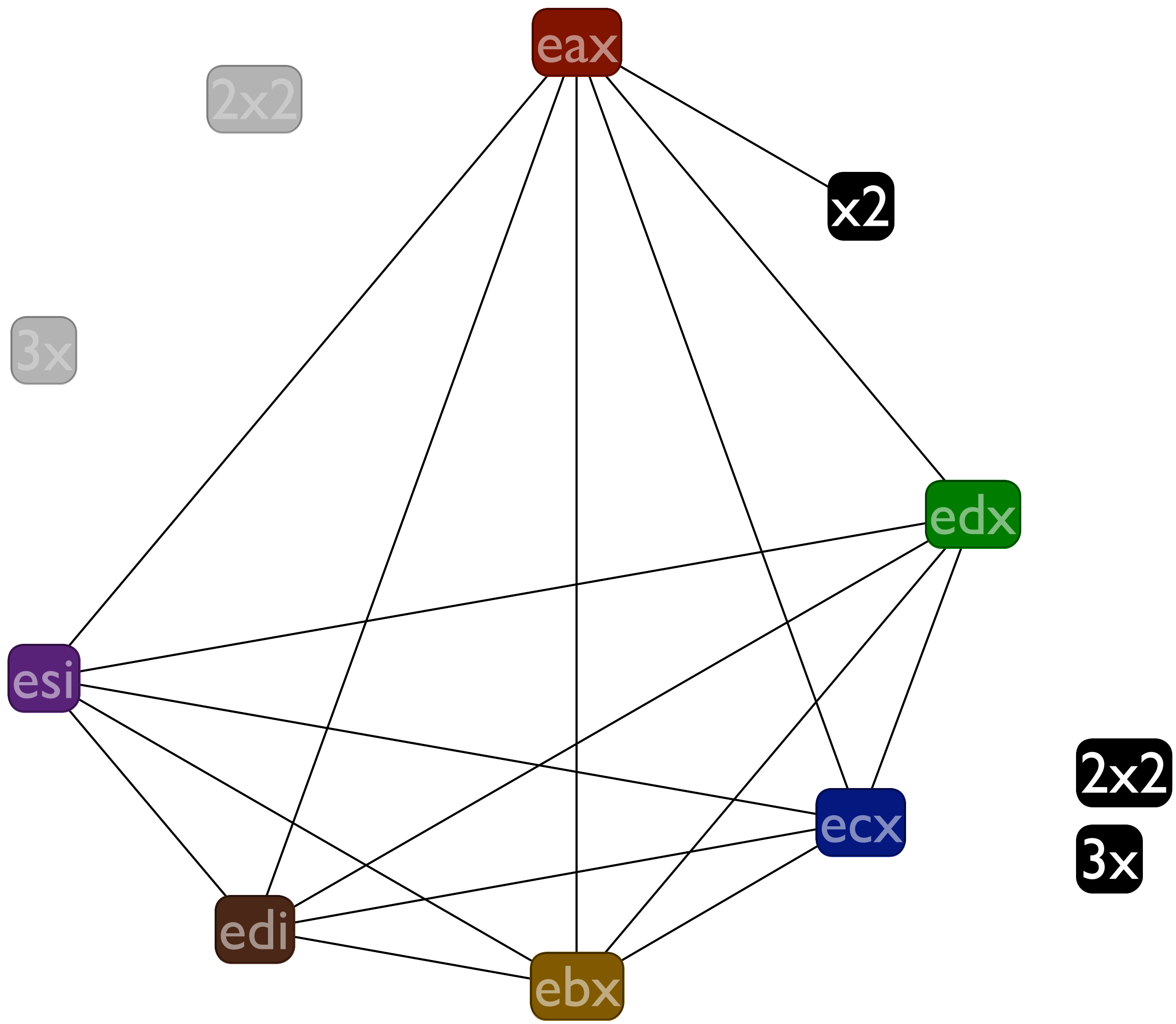

eax
2x2
x2
3x
edx
esi
ecx
edi
ebx
edx
eax
x2
2x2
3x

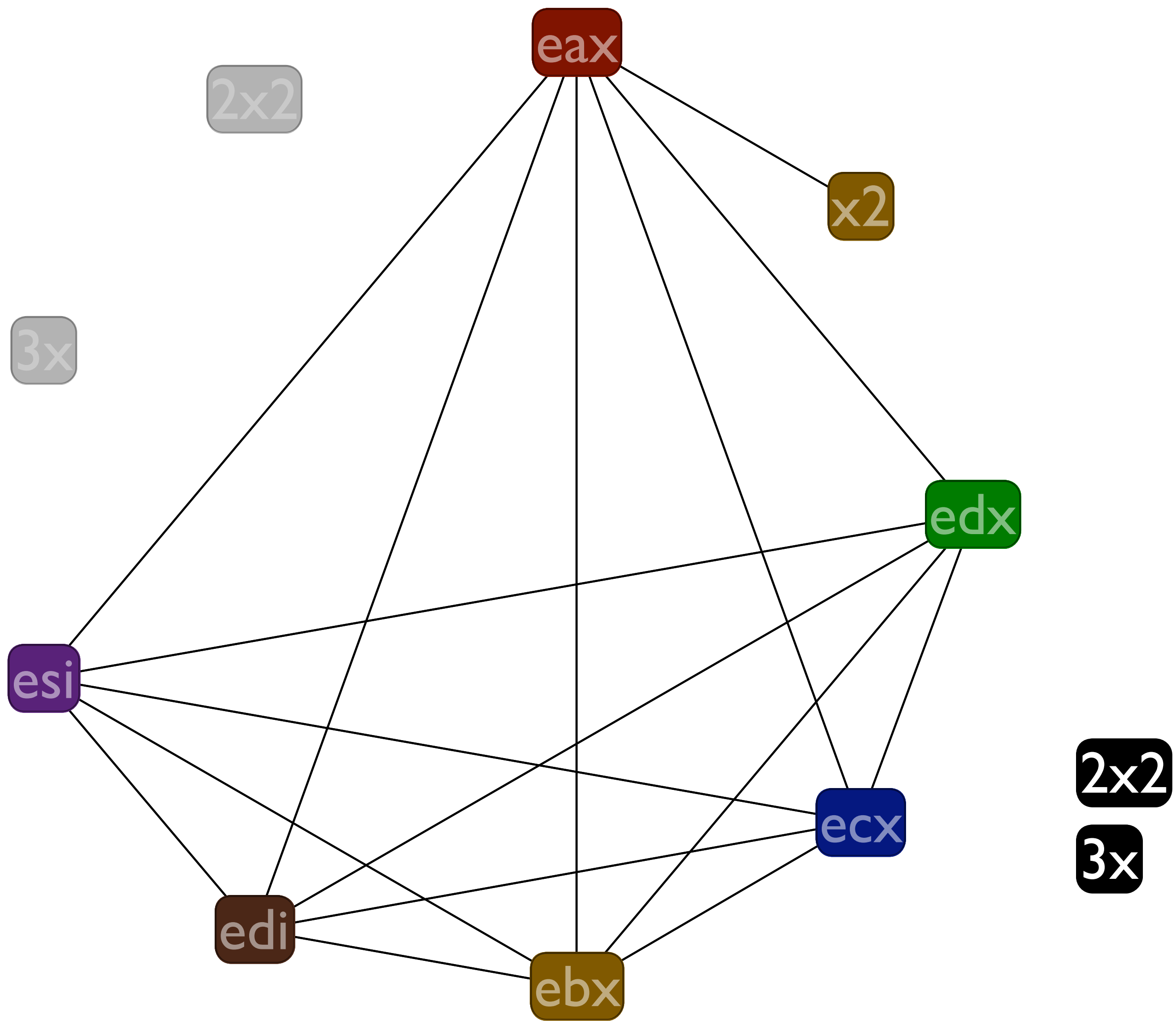

eax
2x2
x2
3x
edx
eax
esi
x2
2x2
ecx
3x
edi
ebx

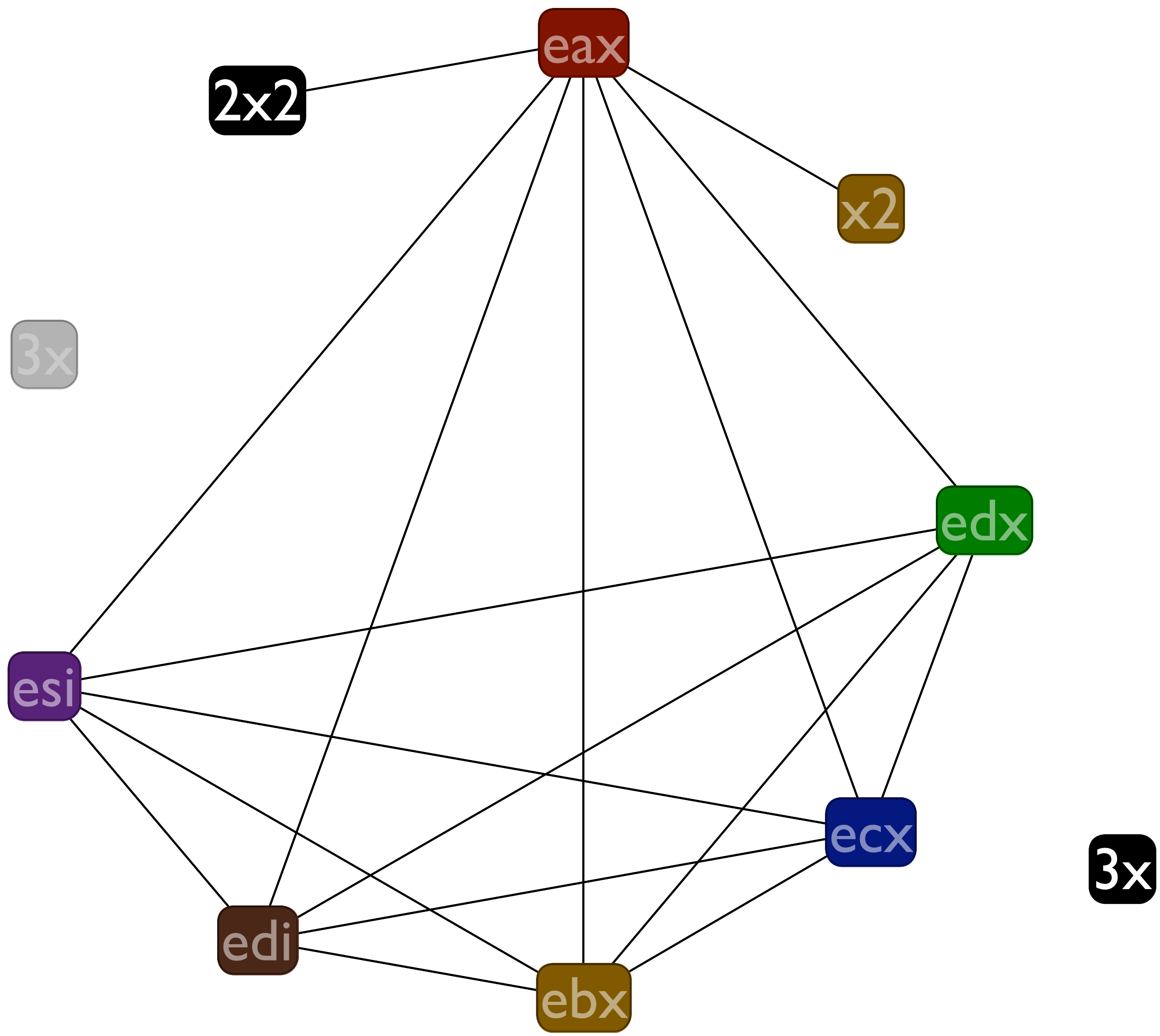

eax
2x2
x2
3x
edx
esi
ecx
edi
ebx
eax
x2
2x2
3x

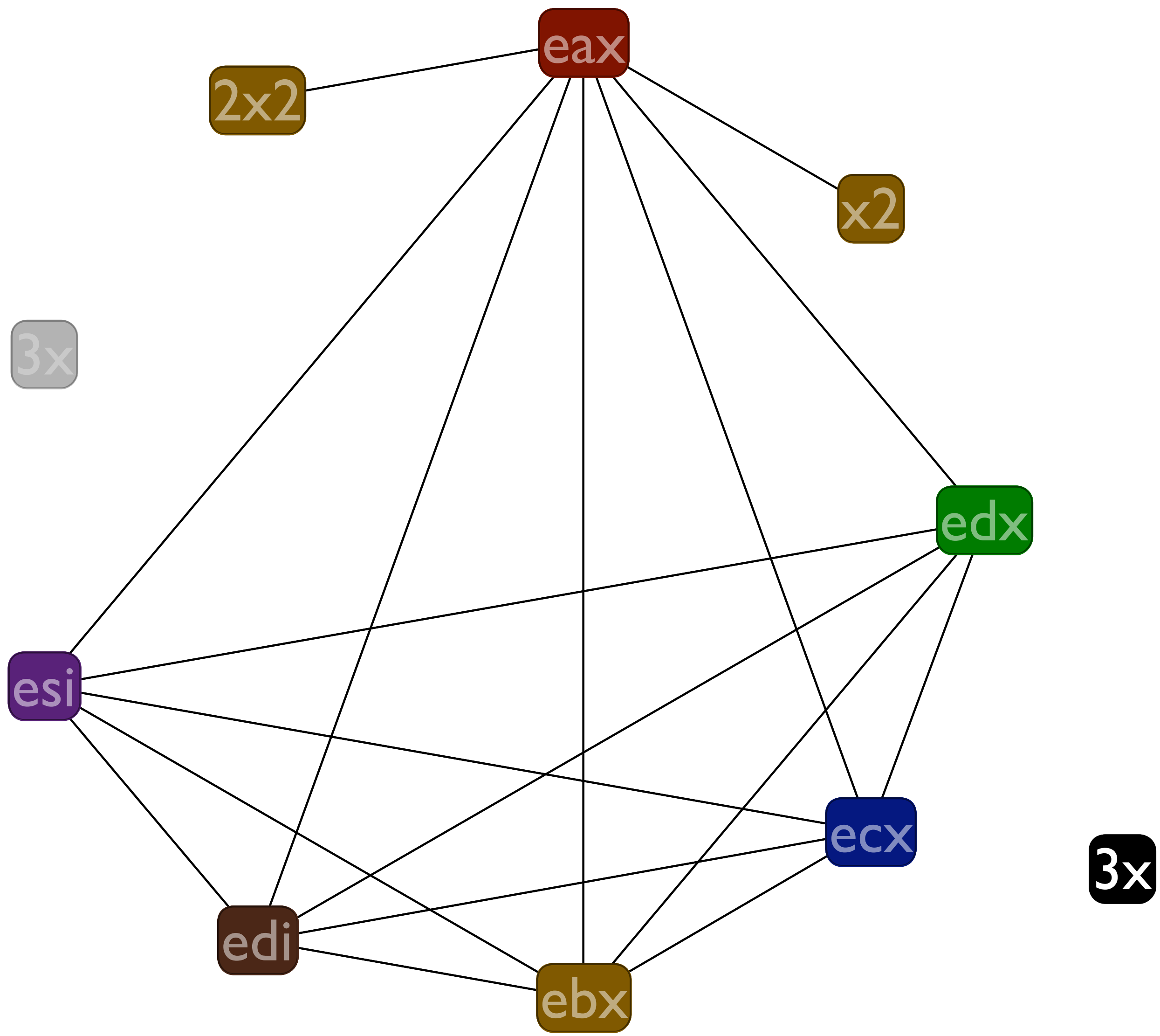

eax
2x2
x2
3x
edx
esi
ecx
edi
ebx
x2
2x2
3x

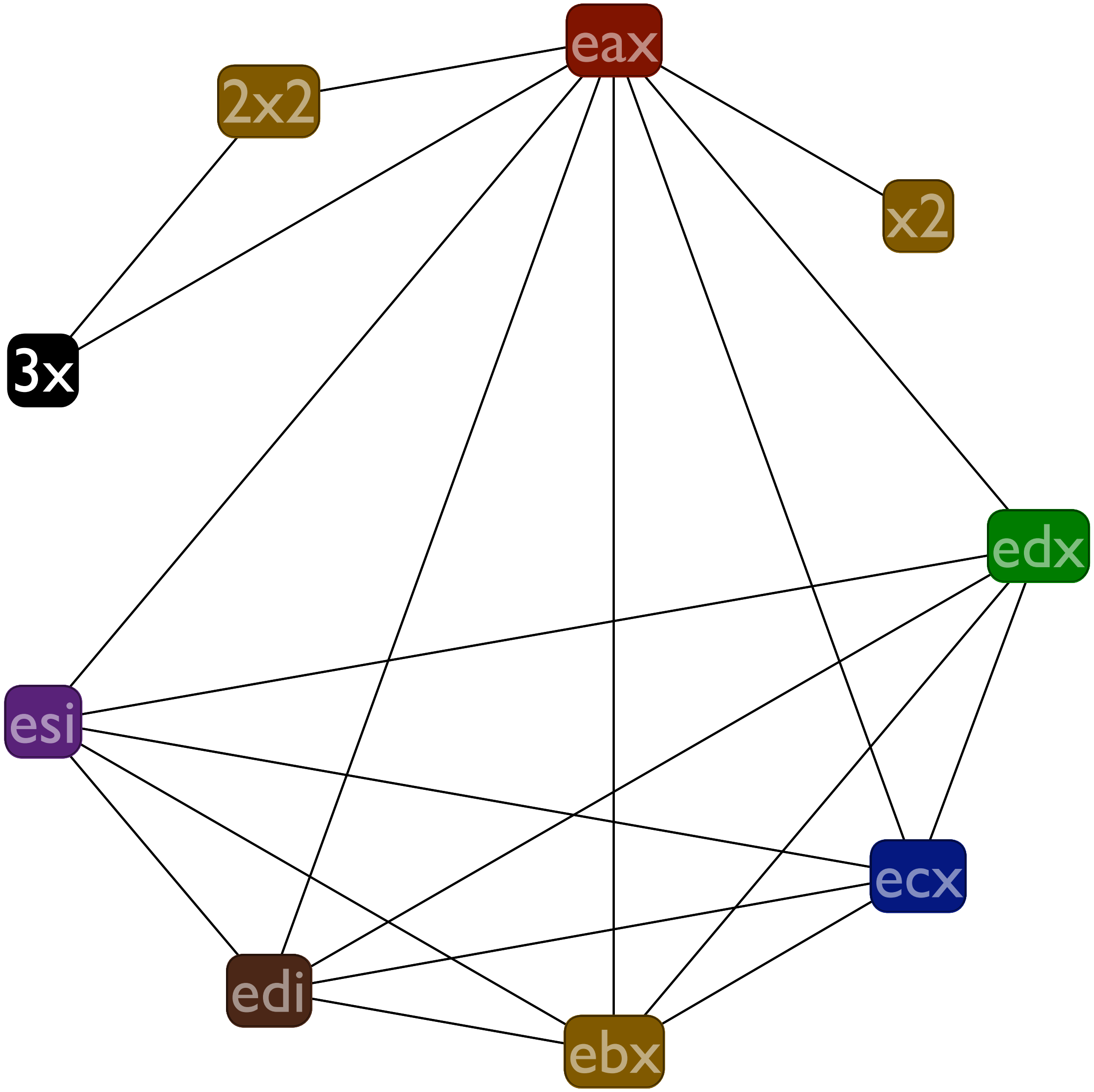

eax
2x2
x2
3x
edx
esi
x2
2x2
3x
ecx
edi
ebx


eax
2x2
x2
3x
edx
esi
2x2
ecx
3x
edi
ebx

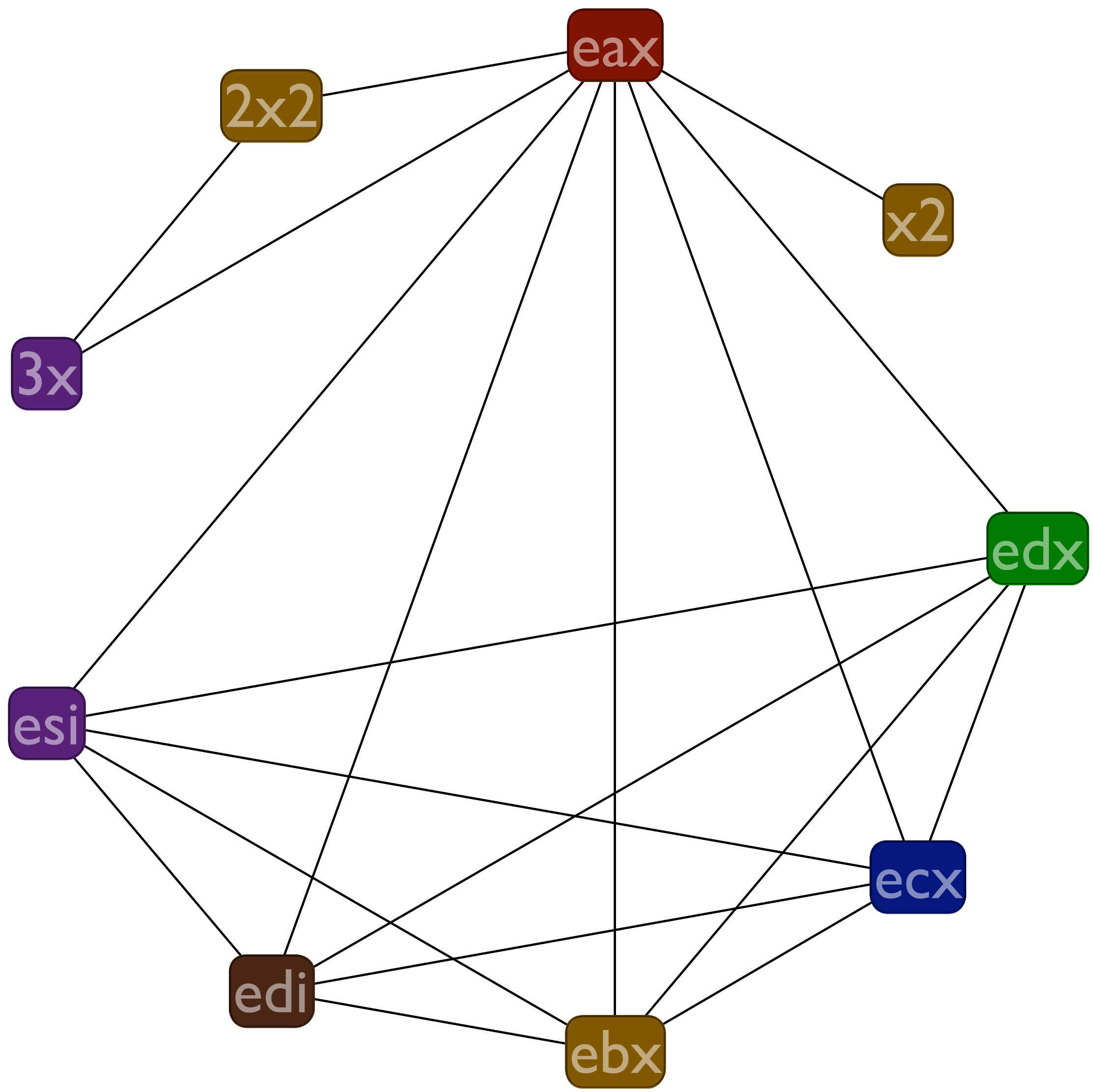

eax
2x2
x2
3x
edx
esi
2x2
ecx
3x
edi
ebx


eax
2x2
x2
3x
edx
esi
ecx
3x
edi
ebx


eax
2x2
x2
3x
edx
esi
ecx
3x
edi
ebx



eax
2x2
x2
3x
edx
esi
ecx
edi
ebx


eax
2x2
x2
3x
edx
esi
ecx
edi
ebx

Graph coloring algorithm

Some of the finer details of the graph coloring algorithm

- If possible, prefer to pull out nodes that have 5 or fewer edges at each stage
- When inserting & coloring nodes, always use the "smallest" color possible (according to any ordering on the colors you want). That is, imagine an ordering on the registers (say, alphabetical) and use the color of the first register in that order that works
- Ignore `ebp` and `esp` registers when building graph

Caller and callee save registers

Uhoh: the result of coloring the example graph is wrong, since we didn't account for the callee save registers! For example, **`3x`** clobbers **`esi`**, breaking the calling convention.

Gen/Kill overview

caller save ecx edx eax ebx
args eax edx ecx
callee save esi edi
result eax

	gen	kill
`(call s)`	`s` args	caller save result
`(tail-call s)`	`s` args callee save	
`(return)`	result callee save	

Call may kill caller save regs

	gen	kill
`(call s)`	`s` args	caller save result
`(tail-call s)`	`s` args callee save	
`(return)`	result callee save	

```
(a <- 2)
(eax <- 1)
(call :f)
(a *= a)
```

Since the caller is responsible for saving the caller save registers, e.g., **`ecx`** then **`:f`** is free to clobber it. Thus we must be sure not to use **`ecx`** for **`a`**.

Return gens result

	gen	kill
`(call s)`	`s` args	caller save result
`(tail-call s)`	`s` args callee save	
`(return)`	result callee save	

```
(eax <- 1)
(a <- edx)
(a *= a)
((mem b 4) <- a)
(return)
```

Must make sure that **`a`** is not allocated into the result register **`eax`**. With a reference (aka gen) at the **`return`** then the live range spans the assignment to **`a`** and thus **`a`** won't go into **`eax`**

Ensure callee saves are saved

	gen	kill
`(call s)`	`s` args	caller save result
`(tail-call s)`	`s` args callee save	
`(return)`	result callee save	

```
(eax <- 1)
(a <- edx)
(a *= a)
((mem b 4) <- a)
(return)
```

Enforces the calling convention; specifically that the callee save registers are actually saved. In the code on the left, `a` must not be in `esi`

Call kills return

	gen	kill
`(call s)`	`s` args	caller save result
`(tail-call s)`	`s` args callee save	
`(return)`	result callee save	

```
(a <- 1)
(call :f)
(b <- a)
```

If we put `a` into `eax` then the call to `:f` would clobber it to save the return value, so we must avoid that by treating the `call` as a kill to the return register, `eax`
(Yep, this is the second reason to do this; other calling conventions may not have both reasons tho)

Call/tail-call gen arguments

	gen	kill
`(call s)`	`s` args	caller save result
`(tail-call s)`	`s` args callee save	
`(return)`	result callee save	

```
(ecx <- 1)
(a <- 2)
(call :f)
```

