
Operational notes

Document updated on **March 27, 2022**.

The following colors are **not** part of the final product, but serve as highlights in the editing/review process:

- text that needs attention from the Subject Matter Experts: Mirco, Anna,& Jan
- terms that have not yet been defined in the book
- things that need to be checked only at the very final typesetting stage (and it doesn't make sense to do them before)
- text that needs advice from the communications/marketing team: Aaron & Shane
- text that needs to be completed or otherwise edited (by Sylvia)

11 Todo list

12	zero-knowledge proofs	12
13	played with	12
14	finite field	12
15	elliptic curve	12
16	Update reference when content is finalized	12
17	methatical	12
18	add some more informal explanation of absolute value	14
19	We haven't really talked about what a ring is at this point	14
20	What's the significance of this distinction?	15
21	reverse	15
22	Turing machine	15
23	polynomial time	15
24	sub-exponentially, with $\mathcal{O}((1 + \varepsilon)^n)$ and some $\varepsilon > 0$	15
25	Add text	16
26	\mathbb{Q} of fractions	16
27	Division in the usual sense is not defined for integers	16
28	Add more explanation of how this works	17
29	pseudocode	18
30	check reference	18
31	modular arithmetics	18
32	actual division	18
33	multiplicative inverses	18
34	factional numbers	18
35	exponentiation function	20
36	See XXX	20
37	once they accept that this is a new kind of calculations, its actually not that hard	20
38	perform Euclidean division on them	20
39	This Sage snippet should be described in more detail.	21
40	prime fields	23
41	residue class rings	23
42	Algorithm sometimes floated to the next page, check this for final version	23
43	Add a number and title to the tables	25
44	(-1) should be $(-a)$?	26
45	add reference	28
46	rephrase	32
47	subtrahend	33
48	minuend	33
49	what does this mean?	37
50	Def Subgroup, Fundamental theorem of cyclic groups.	40

51	add reference	41
52	Add real-life example of 0?	41
53	add reference	41
54	check reference	42
55	check references to previous examples	43
56	RSA crypto system	43
57	size 2048-bits	43
58	check reference	43
59	add reference: 27?	43
60	check reference	44
61	polynomial time	44
62	exponential time	44
63	TODO: Fundamental theorem of finite cyclic groups	44
64	check reference	44
65	runtime complexity	45
66	add reference	45
67	S: what does “efficiently” mean here?	45
68	computational hardness assumptions	45
69	check reference	45
70	check reference	46
71	explain last sentence more	46
72	“equation”?	47
73	check reference	47
74	what’s the difference between \mathbb{F}_p^* and \mathbb{Z}_p^* ?	47
75	Legendre symbol	47
76	Euler’s formular	47
77	These are only explained later in the text, “	47
78	are these going to be relevant later?	48
79	TODO: theorem: every factor of order defines a subgroup...	48
80	Is there a term for this property?	49
81	a few examples?	51
82	check reference	51
83	TODO: DOUBLE CHECK THIS REASONING.	51
84	Mirco: We can do better than this	53
85	check reference	54
86	add reference	55
87	pseudorandom	55
88	oracle	55
89	check reference	55
90	add text on this	55
91	check reference	57
92	check reference	57
93	check reference	57
94	check reference	58
95	add more examples protocols of SNARK	58
96	check reference	58
97	add reference	58
98	Abelian groups	58

99	■ codomain	58
100	■ Check change of wording	59
101	■ add reference	60
102	■ Expand on this?	60
103	■ check reference	60
104	■ unify \mathbb{Z}_5 and \mathbb{F}_5 across the board?	61
105	■ S: are we introducing elliptic curves in section 1 or 2?	61
106	■ check reference	62
107	■ check reference	62
108	■ add reference	62
109	■ check reference	62
110	■ write paragraph on exponentiation	63
111	■ add reference	63
112	■ check reference	63
113	■ add reference	63
114	■ group pairings	63
115	■ add reference	64
116	■ check reference	64
117	■ check reference	67
118	■ add reference	68
119	■ TODO: Elliptic Curve asymmetric cryptography examples. Private key, generator,	
120	public key.	70
121	■ add reference	70
122	■ maybe remove this sentence?	70
123	■ affine space	70
124	■ cusps	71
125	■ self-intersections	71
126	■ check reference	72
127	■ check reference	73
128	■ jubjub	73
129	■ check reference	73
130	■ affine plane	73
131	■ check reference	74
132	■ check reference	74
133	■ check reference	75
134	■ sign	75
135	■ more explanation of what the sign is	75
136	■ check reference	75
137	■ S: I don't follow this at all	76
138	■ check reference	76
139	■ add explanation of how this shows what we claim	76
140	■ should this def. be moved even earlier?	77
141	■ chord line	77
142	■ tangential	77
143	■ tangent line	77
144	■ remove Q ?	77
145	■ where?	78
146	■ check reference	78

















