If we put `a` into `ecx` then the call to `:f` would get the wrong arguments. Avoid this by making `call` and `tail-call` refer to (i.e., gen) the argument registers. (Probably your compiler won't generate code like this, but spilling could put you into this kind of situation.)

Constrained arithmetic operators

The **(cx <- s cmp s)** instruction in L1 is limited to only 4 possible destinations.

The **(x sop= sx)** instruction in L1 is limited to only shifting by the value of **ecx** (or by a constant in the other form).

Constrained arithmetic operators

Add interference edges to disallow the illegal registers when building the interference graph, before starting the coloring.

E.g., if you have this instruction `(a <- y < x)` then add edges between `a` and the registers `edi` and `esi`, ensuring `a` ends up in `eax`, `ecx`, `edx`, `ebx`, or spilled.

Do over

Lets redo the coloring, now with the callee and caller save register information in the graph

Gen & Kill

		gen	kill
1:	`:f`	()	()
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2)	(2x2)
6:	`(3x <- eax)`	(eax)	(3x)
7:	`(3x *= 3)`	(3x)	(3x)
8:	`(eax <- 2x2)`	(2x2)	(eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	()	()
2:	`(x2 <- eax)`	()	()
3:	`(x2 *= x2)`	()	()
4:	`(2x2 <- x2)`	()	()
5:	`(2x2 *= 2)`	()	()
6:	`(3x <- eax)`	()	()
7:	`(3x *= 3)`	()	()
8:	`(eax <- 2x2)`	()	()
9:	`(eax += 3x)`	()	()
10:	`(eax += 4)`	()	()
11:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	()	()
2:	`(x2 <- eax)`	(eax)	()
3:	`(x2 *= x2)`	(x2)	()
4:	`(2x2 <- x2)`	(x2)	()
5:	`(2x2 *= 2)`	(2x2)	()
6:	`(3x <- eax)`	(eax)	()
7:	`(3x *= 3)`	(3x)	()
8:	`(eax <- 2x2)`	(2x2)	()
9:	`(eax += 3x)`	(3x eax)	()
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	()	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2)	(eax)
6:	`(3x <- eax)`	(eax)	(3x)
7:	`(3x *= 3)`	(3x)	(2x2)
8:	`(eax <- 2x2)`	(2x2)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2 eax)	(eax)
6:	`(3x <- eax)`	(eax)	(3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(eax)
6:	`(3x <- eax)`	(eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	**out**
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax)
6:	`(3x <- eax)`	(2x2 eax)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax)
5:	`(2x2 *= 2)`	(2x2 eax edi esi)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax x2)	(2x2 eax edi esi)
5:	`(2x2 *= 2)`	(2x2 eax edi esi)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax x2)
4:	`(2x2 <- x2)`	(eax edi esi x2)	(2x2 eax edi esi)
5:	`(2x2 *= 2)`	(2x2 eax edi esi)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax x2)	(eax edi esi x2)
4:	`(2x2 <- x2)`	(eax edi esi x2)	(2x2 eax edi esi)
5:	`(2x2 *= 2)`	(2x2 eax edi esi)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax x2)
3:	`(x2 *= x2)`	(eax edi esi x2)	(eax edi esi x2)
4:	`(2x2 <- x2)`	(eax edi esi x2)	(2x2 eax edi esi)
5:	`(2x2 *= 2)`	(2x2 eax edi esi)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax)	(eax edi esi x2)
3:	`(x2 *= x2)`	(eax edi esi x2)	(eax edi esi x2)
4:	`(2x2 <- x2)`	(eax edi esi x2)	(2x2 eax edi esi)
5:	`(2x2 *= 2)`	(2x2 eax edi esi)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

Liveness

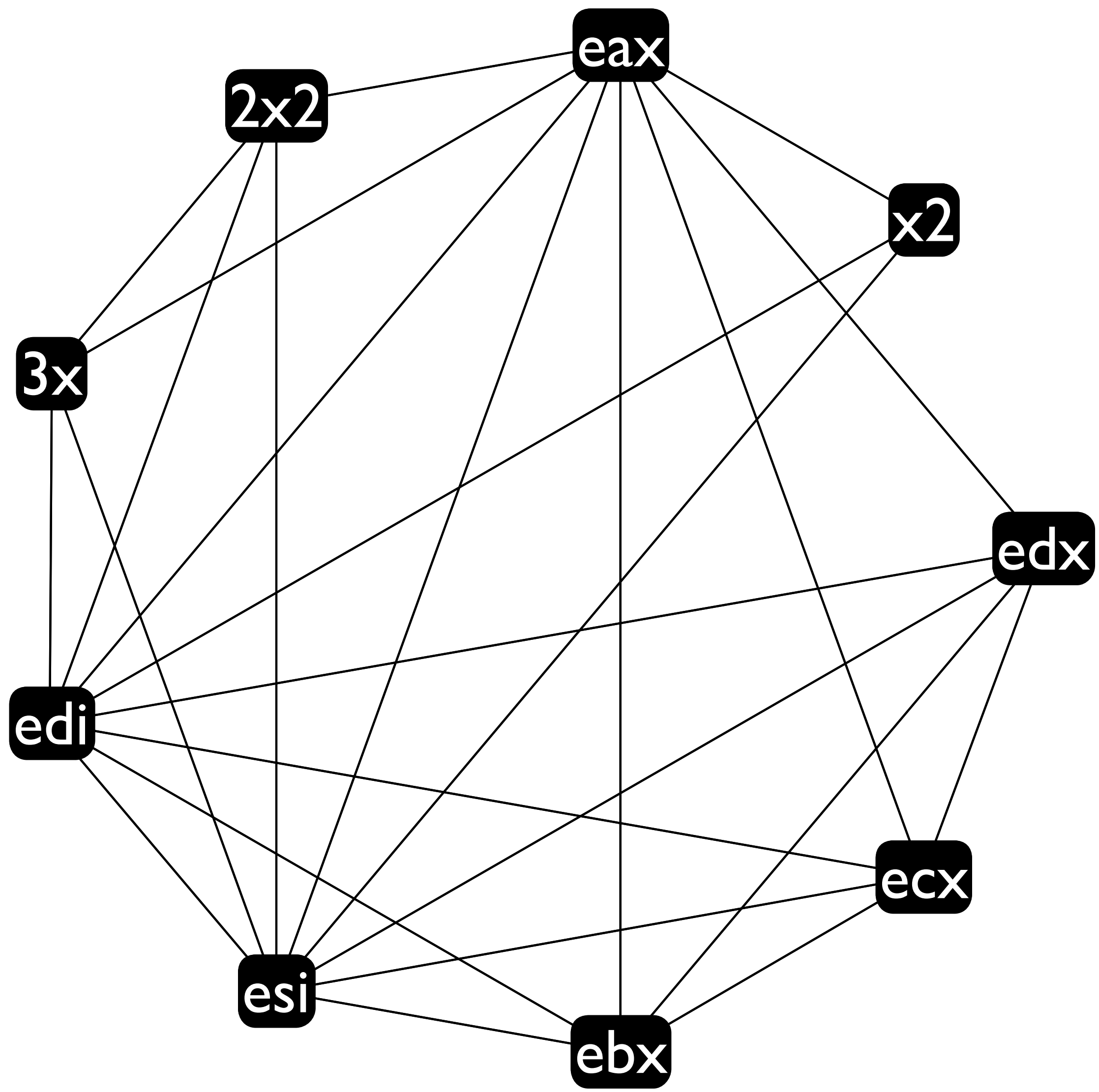
		in	out
1:	`:f`	(eax)	(eax)
2:	`(x2 <- eax)`	(eax edi esi)	(eax edi esi x2)
3:	`(x2 *= x2)`	(eax edi esi x2)	(eax edi esi x2)
4:	`(2x2 <- x2)`	(eax edi esi x2)	(2x2 eax edi esi)
5:	`(2x2 *= 2)`	(2x2 eax edi esi)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

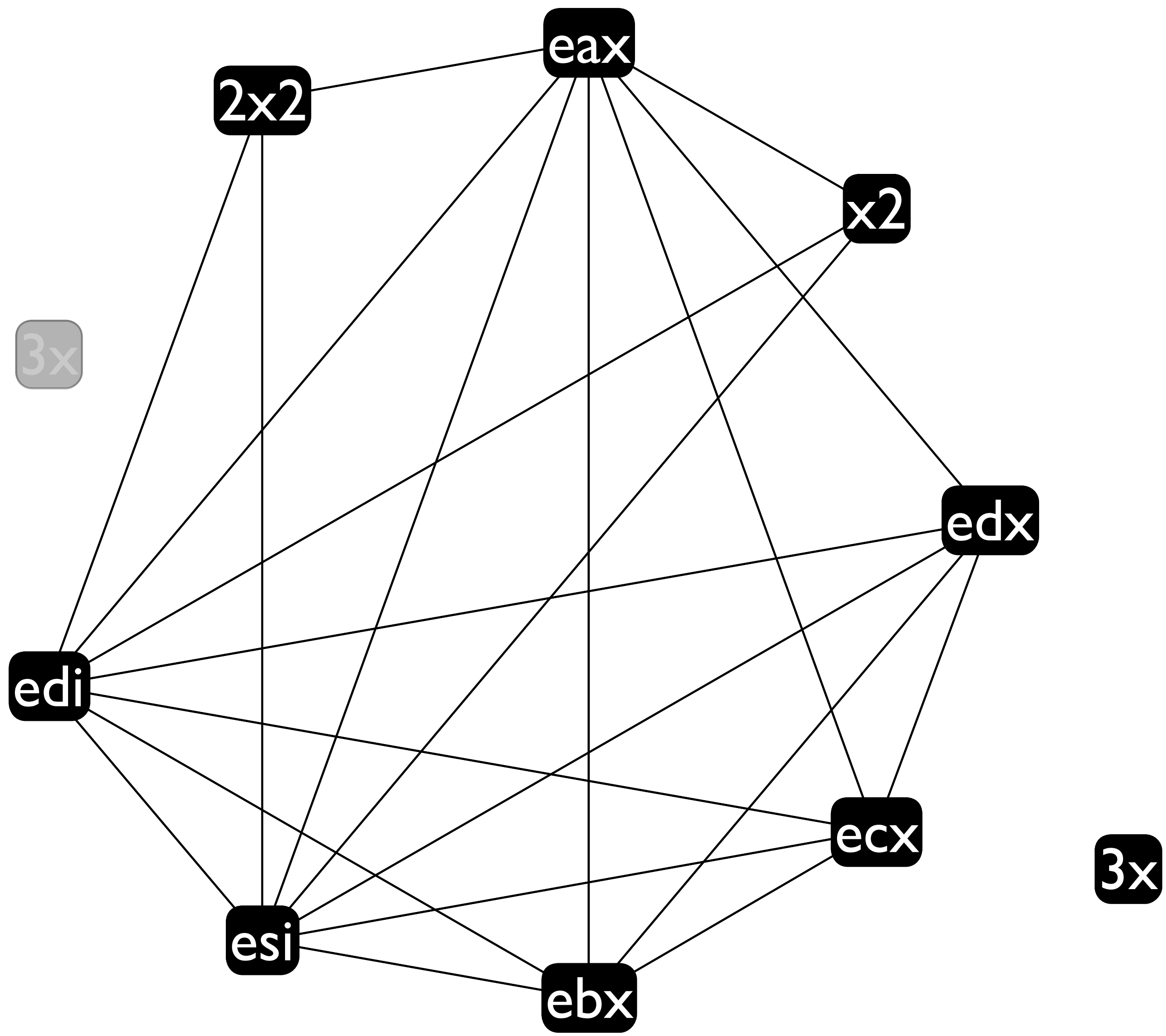
Liveness

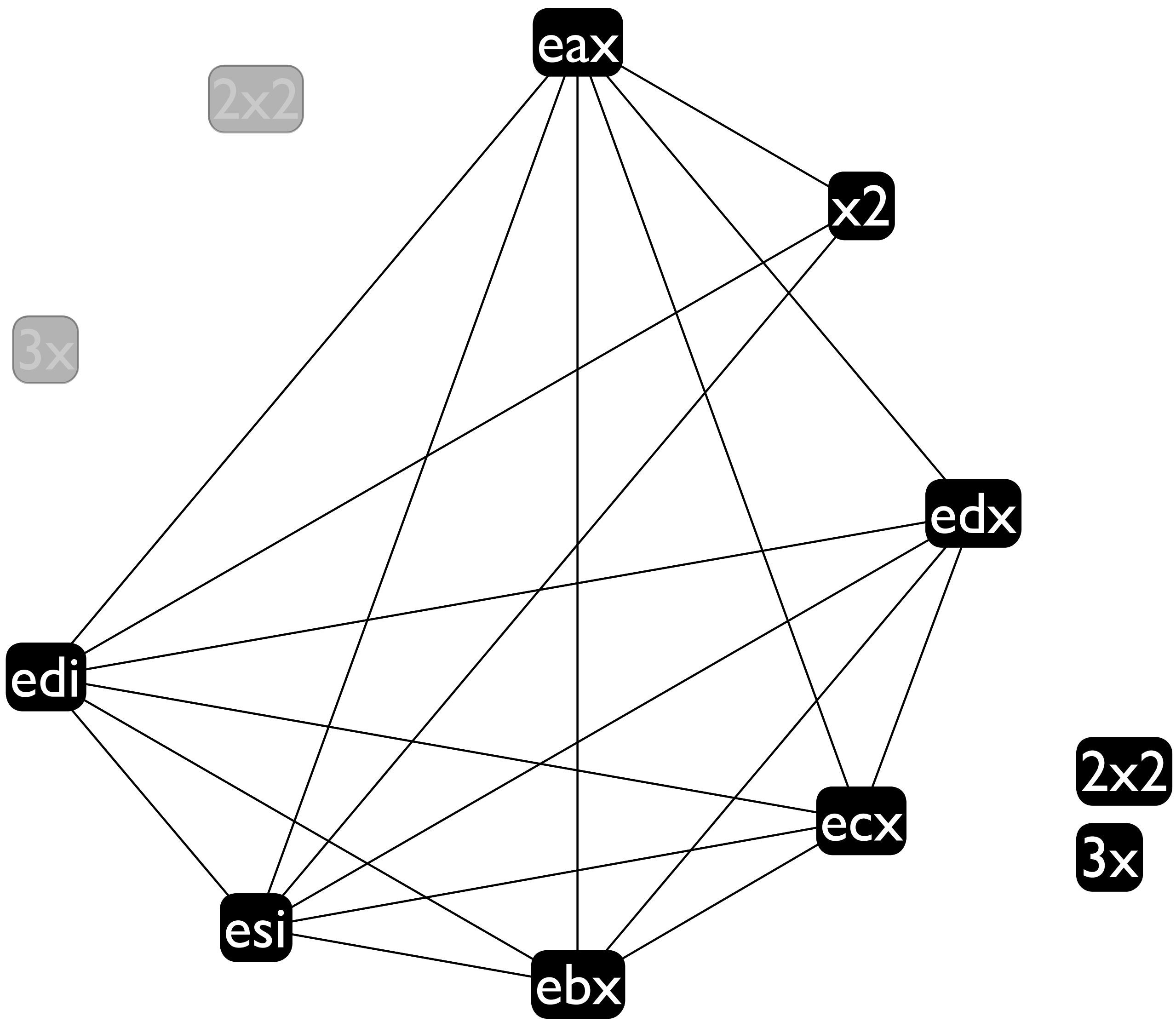
		in	out
1:	`:f`	(eax)	(eax edi esi)
2:	`(x2 <- eax)`	(eax edi esi)	(eax edi esi x2)
3:	`(x2 *= x2)`	(eax edi esi x2)	(eax edi esi x2)
4:	`(2x2 <- x2)`	(eax edi esi x2)	(2x2 eax edi esi)
5:	`(2x2 *= 2)`	(2x2 eax edi esi)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

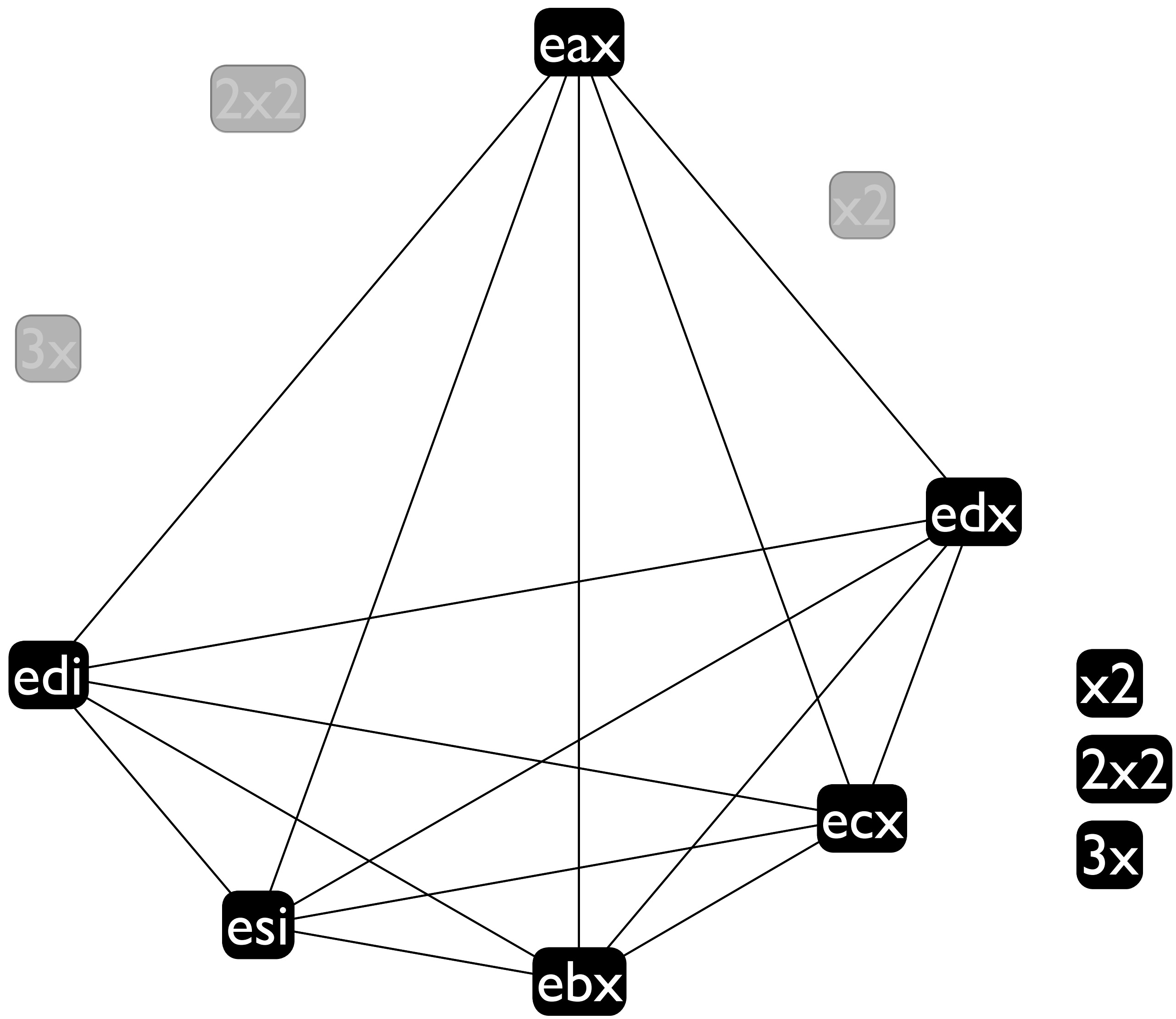
Liveness

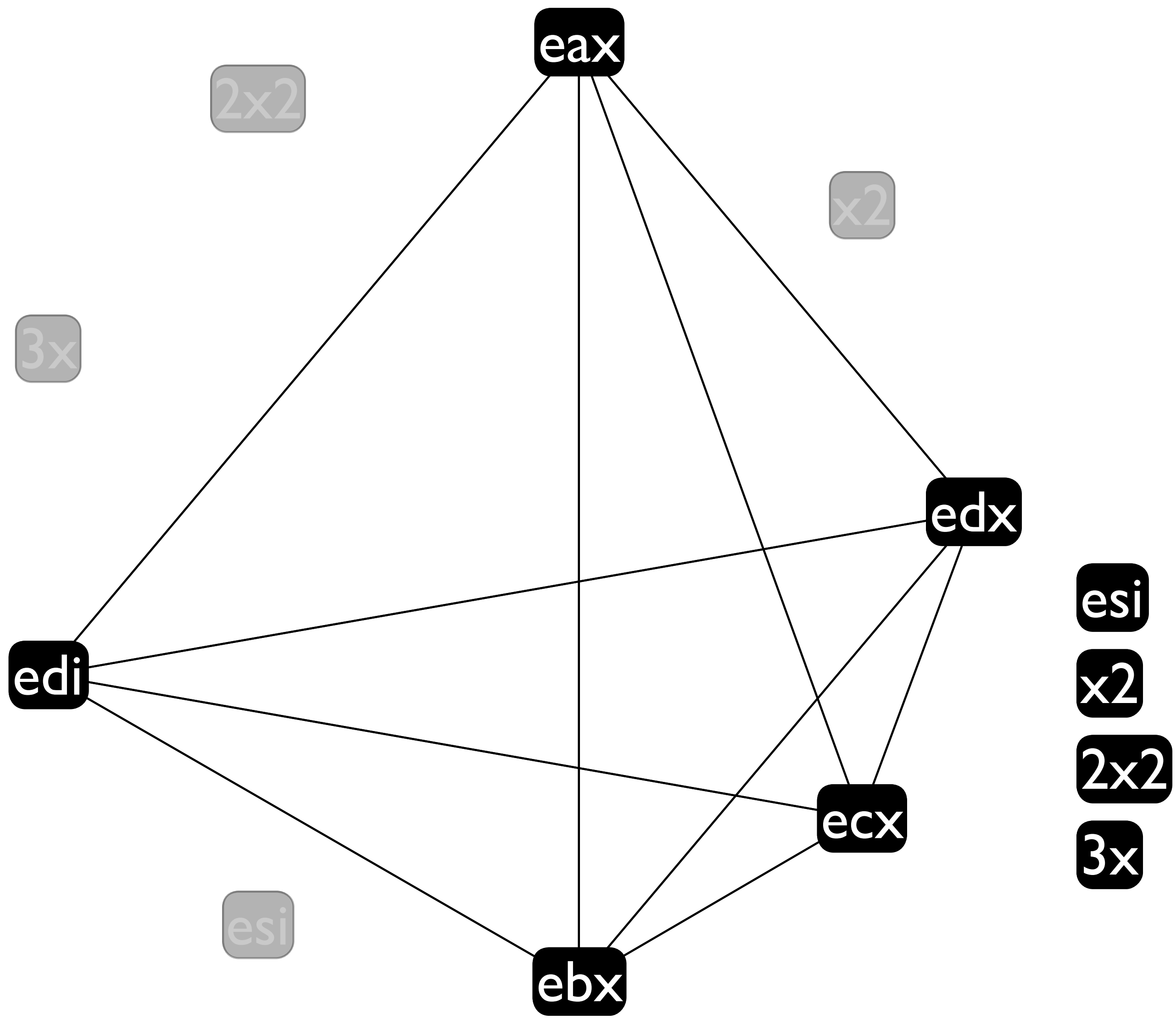
		in	out
1:	`:f`	(eax edi esi)	(eax edi esi)
2:	`(x2 <- eax)`	(eax edi esi)	(eax edi esi x2)
3:	`(x2 *= x2)`	(eax edi esi x2)	(eax edi esi x2)
4:	`(2x2 <- x2)`	(eax edi esi x2)	(2x2 eax edi esi)
5:	`(2x2 *= 2)`	(2x2 eax edi esi)	(2x2 eax edi esi)
6:	`(3x <- eax)`	(2x2 eax edi esi)	(2x2 3x edi esi)
7:	`(3x *= 3)`	(2x2 3x edi esi)	(2x2 3x edi esi)
8:	`(eax <- 2x2)`	(2x2 3x edi esi)	(3x eax edi esi)
9:	`(eax += 3x)`	(3x eax edi esi)	(eax edi esi)
10:	`(eax += 4)`	(eax edi esi)	(eax edi esi)
11:	`(return)`	(eax edi esi)	()