147	check reference	78
148	check reference	78
149	check reference	79
150	check reference	79
151	check reference	80
152	check reference	80
153	check reference	81
154	add term	81
155	add term	81
156	add reference	81
157	cofactor clearing	81
158	add reference	81
159	check reference	81
160	check reference	82
161	add reference	82
162	add reference	82
163	check reference	82
164	check reference	82
165	check reference	83
166	check reference	83
167	check reference	83
168	Explain how	83
169	write example	84
170	check reference	84
171	add reference	84
172	check reference	84
173	add reference	85
174	check reference	85
175	add reference	85
176	check reference	85
177	add reference	85
178	check reference	85
179	add reference	85
180	add reference	85
181	add reference	85
182	check reference	85
183	check reference	85
184	Check if following Alg is floated too far	86
185	add reference	86
186	add reference	86
187	write up this part	86
188	is the label in L ^A T _E X correct here?	88
189	check reference	88
190	check reference	88
191	check reference	88
192	check reference	89
193	check reference	89
194	check reference	90

195	check reference	90
196	check reference	90
197	check reference	90
198	check reference	90
199	add reference	90
200	check reference	92
201	check reference	92
202	check reference	92
203	check reference	92
204	check reference	92
205	change “tiny-jubjub” to “pen-jubjub” throughout?	93
206	check reference	94
207	check reference	94
208	check reference	95
209	either expand on this or delete it	95
210	add reference	95
211	check reference	95
212	check reference	95
213	check reference	95
214	check reference	95
215	check reference	95
216	check reference	96
217	check reference	96
218	check reference	97
219	check reference	97
220	add reference	97
221	add reference	97
222	This needs to be written (in Algebra)	97
223	add reference	97
224	add reference	97
225	check reference	97
226	towers of curve extensions	98
227	check reference	98
228	check reference	98
229	check reference	98
230	check reference	99
231	add reference	99
232	is “huge” a technical term?	99
233	check reference	100
234	S: either add more explanation or move to a footnote	100
235	type 3 pairing-based cryptography	100
236	add references?	100
237	check reference	101
238	check reference	101
239	check floating of algorithm	102
240	add references	102
241	check reference	103
242	add reference	103




















243	check reference	103
244	check reference	103
245	add reference	104
246	should all lines of all algorithms be numbered?	104
247	check reference	105
248	check reference	105
249	check reference	105
250	check if the algorithm is floated properly	105
251	check reference	105
252	again?	107
253	check reference	107
254	circuit	107
255	signature schemes	107
256	this was called “pen-jubjub”.	107
257	check reference	107
258	add reference	108
259	check reference	108
260	add references	108
261	add reference	108
262	reference text to be written in Algebra	108
263	check reference	108
264	check reference	108
265	check reference	109
266	add reference	109
267	algebraic closures	109
268	check reference	109
269	check reference	110
270	check reference	110
271	check reference	110
272	check reference	111
273	disambiguate	111
274	add reference	111
275	unify terminology	111
276	check reference	112
277	actually make this a table?	112
278	exercise still to be written?	113
279	add reference	113
280	check reference	113
281	check reference	113
282	add reference	114
283	check reference	115
284	check reference	115
285	check reference	115
286	add reference	116
287	check reference	116
288	check reference	116
289	check reference	117
290	what does this mean?	118