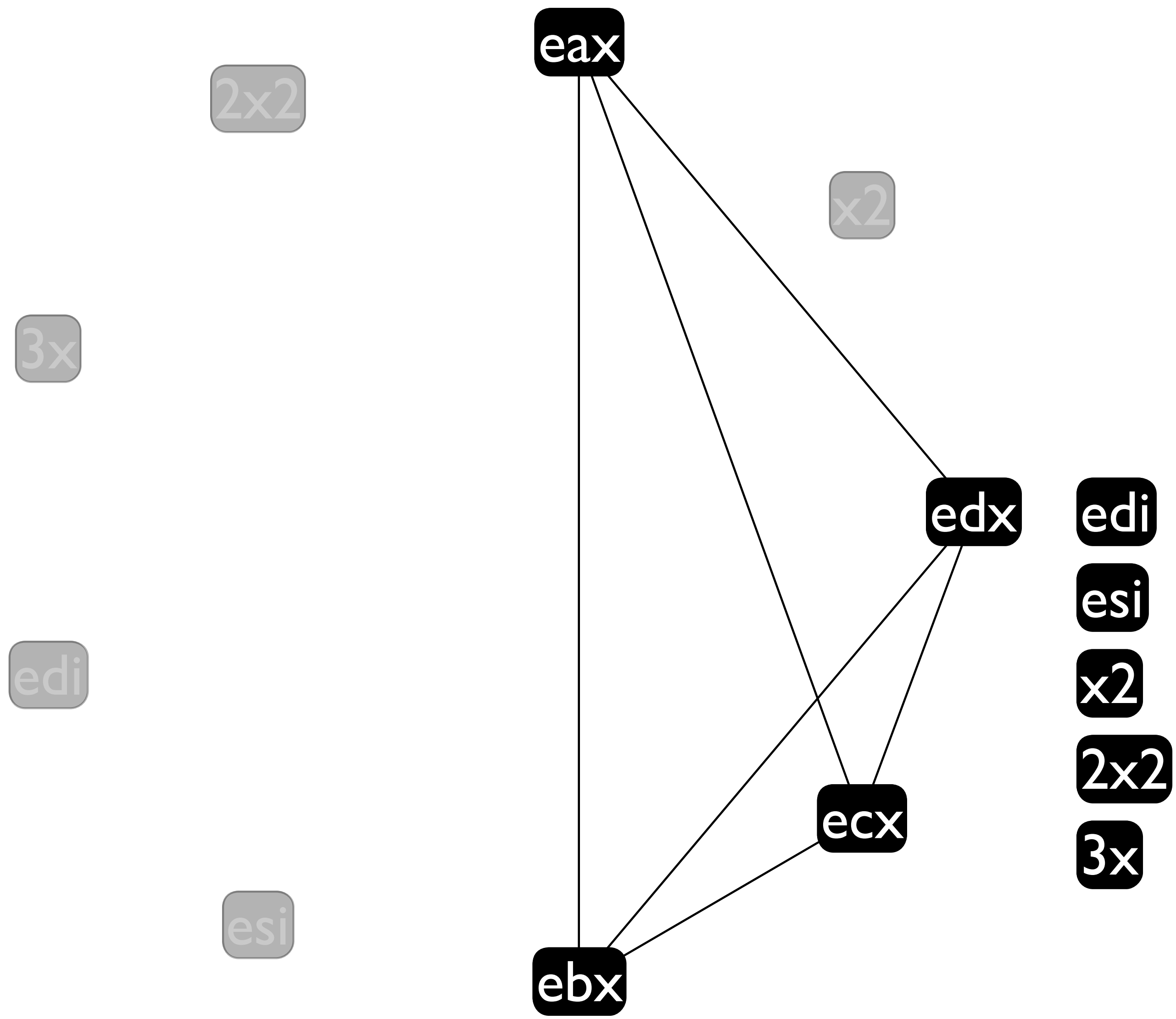

eax
2x2
x2
3x
edx
edi
ecx
esi
ebx

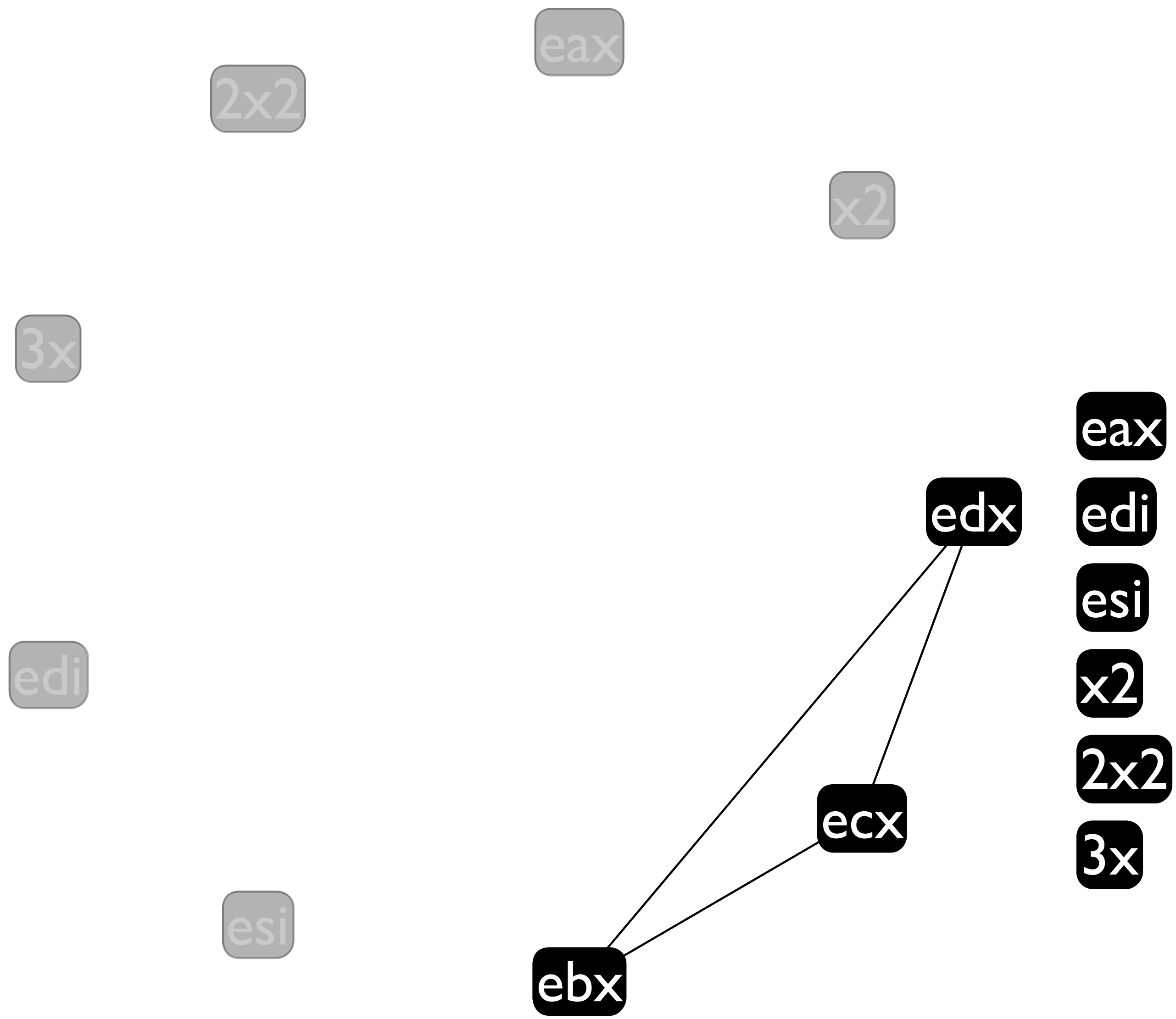

eax
2x2
x2
3x
edx
edi
ecx
3x
esi
ebx

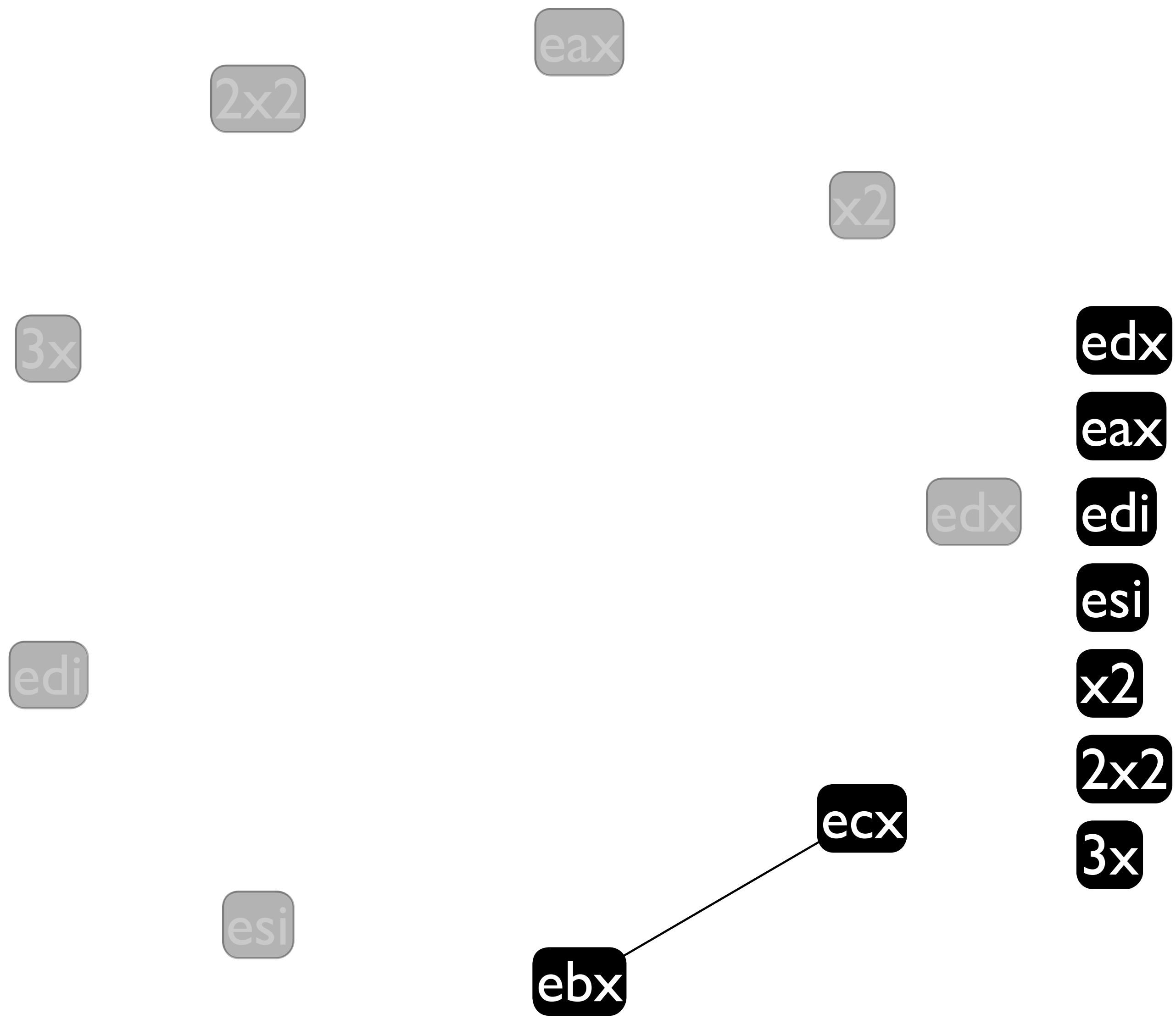

eax
2x2
x2
3x
edx
edi
2x2
ecx
3x
esi
ebx

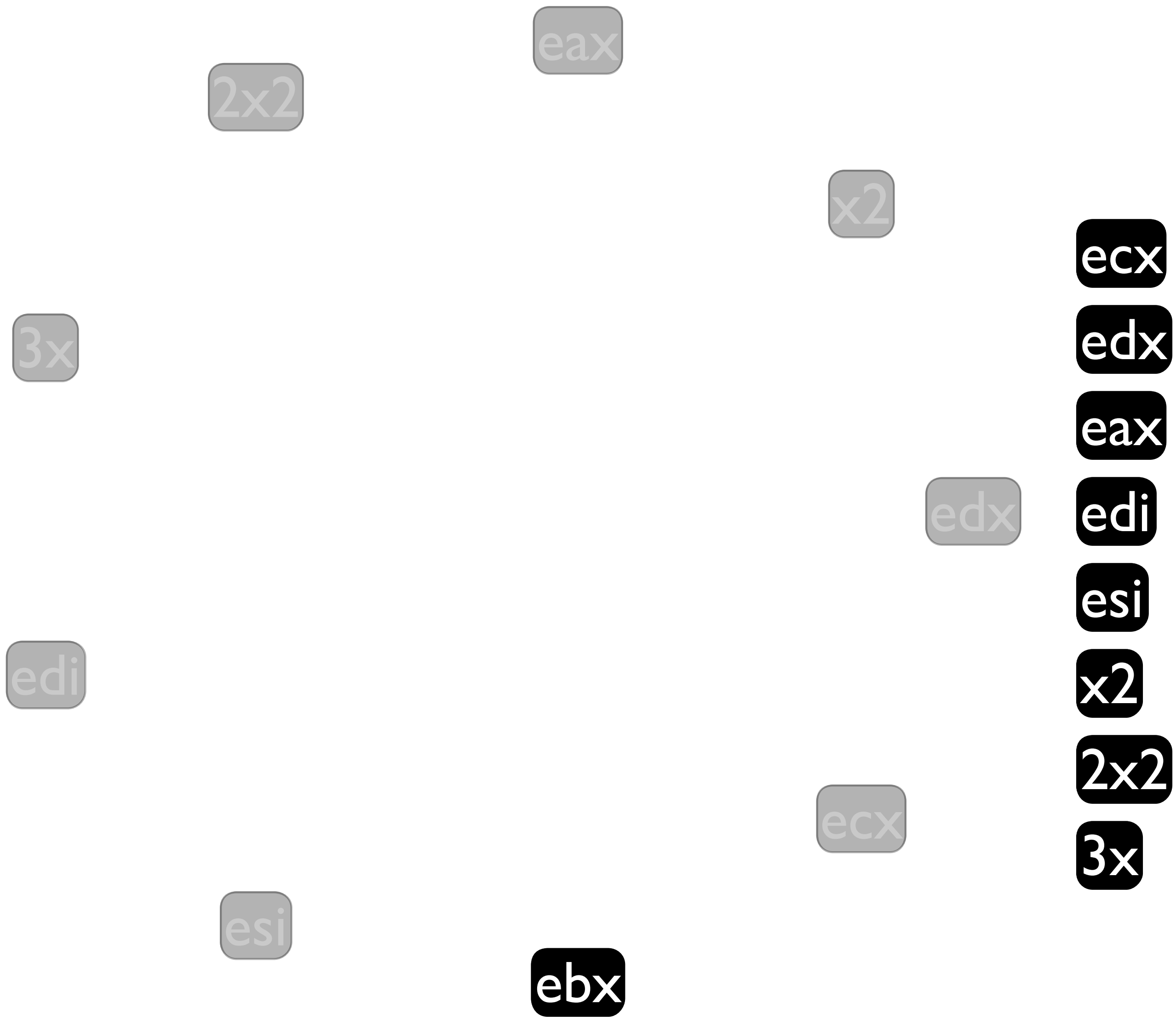

eax
2x2
x2
3x
edx
edi
x2
2x2
ecx
3x
esi
ebx

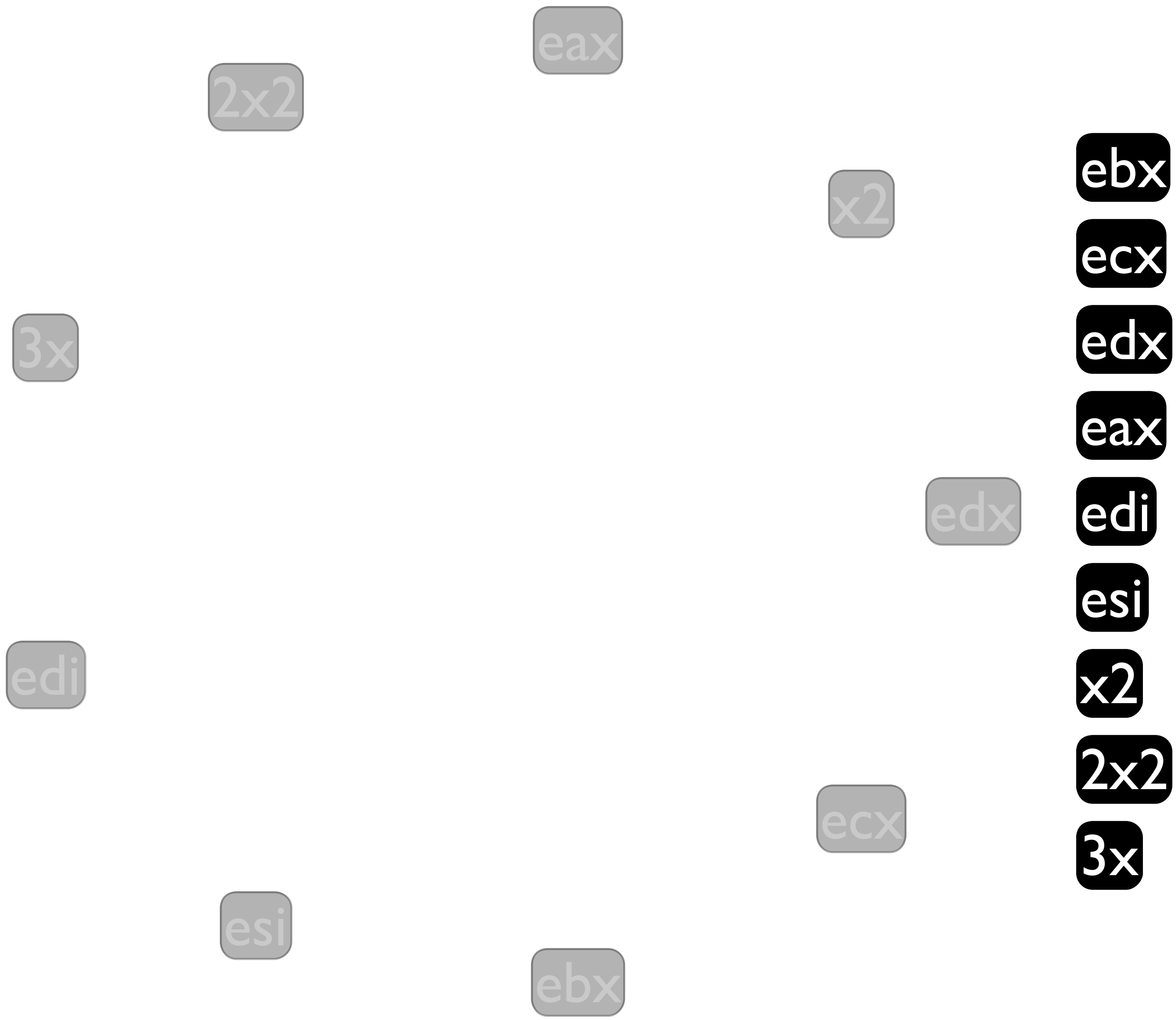

eax
2x2
x2
3x
edx
esi
x2
2x2
3x
edi
ecx
esi
ebx

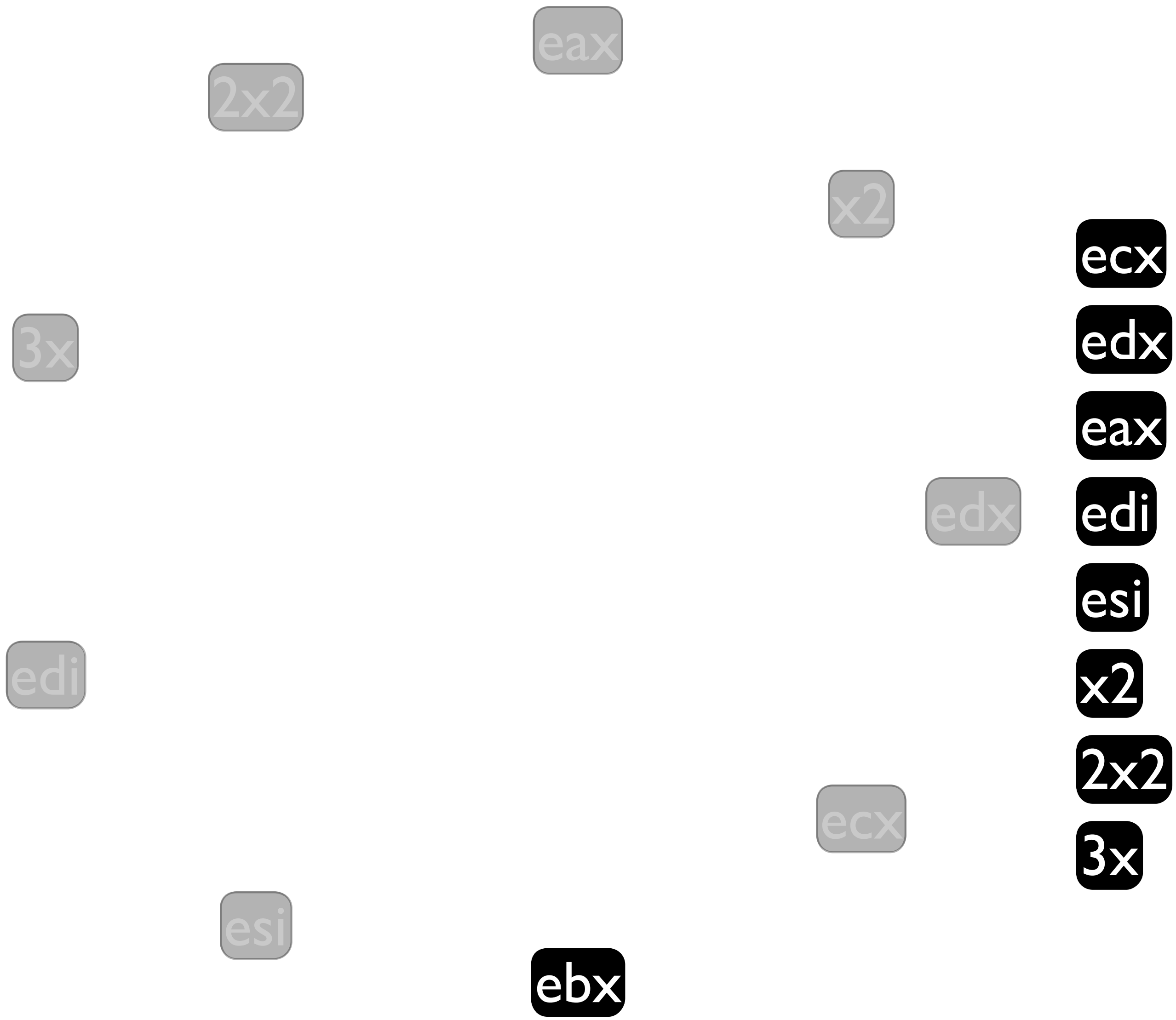

eax
2x2
x2
3x
edx
edi
esi
x2
2x2
3x
edi
ecx
esi
ebx

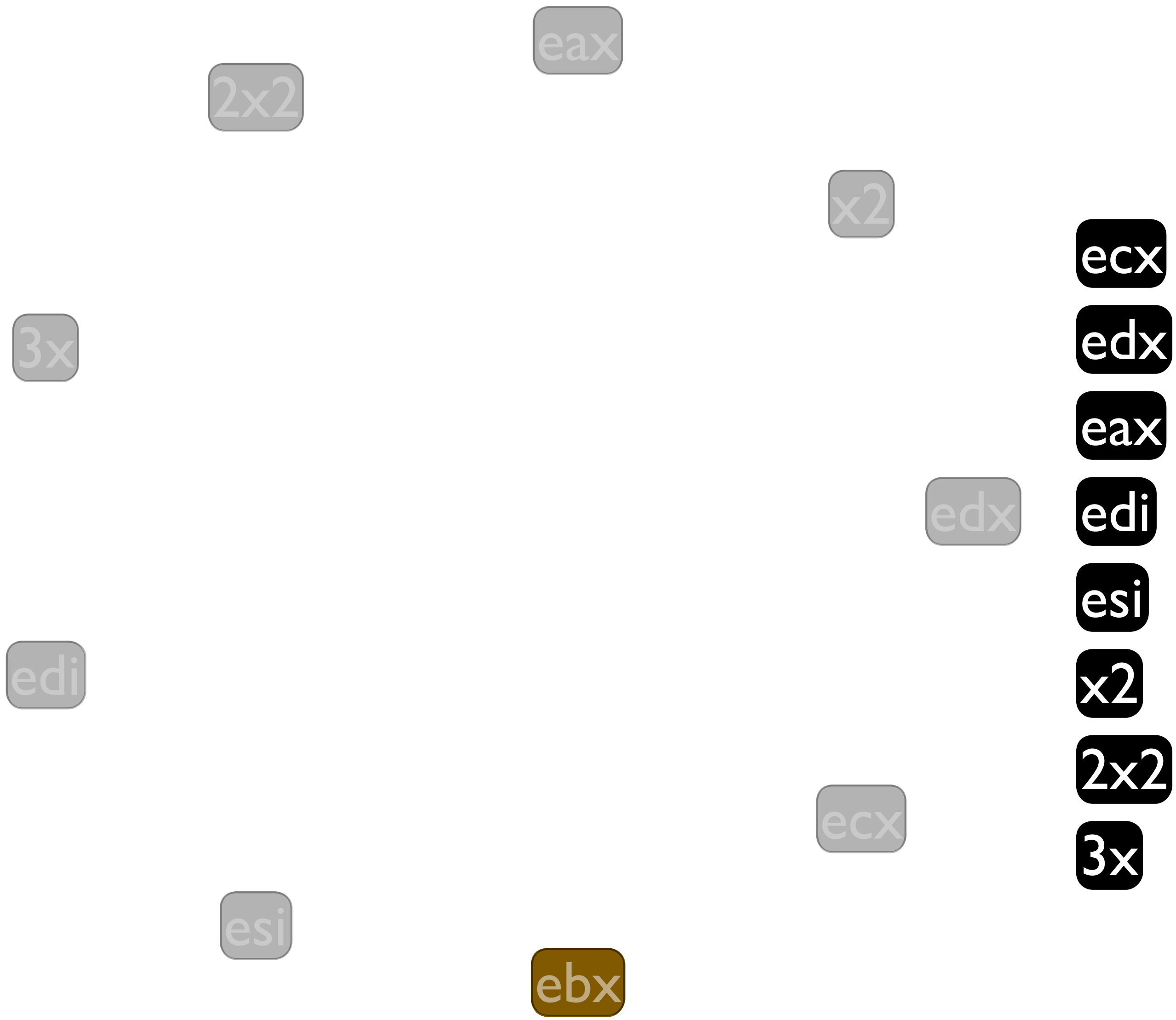

eax
2x2
x2
3x
edi
esi
ebx
ecx
edx
eax
edi
esi
x2
2x2
3x

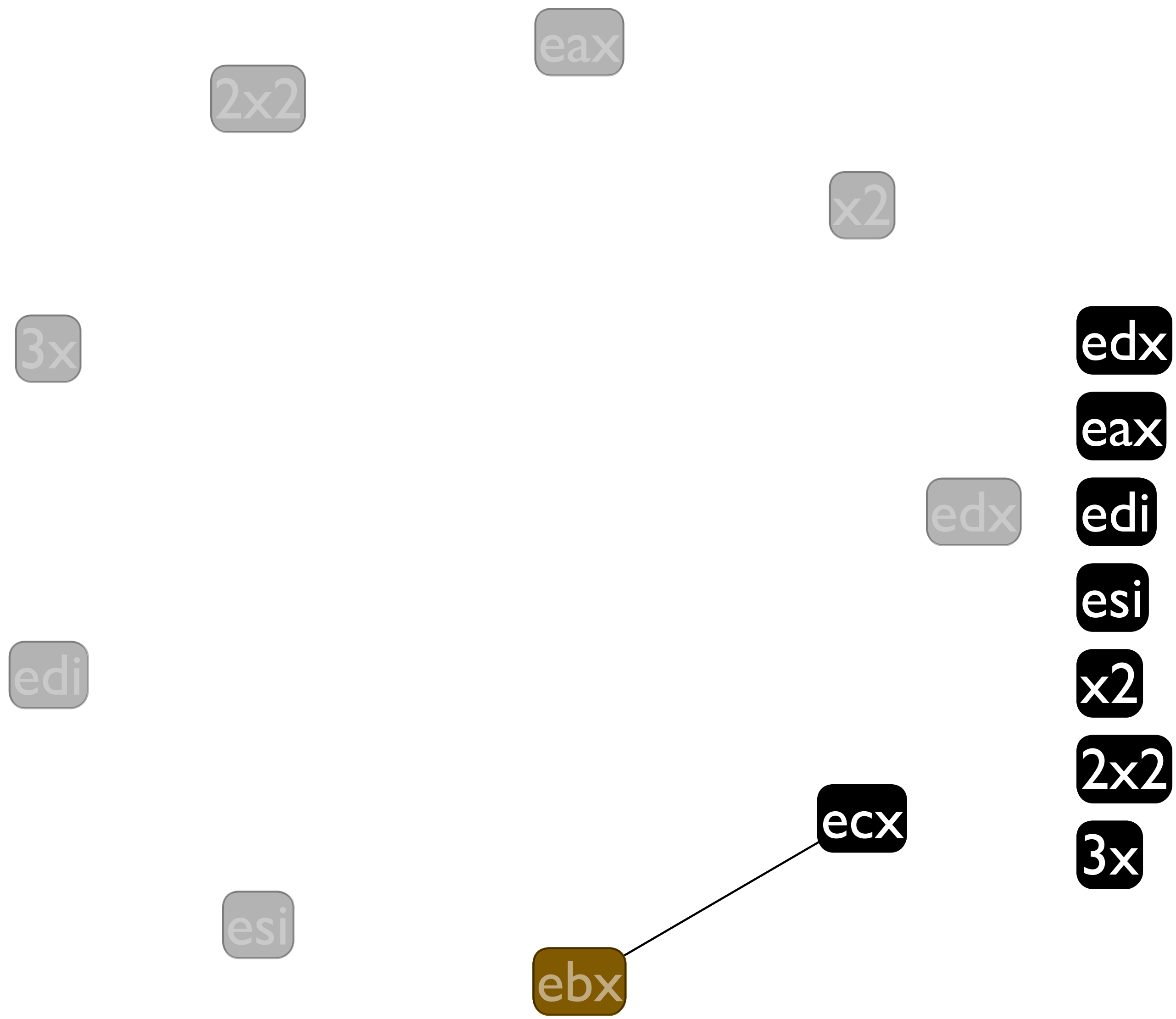

eax
2x2
x2
3x
edx
edi
ecx
esi
ebx
edx
eax
edi
esi
x2
2x2
3x

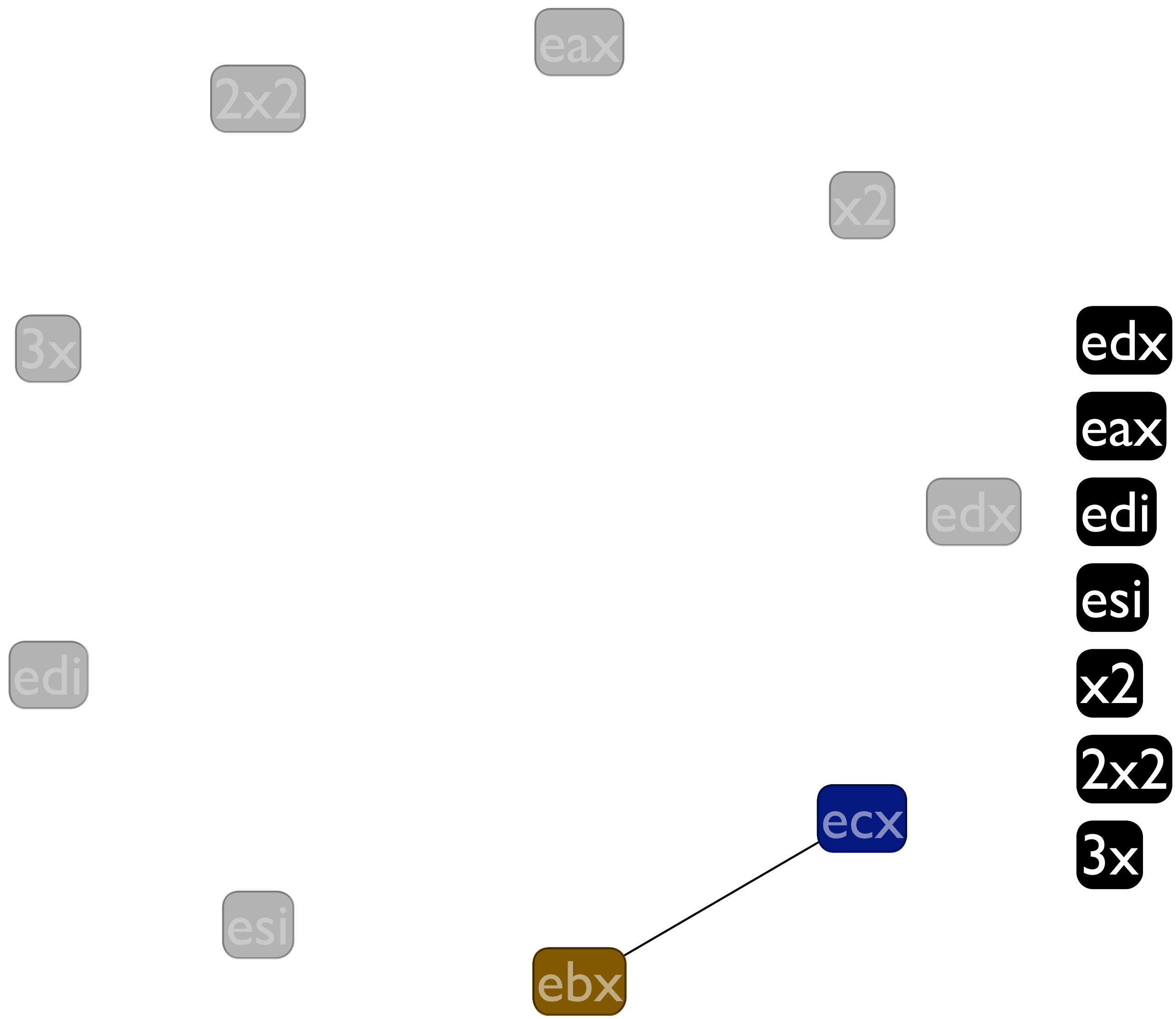

eax
2x2
x2
3x
edx
edi
ecx
esi
ebx
ecx
edx
eax
edi
esi
x2
2x2
3x


eax
2x2
x2
3x
edx
edi
ecx
esi
ebx
ebx
ecx
edx
eax