291	write up this part	119
292	add reference	119
293	check reference	119
294	cyclotomic polynomial	119
295	Pholaard-rho attack	119
296	todo	119
297	why?	120
298	check reference	120
299	check reference	120
300	what does this mean?	120
301	add reference	120
302	add reference	120
303	check reference	120
304	check reference	121
305	add reference	122
306	add exercise	122
307	check reference	123
308	add reference	123
309	add reference	123
310	add reference	123
311	check reference	124
312	check reference	124
313	add reference	124
314	add reference	124
315	add reference	125
316	check reference	125
317	add reference	125
318	add reference	125
319	finish writing this up	126
320	add reference	126
321	correct computations	126
322	fill in missing parts	126
323	add reference	127
324	check equation	127
325	Chapter 1?	128
326	"rigorous"?	128
327	"proving"?	128
328	Add example	129
329	Add more explanation	129
330	I'd delete this, too distracting	129
331	binary tuples	129
332	add reference	130
333	add reference	130
334	check reference	130
335	check reference	130
336	Are we using w and x interchangeably or is there a difference between them?	131
337	check reference	131
338	jubjub	131

339	check reference	131
340	check reference	131
341	check wording	131
342	check reference	131
343	check references	132
344	add reference	132
345	add reference	132
346	check reference	133
347	add reference	133
348	check reference	134
349	check reference	134
350	add reference	135
351	add reference	136
352	Schur/Hadamard product	136
353	add reference	136
354	check reference	136
355	check reference	137
356	add reference	138
357	check reference	139
358	check reference	139
359	check reference	139
360	check reference	139
361	check reference	140
362	add reference	140
363	add reference	141
364	check reference	141
365	check reference	142
366	check reference	142
367	check reference	142
368	add reference	143
369	check reference	145
370	add reference	145
371	check reference	146
372	check reference	146
373	check reference	146
374	Should we refer to R1CS satisfiability (p. 139 here?	147
375	check reference	148
376	add reference	148
377	check reference	148
378	check reference	149
379	check reference	149
380	check reference	150
381	check reference	152
382	add reference	153
383	"by"?	153
384	check reference	153
385	check reference	153
386	add reference	153

387	 add reference	153
388	 check reference	153
389	 add reference	153
390	 clarify language	155
391	 check reference	156
392	 add reference	156
393	 check reference	156
394	 add reference	156
395	 check references	159
396	 add references to these languages?	159
397	 check reference	162
398	 check reference	163
399	 check reference	163
400	 check reference	164
401	 check reference	165
402	 check reference	165
403	 check reference	167
404	 check reference	167
405	 check reference	168
406	 add reference	168
407	 check reference	168
408	 add reference	168
409	 add reference	168
410	 check reference	169
411	 check reference	169
412	 check reference	169
413	 check reference	169
414	 add reference	169
415	 check reference	170
416	 check reference	171
417	 "constraints" or "constrained"?	171
418	 check reference	172
419	 "constraints" or "constrained"?	172
420	 add reference	172
421	 "constraints" or "constrained"?	172
422	 add reference	173
423	 check references	173
424	 check reference	173
425	 add reference	174
426	 can we rotate this by 90°?	174
427	 check reference	175
428	 add reference	175
429	 add reference	175
430	 shift	177
431	 bishift	178
432	 add reference	179
433	 check reference	180
434	 add reference	182

435	add reference	183
436	check reference	184
437	add reference	184
438	add reference	184
439	check reference	185
440	add reference	185
441	add reference	185
442	add reference	186
443	add reference	187
444	add reference	188
445	add reference	188
446	add reference	188
447	add reference	189
448	add reference	189
449	add reference	189
450	add reference	189
451	add reference	189
452	"invariable"?	189
453	add reference	190
454	add reference	190
455	add reference	190
456	add reference	191
457	add reference	191
458	add reference	192
459	add reference	193
460	add reference	193
461	add reference	194
462	add reference	194
463	add reference	194
464	add reference	194
465	add reference	194
466	add reference	195
467	add reference	195
468	add reference	195
469	add reference	195
470	add reference	195
471	add reference	195
472	add reference	195
473	add reference	195
474	add reference	195
475	add reference	196
476	add reference	196
477	add reference	196
478	add reference	196
479	add reference	198
480	add reference	198
481	add reference	198
482	add reference	198

483		add reference	198
484		add reference	198
485		add reference	199
486		add reference	199
487		add reference	199
488		add reference	199
489		add reference	199
490		add reference	200
491		add reference	200
492		add reference	200
493		add reference	200
494		add reference	201
495		add reference	201
496		add reference	201
497		add reference	201
498		add reference	201
499		add reference	201
500		add reference	201
501		add reference	201

502

MoonMath manual

503

TechnoBob and the Least Scruples crew

504

March 27, 2022

Contents

506	1	Introduction	5
507	1.1	Target audience	5
508	1.2	The Zoo of Zero-Knowledge Proofs	6
509		To Do List	8
510		Points to cover while writing	8
511	2	Preliminaries	9
512	2.1	Preface and Acknowledgements	9
513	2.2	Purpose of the book	9
514	2.3	How to read this book	10
515	2.4	Cryptological Systems	10
516	2.5	SNARKS	10
517	2.6	complexity theory	10
518	2.6.1	Runtime complexity	10
519	2.7	Software Used in This Book	11
520	2.7.1	Sagemath	11
521	3	Arithmetics	12
522	3.1	Introduction	12
523	3.1.1	Aims and target audience	12
524	3.1.2	The structure of this chapter	13
525	3.2	Integer Arithmetics	13
526		Euclidean Division	16
527		The Extended Euclidean Algorithm	18
528	3.3	Modular arithmetic	19
529		Congurency	20
530		Modular Arithmetics	20
531		The Chinese Remainder Theorem	23
532		Modular Inverses	26
533	3.4	Polynomial Arithmetics	29
534		Polynomial Arithmetics	33
535		Euklidean Division	34
536		Prime Factors	36
537		Lange interpolation	37
538	4	Algebra	40
539	4.1	Groups	40
540		Commutative Groups	41
541		Finite groups	43

542		Generators	43
543		The discrete Logarithm problem	43
544	4.1.1	Cryptographic Groups	44
545		The discrete logarithm assumption	45
546		The decisional Diffie–Hellman assumption	47
547		The computational Diffie–Hellman assumption	47
548		Cofactor Clearing	48
549	4.1.2	Hashing to Groups	48
550		Hash functions	48
551		Hashing to cyclic groups	50
552		Hashing to modular arithmetics	51
553		Pedersen Hashes	55
554		MimC Hashes	55
555		Pseudorandom Functions in DDH-A groups	55
556	4.2	Commutative Rings	55
557		Hashing to Commutative Rings	58
558	4.3	Fields	58
559		Prime fields	60
560		Square Roots	61
561		Exponentiation	63
562		Hashing into prime fields	63
563		Extension Fields	63
564		Hashing into extension fields	67
565	4.4	Projective Planes	67
566	5	Elliptic Curves	70
567	5.1	Elliptic Curve Arithmetics	70
568	5.1.1	Short Weierstraß Curves	70
569		Affine short Weierstraß form	71
570		Affine compressed representation	75
571		Affine group law	76
572		Scalar multiplication	81
573		Projective short Weierstraß form	84
574		Projective Group law	85
575		Coordinate Transformations	86
576	5.1.2	Montgomery Curves	86
577		Affine Montgomery Form	88
578		Affine Montgomery coordinate transformation	89
579		Montgomery group law	91
580	5.1.3	Twisted Edwards Curves	92
581		Twisted Edwards Form	92
582		Twisted Edwards group law	94
583	5.2	Elliptic Curve Pairings	95
584		Embedding Degrees	95
585		Elliptic Curves over extension fields	96
586		Full torsion groups	97
587		Torsion subgroups	100
588		The Weil pairing	102

589	5.3	Hashing to Curves	104
590		Try-and-increment hash functions	105
591	5.4	Constructing elliptic curves	107
592		The Trace of Frobenius	108
593		The j -invariant	109
594		The Complex Multiplication Method	110
595		The <i>BLS6_6</i> pen-and-paper curve	119
596		Hashing to pairing groups	126
597	6	Statements	128
598	6.1	Formal Languages	128
599		Decision Functions	129
600		Instance and Witness	132
601		Modularity	135
602	6.2	Statement Representations	135
603	6.2.1	Rank-1 Quadratic Constraint Systems	135
604		R1CS representation	136
605		R1CS Satisfiability	138
606		Modularity	140
607	6.2.2	Algebraic Circuits	140
608		Algebraic circuit representation	140
609		Circuit Execution	145
610		Circuit Satisfiability	147
611		Associated Constraint Systems	148
612	6.2.3	Quadratic Arithmetic Programs	153
613		QAP representation	153
614		QAP Satisfiability	155
615	7	Circuit Compilers	159
616	7.1	A Pen-and-Paper Language	159
617	7.1.1	The Grammar	159
618	7.1.2	The Execution Phases	161
619		The Setup Phase	161
620		The Prover Phase	163
621	7.2	Common Programing concepts	163
622	7.2.1	Primitive Types	163
623		The base-field type	164
624		The Subtraction Constraint System	167
625		The Inversion Constraint System	168
626		The Division Constraint System	169
627		The boolean Type	170
628		The boolean Constraint System	170
629		The AND operator constraint system	171
630		The OR operator constraint system	171
631		The NOT operator constraint system	172
632		Modularity	173
633		Arrays	176
634		The Unsigned Integer Type	176