edi
esi
x2
2x2
3x

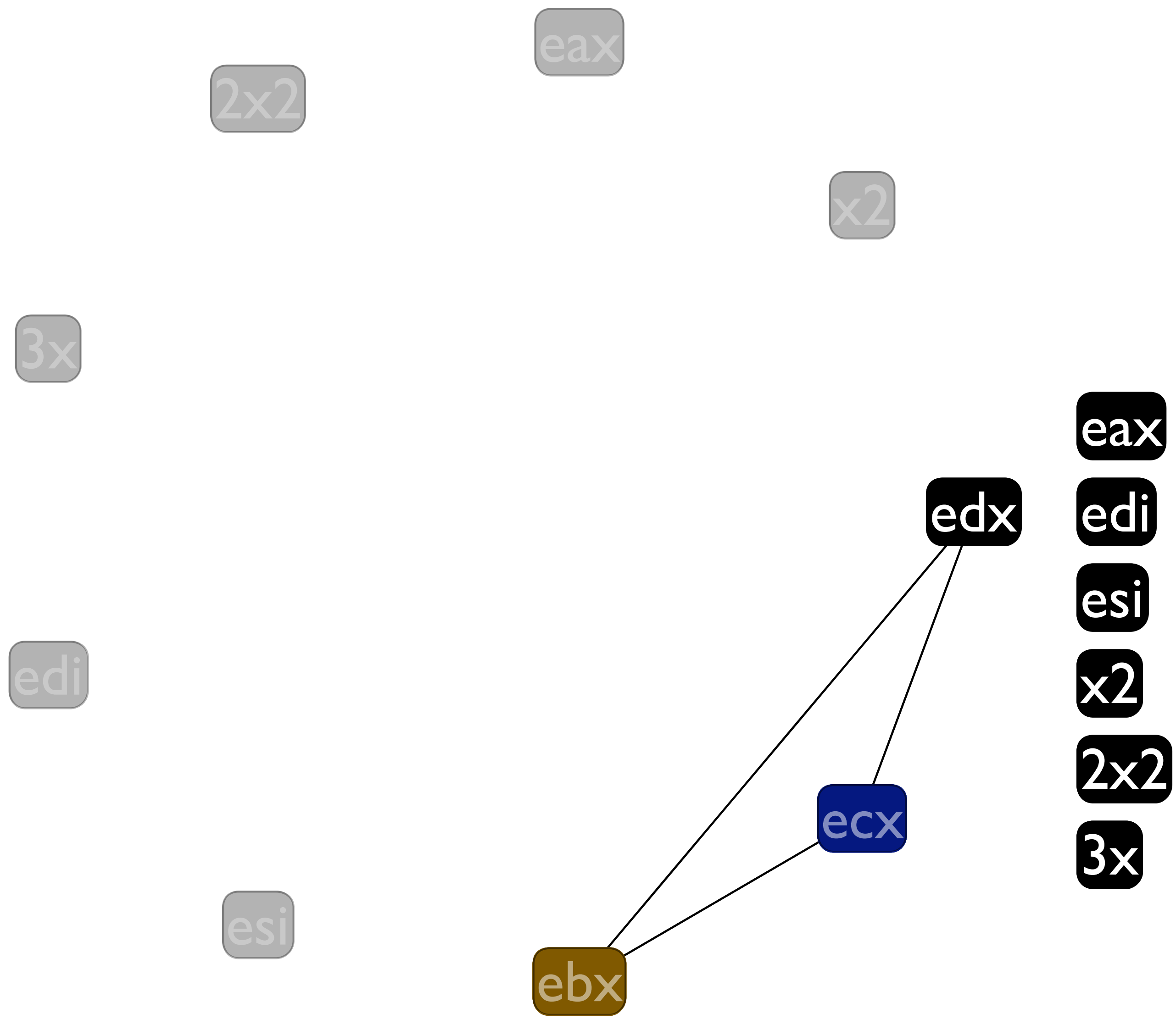

eax
2x2
x2
3x
edx
edi
ecx
esi
ebx
ecx
edx
eax
edi
esi
x2
2x2
3x


eax
2x2
x2
3x
edx
edi
ecx
esi
ebx
ecx
edx
eax
edi
esi
x2
2x2
3x

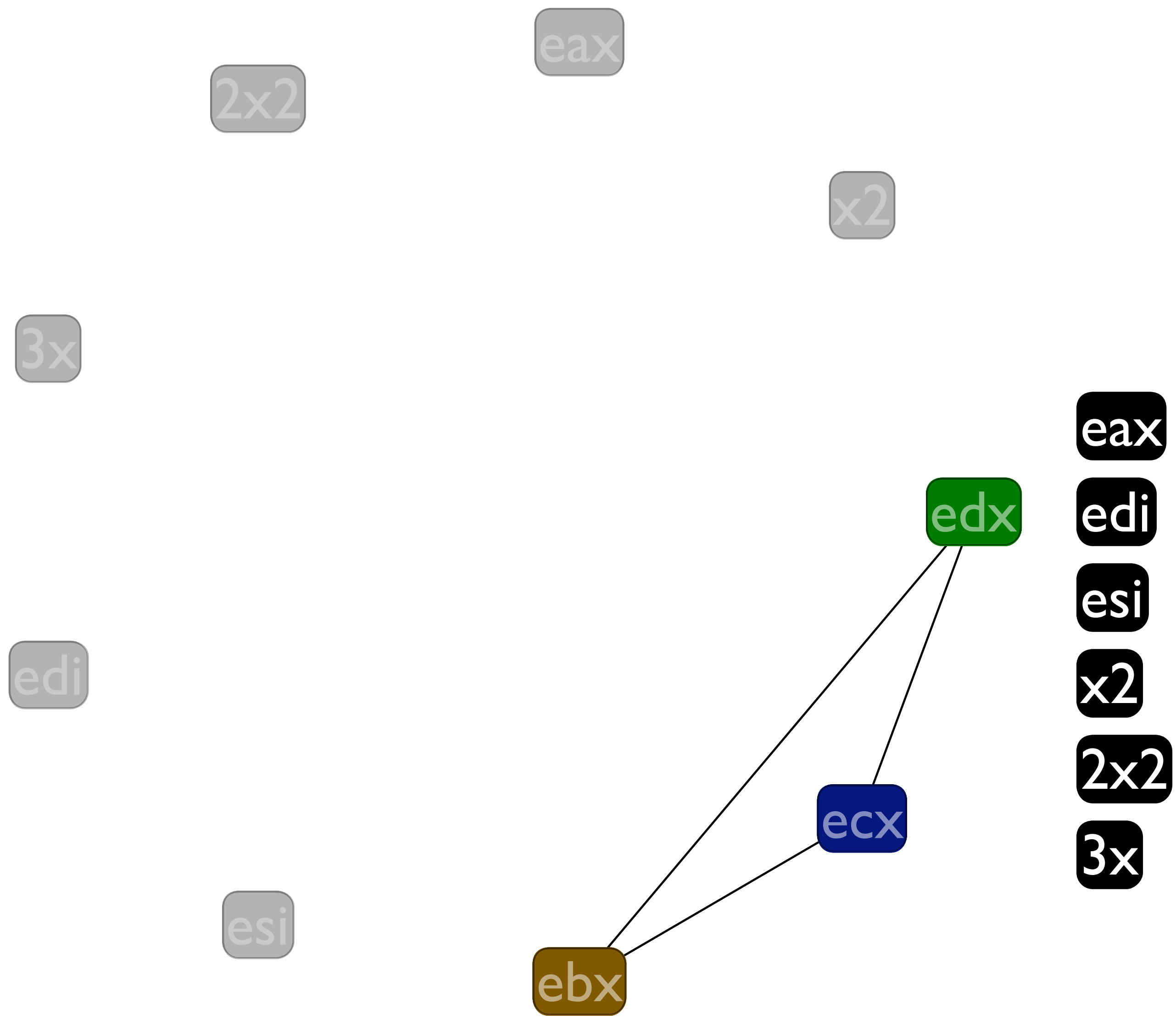

eax
2x2
x2
3x
edx
edi
ecx
esi
ebx
edx
eax
edi
esi
x2
2x2
3x

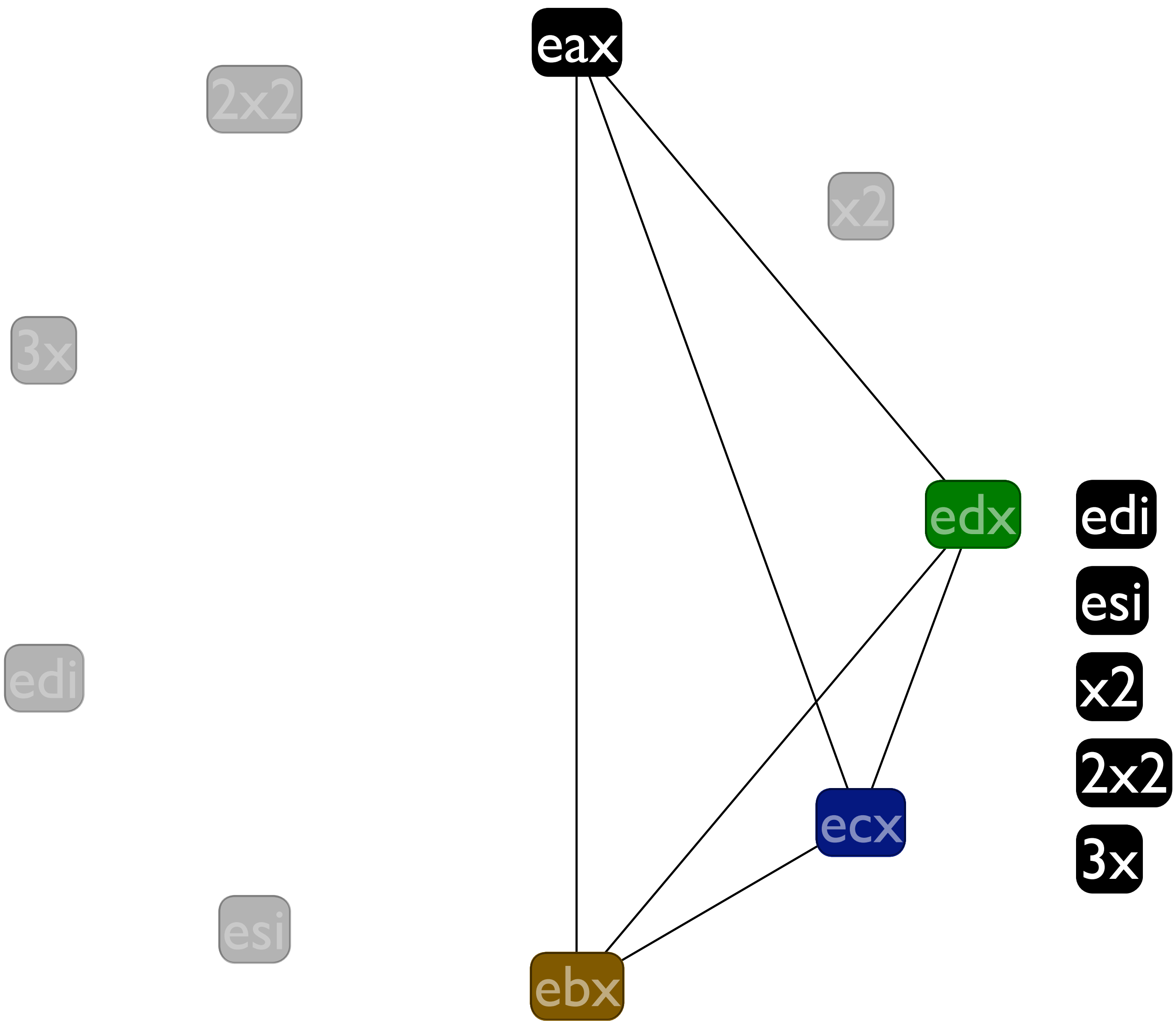


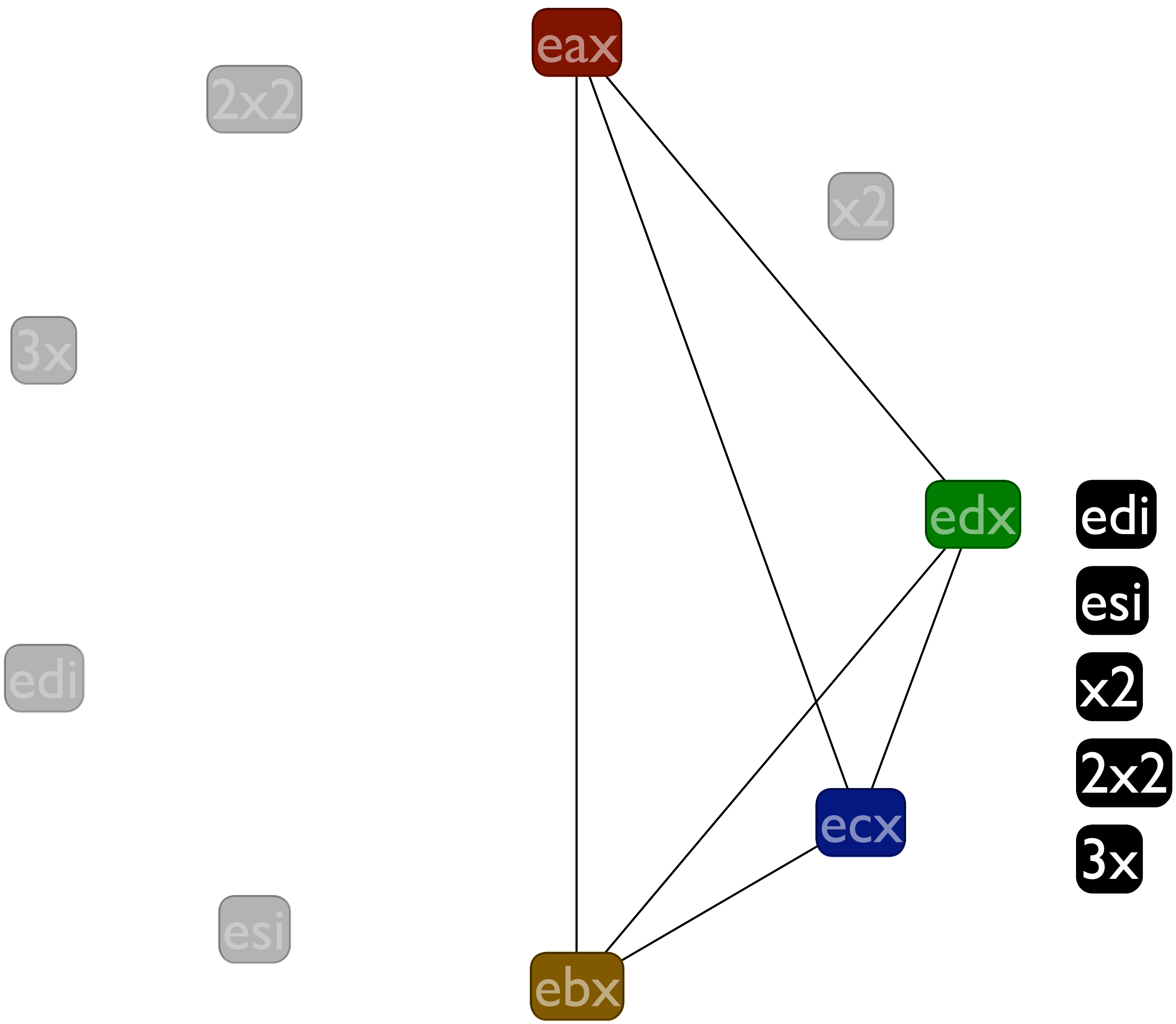
eax
2x2
x2
3x
edx
edi
ecx
esi
ebx
edx
eax
edi
esi
x2
2x2
3x