635		The uN Constraint System	177
636		The Unsigned Integer Operators	178
637	7.2.2	Control Flow	179
638		The Conditional Assignment	179
639		Loops	181
640	7.2.3	Binary Field Representations	182
641	7.2.4	Cryptographic Primitives	184
642		Twisted Edwards curves	184
643		Twisted Edwards curve constraints	184
644		Twisted Edwards curve addition	185
645	8	Zero Knowledge Protocols	186
646	8.1	Proof Systems	186
647	8.2	The “Groth16” Protocol	187
648		The Setup Phase	189
649		The Proofer Phase	194
650		The Verification Phase	197
651		Proof Simulation	199
652	9	Exercises and Solutions	202

Chapter 7

Circuit Compilers

As we have seen in the previous chapter, statements can be formalized as membership or knowledge claims in formal language, and algebraic circuits as well as rank-1 constraint systems are two practically important ways to define those languages.

However, both algebraic circuits and rank-1 constraint systems are not ideal from a developers point of view, because they deviate substantially from common programming paradigms. Writing real-world applications as circuits and the associated verification in terms of rank-1 constraint systems is at least as troublesome as writing any other low-level language like assembler code. To allow for complex statement design, it is therefore necessary to have some kind of compiler framework, capable of transforming high-level languages into arithmetic circuits and associated rank-1 constraint systems.

As we have seen in chapter 6 and in 6.2.1.1., both arithmetic circuits and rank-1 constraint systems have a modularity property by which it is possible to synthesize complex circuits from simple ones. A basic approach taken by many circuit/R1CS compilers is therefore to provide a library of atomic and simple circuits and then define a way to combine those basic building blocks into arbitrary complex systems.

In this chapter, we provide an introduction to basic concepts of so-called **circuit compilers** and derive a toy language which we can “compile” in a pen-and-paper approach into algebraic circuits and their associated rank-1 constraint systems.

We start with a general introduction to our language, and then introduce atomic types like booleans and unsigned integers. Then we define the fundamental control flow primitives like the if-then-else conditional and the bounded loop. We will look at basic functionality primitives like elliptic curve cryptography. Primitives like these are often called **gadgets** in the literature.

7.1 A Pen-and-Paper Language

To explain basic concepts of circuit compilers and their associated high-level languages, we derive an informal toy language and associated “brain-compiler” which we name PAPER (**Pen-And-Paper Execution Rules**). PAPER allows programmers to define statements in Rust-like pseudo-code. The language is inspired by ZOKRATES and `circom`.

7.1.1 The Grammar

In PAPER, any statement is defined as an ordered list of functions, where any function has to be declared in the list before it is called in another function of that list. The last entry in a statement has to be a special function, called `main`. Functions take a list of typed parameters as inputs

check
references

add refer-
ences to
these lan-
guages?

and compute a tuple of typed variables as output, where types are special functions that define how to transform that type into another type, ultimately transforming any type into elements of the base field where the circuit is defined over.

Any statement is parameterized over the field that the circuit will be defined on, and has additional optional parameters of unsigned type, needed to define the size of array or the counter of bounded loops. The following definition makes the grammar of a statement precise using a command line language like description:

```
statement <Name> {F:<Field> [ , <N_1: unsigned>, ... ] } {
  [fn <Name>([[pub]<Arg>:<Type>, ...]) -> (<Type>, ...)] {
    [let [pub] <Var>:<Type> ; ... ]
    [let const <Const>:<Type>=<Value> ; ... ]
    Var<==>(fn ([<Arg>|<Const>|<Var>, ...]) | (<Arg>|<Const>|<Var>)) ;
    return (<Var>, ...) ;
  } ; ...]
  fn main([[pub]<Arg>:<Type>, ...]) -> (<Type>, ...) {
    [let [pub] <Var>:<Type> ; ... ]
    [let const <Const>:<Type>=<Value> ; ... ]
    Var<==>(fn ([<Arg>|<Const>|<Var>, ...]) | (<Arg>|<Const>|<Var>)) ;
    return (<Var>, ...) ;
  } ;
}
```

Function arguments and variables are private by default, but can be declared as public by the `pub` specifier. Declaring arguments and variables as public always overwrites any previous or conflicting private declarations. Every argument, constant or variable has a type, and every type is defined as a function that transforms that type into another type:

```
type <TYPE>( t1 : <TYPE_1>) -> TYPE_2{
  let t2: TYPE_2 <== fn(TYPE_1)
  return t2
}
```

Many real-world circuit languages are based on a similar, but of course more sophisticated approach than PAPER. The purpose of PAPER is to show basic principles of circuit compilers and their associated high-level languages.

Example 124. To get a better understanding of the grammar of PAPER, the following constitutes proper high-level code that follows the grammar of the PAPER language, assuming that all types in that code have been defined elsewhere.

```
statement MOCK_CODE {F: F_43, N_1 = 1024, N_2 = 8} {
  fn foo(in_1 : F, pub in_2 : TYPE_2) -> F {
    let const c_1 : F = 0 ;
    let const c_2 : TYPE_2 = SOME_VALUE ;
    let pub out_1 : F ;
    out_1<== c_1 ;
    return out_1 ;
  } ;

  fn bar(pub in_1 : F) -> F {
    let out_1 : F ;
    out_1<==foo(in_1);
    return out_1 ;
  } ;
}
```

```

4884   } ;
4885
4886   fn main(in_1 : TYPE_1) -> (F, TYPE_2) {
4887       let const c_1 : TYPE_1 = SOME_VALUE ;
4888       let const c_2 : F = 2;
4889       let const c_3 : TYPE_2 = SOME_VALUE ;
4890       let pub out_1 : F ;
4891       let out_2 : TYPE_2 ;
4892       c_1 <== in_1 ;
4893       out_1 <== foo(c_2) ;
4894       out_2 <== TYPE_2 ;
4895       return (out_1, out_2) ;
4896   } ;
4897 }

```

7.1.2 The Execution Phases

In contrast to normal executable programs, programs for circuit compilers have two modes of execution. The first mode, usually called the **setup phase**, is executed in order to generate the circuit and its associated rank-1 constraint system, the latter of which is then usually used as input to some zero-knowledge proof system.

The second mode of execution is usually called the **prover phase**. In this phase, a prover usually computes a valid assignment to the circuit. Depending on the use case, this valid assignment is then either directly used as constructive proof for proper circuit execution or is transferred as input to the proof generation algorithm of some zero-knowledge proof system, where the full-sized, non hiding constructive proof is processed into a succinct proof with various levels of zero knowledge.

Modern circuit languages and their associated compilers abstract over those two phases and provide a unified **interphase** to the developer, who then writes a single program that can be used in both phases.

To give the reader a clear, conceptual distinction between the two phases, PAPER keeps them separated. Code can be “brain-compiled” during the **setup-phase** in a pen-and-paper approach into visual circuits. Once a circuit is derived, it can be executed in a **prover phase** to generate a valid assignment. The valid assignment is then interpreted as a constructive proof for a knowledge claim in the associated language.

The Setup Phase In PAPER, the task of the setup phase is to compile code in the PAPER language into a visual representation of an algebraic circuit. Deriving the circuit from the code in a pen-and-paper style is what we call **brain-compiling**.

Given some statement description that adheres to the correct grammar, we start circuit development with an empty circuit, compile the main function first and then inductively compile all other functions as they are called during the process.

For every function we compile, we draw a box-node for every argument, every variable and every constant of that function. If the node represents a variable, we label it with that variable’s name, and if it represents a constant, we label it with that constant’s value. We group arguments into a subgraph labeled “inputs” and return values into a subgraph labeled “outputs”. We then group everything into a subgraph and label that subgraph with the function’s name.

After this is done, we have to do a consistency and type check for every occurrence of the

assignment operator `<==`. We have to ensure that the expression on the right side of the operator is well defined and that the types of both side match.

Then we compile the right side of every occurrence of the assignment operator `<==`. If the right side is a constant or variable defined in this function, we draw a dotted line from the box-node that represents the left side of `<==` to the box node that represents the right side of the same operator. If the right side represents an argument of that function we draw a line from the box-node that represents the left side of `<==` to the box node that represents the right side of the same operator.