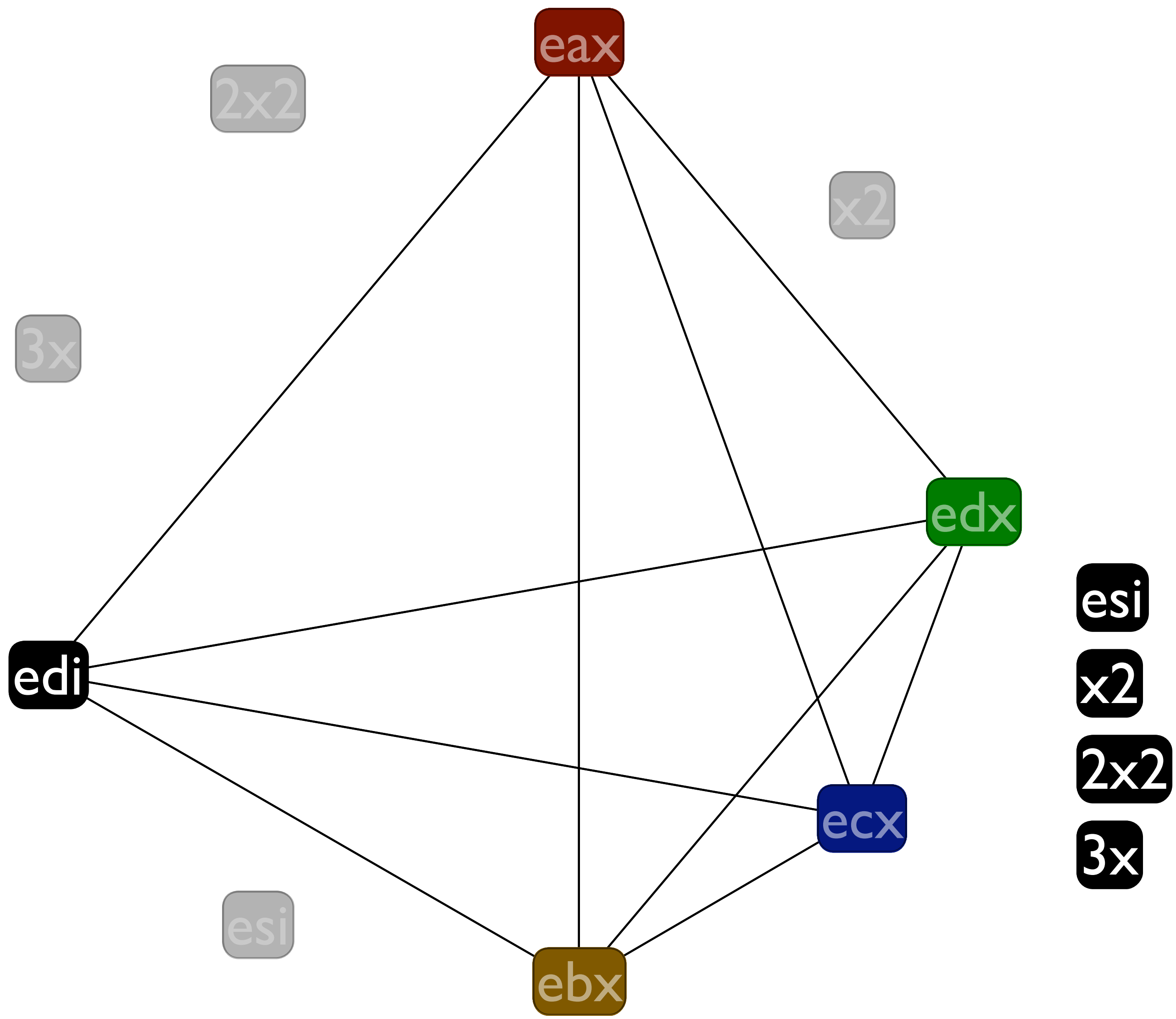


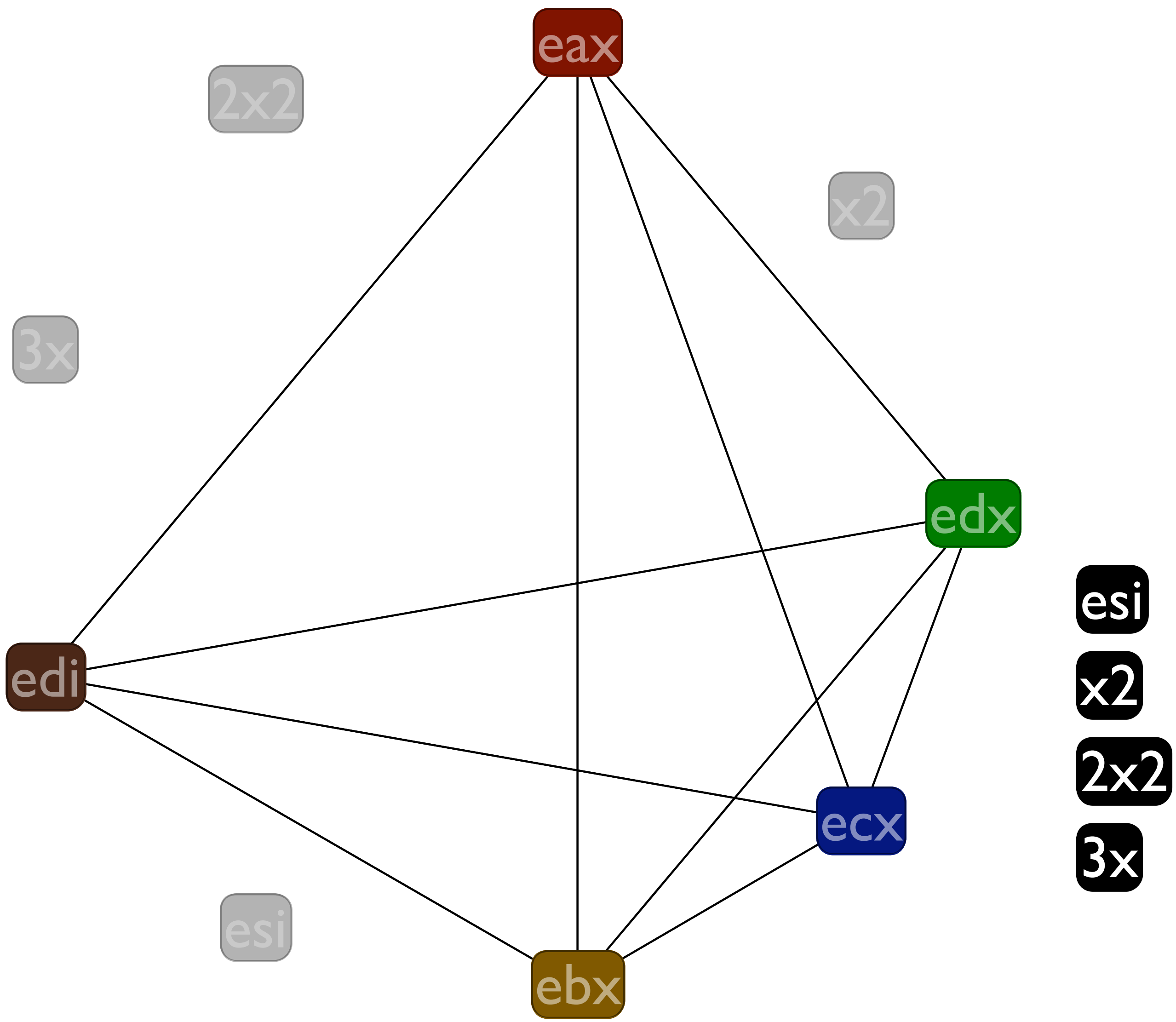
eax
2x2
x2
3x
edi
esi
ebx
ecx
edx
eax
edi
esi
x2
2x2
3x

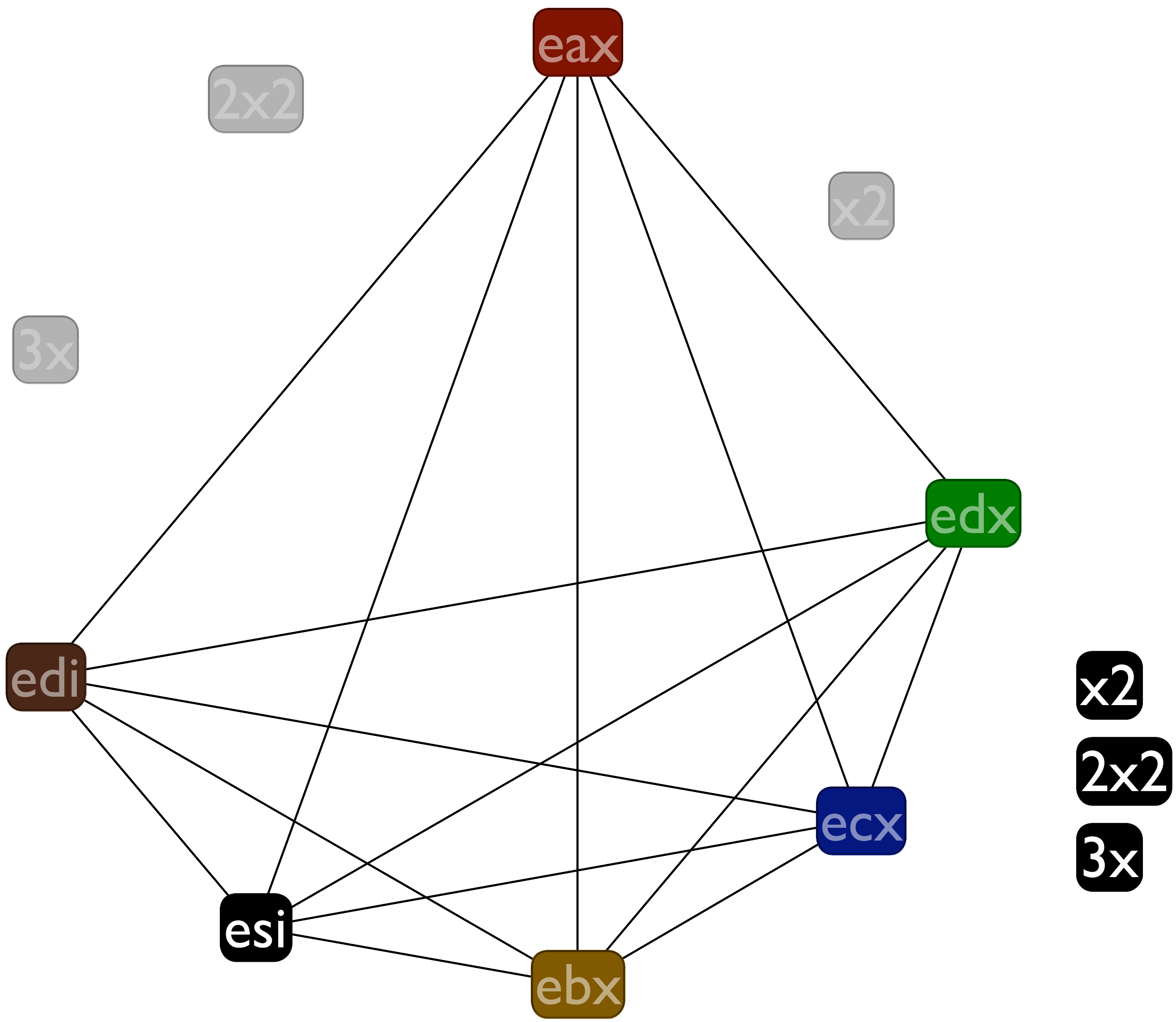

eax
2x2
x2
3x
edx
edi
ecx
esi
ebx
eax
edi
esi
x2
2x2
3x

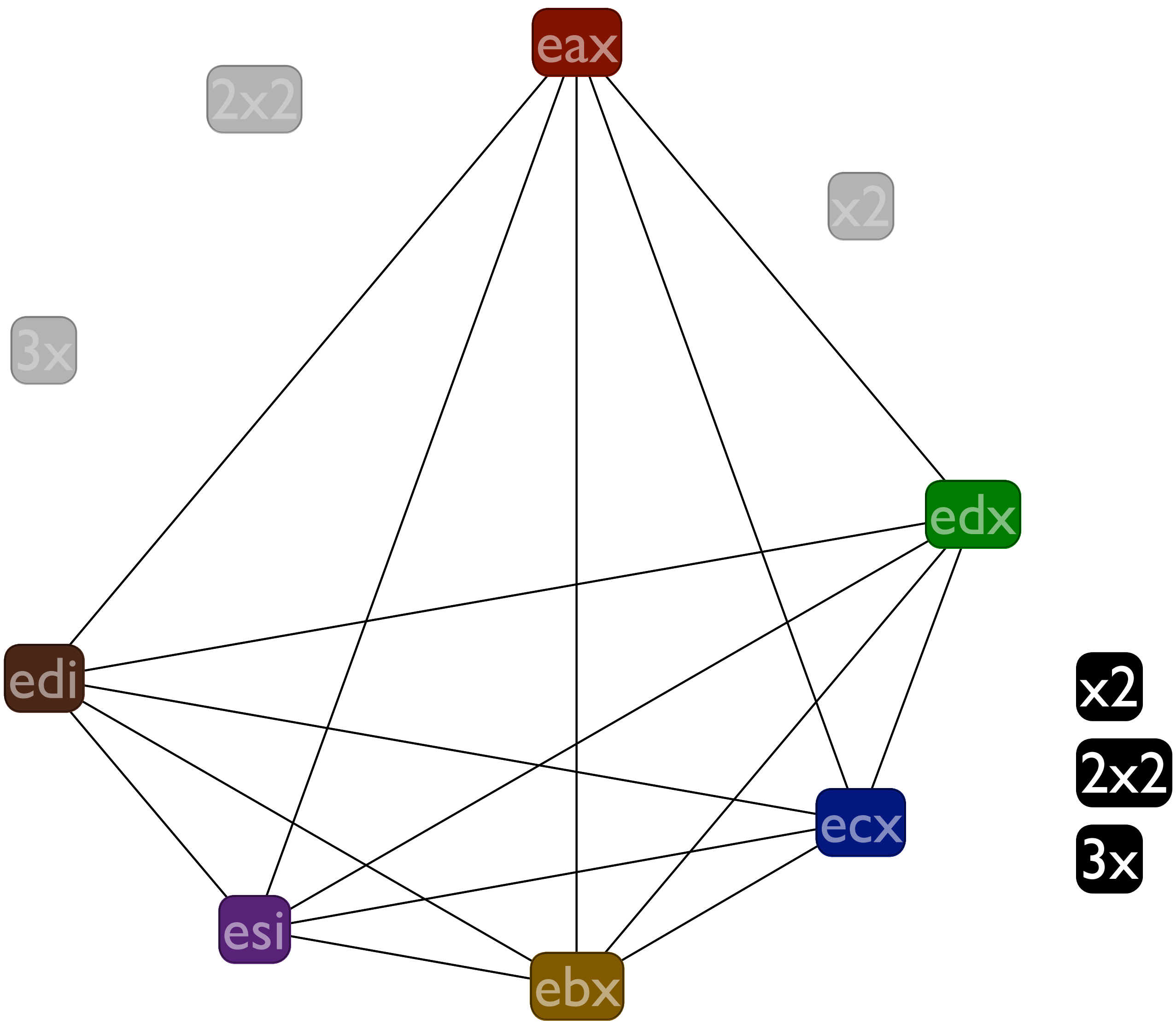

eax
2x2
x2
3x
edx
edi
esi
x2
2x2
3x
edi
ecx
esi
ebx

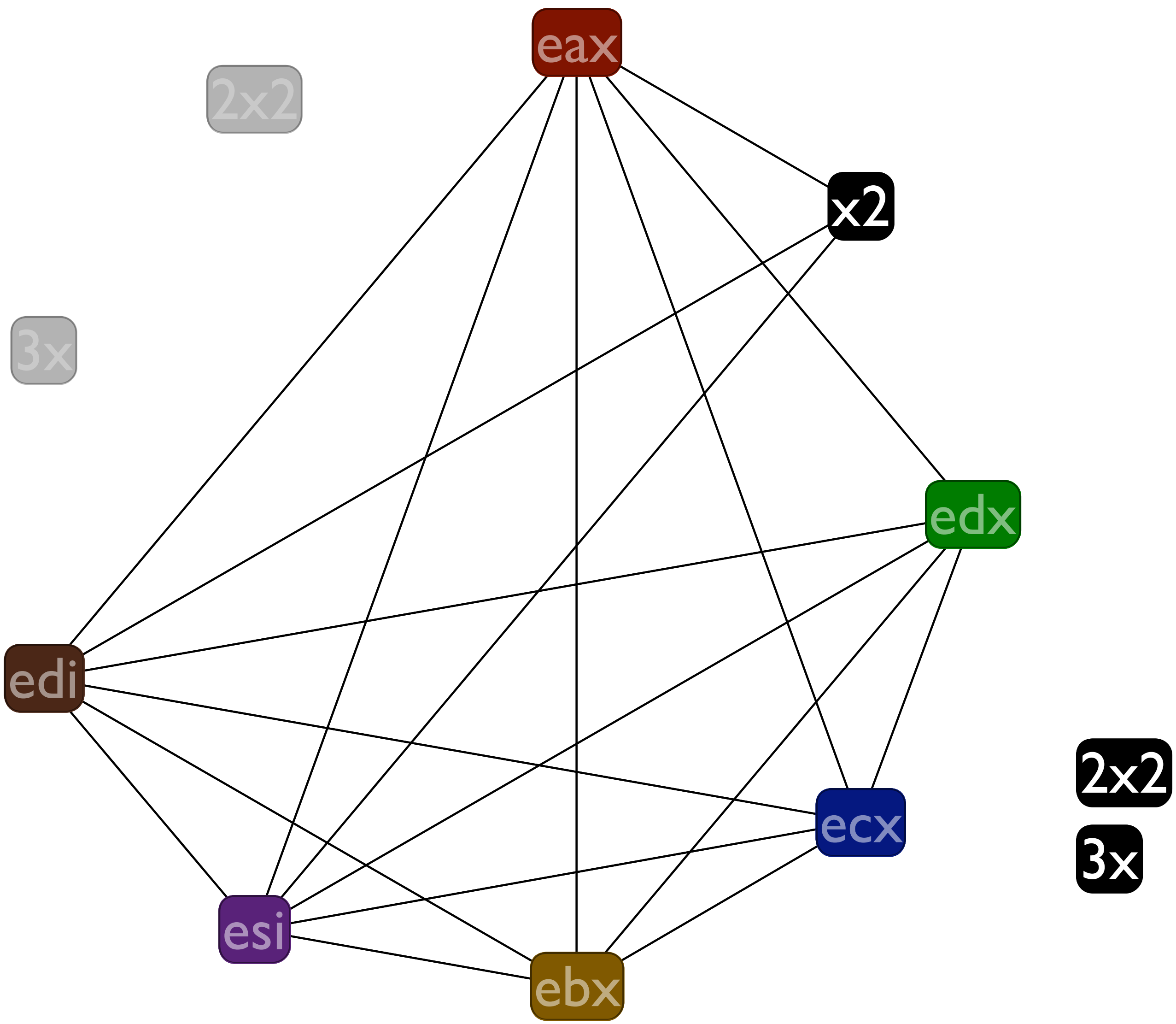

eax
2x2
x2
3x
edx
edi
esi
x2
2x2
3x
edi
ecx
esi
ebx

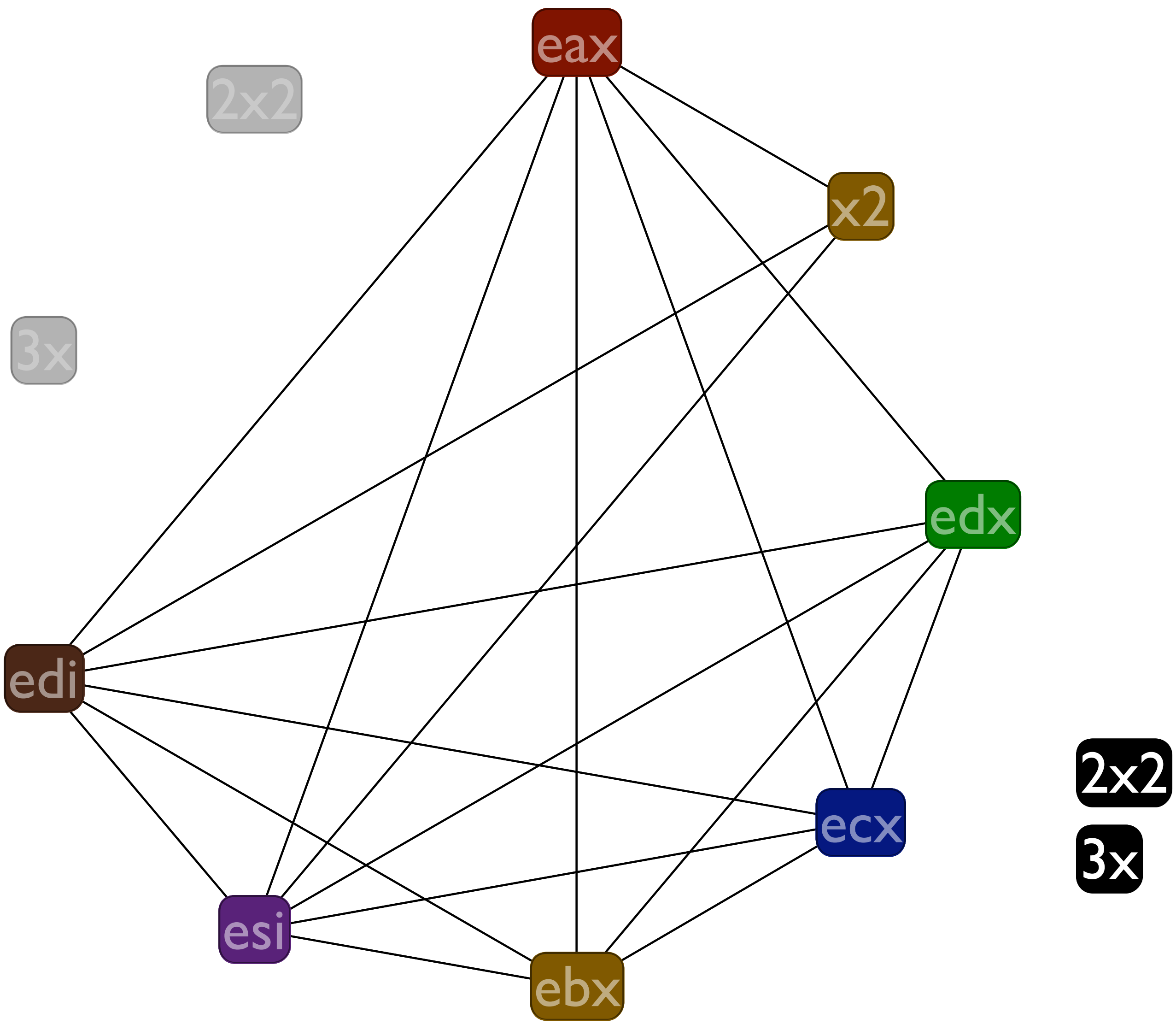

eax
2x2
x2
3x
edx
esi
x2
2x2
3x
edi
ecx
esi
ebx

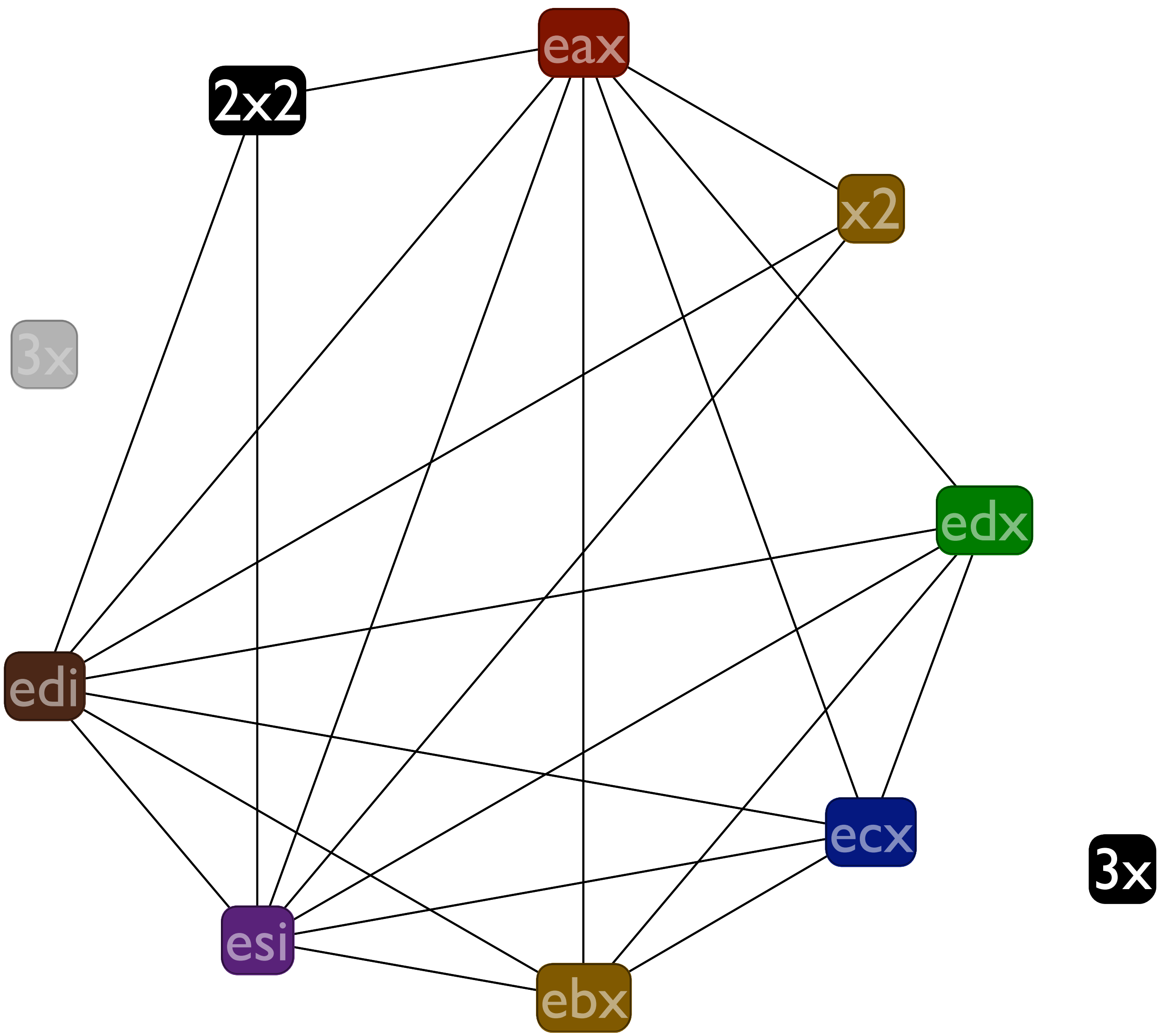

eax
2x2
x2
3x
edx
edi
ecx
esi
ebx
esi
x2
2x2
3x


eax
2x2
x2
3x
edx
edi
x2
2x2
ecx
3x
esi
ebx

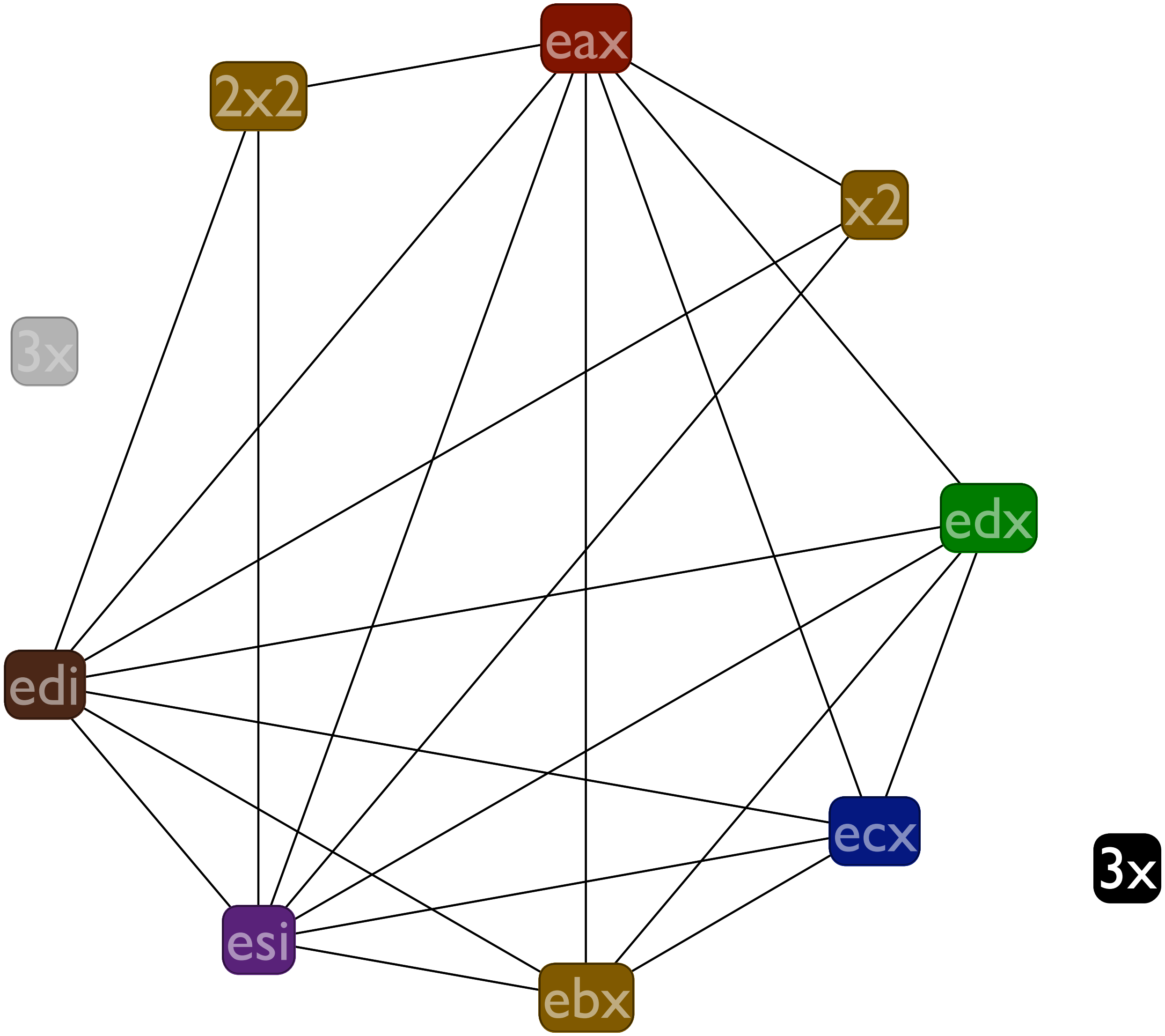

eax
2x2
x2
3x
edx
edi
x2
2x2
ecx
3x
esi
ebx

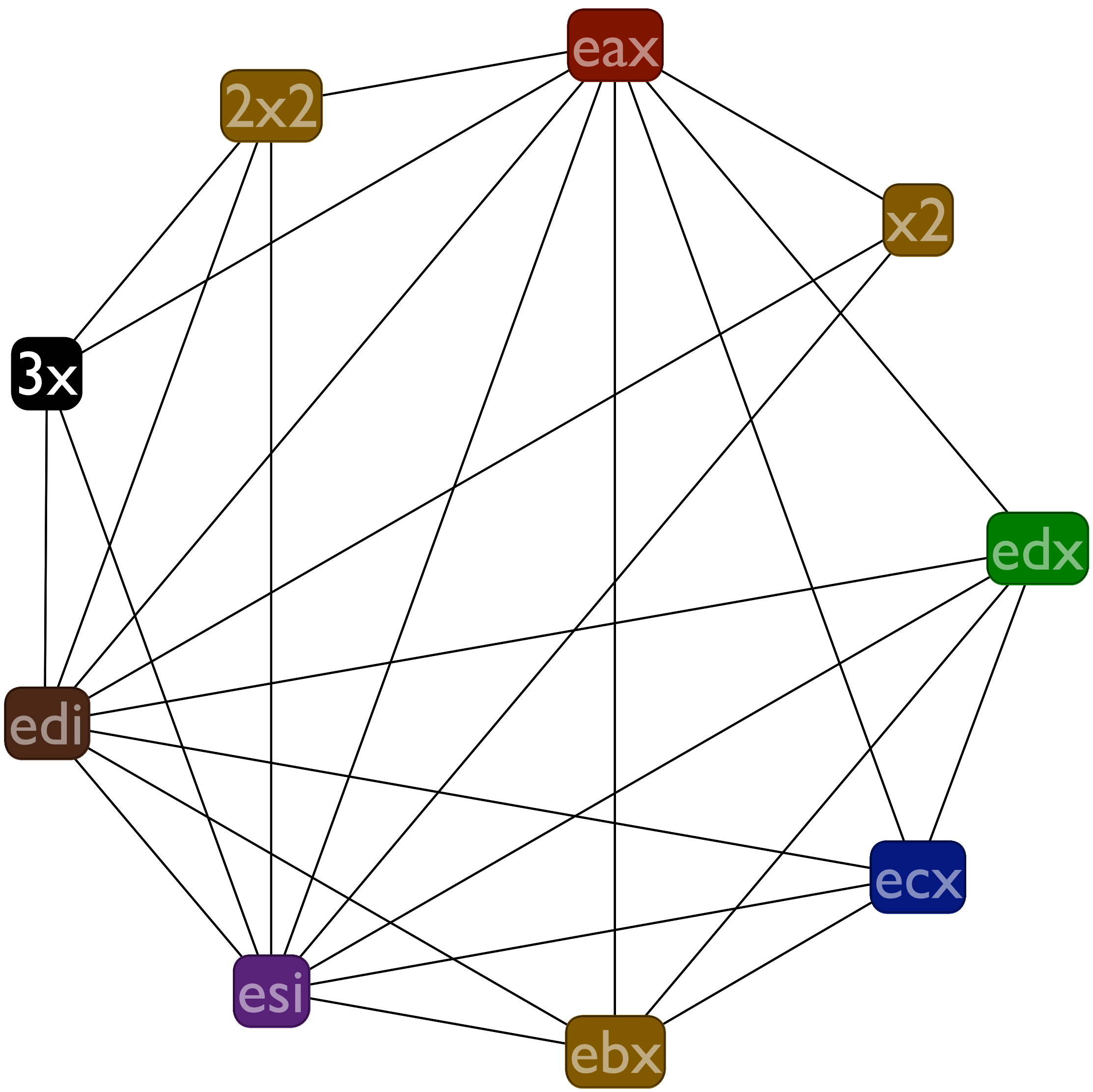

eax
2x2
x2
3x
edx
edi
2x2
3x
ecx
esi
ebx

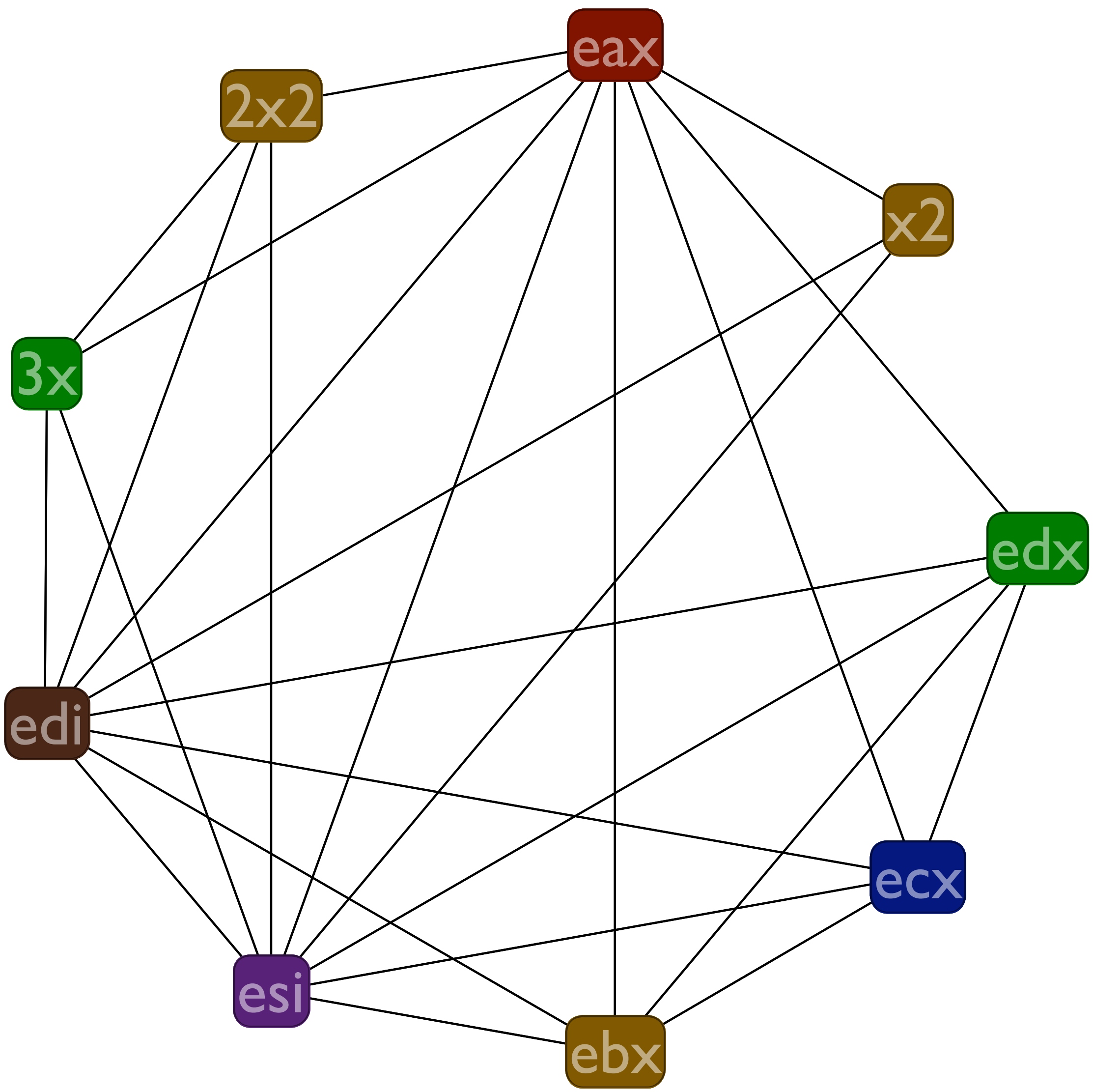

eax
2x2
x2
3x
edx
edi
2x2
3x
ecx
esi
ebx



eax
2x2
x2
3x
edx
edi
ecx
3x
esi
ebx


eax
2x2
x2
3x
edx
edi
ecx
3x
esi
ebx


eax
2x2
x2
3x
edx
edi
ecx
esi
ebx


eax
2x2
x2
3x
edx
edi
ecx
esi
ebx