If the right side of the `<==` operator is a function, we look into our database, find its associated circuit and draw it. If no circuit is associated to that function yet, we repeat the compilation process for that function, drawing edges from the function's argument to its input nodes and from the functions output nodes to the nodes on the right side of `<==`.

During that process, edge labels are drawn according to the rules from 6.2.2.1. If the associated variable represents a private value, we use the *W* label to indicate a witness, and if it represents a public value, we use the *I* label to indicate an instance.

check
reference

Once this is done, we compile all occurring types in a function, by compiling the function of each type. We do this inductively until we reach the type of the base field. Circuits have no notion of types, only of field elements; hence, every type needs to be compiled to the field type in a sequence of compilation steps.

The compilation stops once we have inductively replaced all functions by their circuits. The result is a circuit that contains many unnecessary box nodes. In a final optimization step, all box nodes that are directly linked to each other are collapsed into a single node, and all box nodes that represent the same constants are collapsed into a single node.

Of course, PAPER's brain-compiler is not properly defined in any formal manner. Its purpose is to highlight important steps that real-world compilers undergo in their setup phases.

Example 125 (A trivial Circuit). To give an intuition of how to write and compile circuits in the PAPER language, consider the following statement description:

```
statement trivial_circuit {F:F_13} {
  fn main{F}(in1 : F, pub in2 : F) -> (F,F) {
    let const outc1 : F = 0 ;
    let const incl : F = 7 ;
    let out1 : F ;
    let out2 : F ;
    out1 <== incl;
    out2 <== in1;
    outc1 <== in2;
    return (out1, out2) ;
  }
}
```

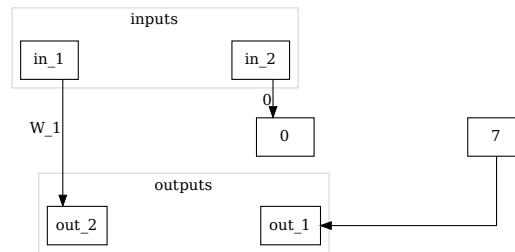
To brain-compile this statement into an algebraic circuit with PAPER, we start with an empty circuit and evaluate function `main`, which is the only function in this statement.

We draw box-nodes for every argument, every constant and every variable of the function and label them with their names or values, respectively. Then we do a consistency and type check for every `<==` operator in the function. Since the circuit only wires inputs to outputs and all elements have the same type, the check is valid.

Then we evaluate the right side of the assignment operators. Since, in our case, the right side of each operator is not a function, we draw edges from the box-nodes on the right side to the associated box node on the left side. To label those edges, we use the general rules of

algebraic circuits as defined in 6.2.2.1. According to those rules, every incoming edge of a sink node has a label and every outgoing edge of a source node has a label, if the node is labeled with a variable. Since nodes that represent constants are implicitly assumed to be private, and since the public specifier determines if an edge is labeled with W or I , we get the following circuit:

check
reference



The Prover Phase In PAPER, a so-called **prover phase** can be executed once the setup phase has generated a circuit image from its associated high-level code. This is done by executing the circuit while assigning proper values to all input nodes of the circuit. However, in contrast to most real-world compilers, PAPER does not tell the prover how to find proper input values to a given circuit. Real-world programming languages usually provide this data by computations that are done outside of the circuit.

Example 126. Consider the circuit from example 125. Valid assignments to this circuit are constructive proofs that the pair of inputs (S_1, S_2) is a point on the tiny-jubjub curve. However, the circuit does not provide a way to actually compute proper values for S_1 and S_2 . Any real-world system therefore needs an auxiliary computation that provides those values.

check
reference

7.2 Common Programing concepts

In this section, we cover concepts that appear in almost every programming language, and see how they can be implemented in circuit compilers.

7.2.1 Primitive Types

Primitive data types like booleans, (unsigned) integers, or strings are the most basic building blocks one can expect to find in every general high-level programming language. In order to write statements as computer programs that compile into circuits, it is therefore necessary to implement primitive types as constraint systems, and define their associated operations as circuits.

In this section, we look at some common ways to achieve this. After a recapitulation of the atomic type of prime field elements, we start with an implementation of the boolean type and its associated boolean algebra as circuits. After that, we define unsigned integers based on the boolean type, and leave the implementation of signed integers as an exercise to the reader.

It should be noted, however, that while primitive data types in common programming languages (like C, Go, or Rust) have a one-to-one correspondence with objects in the computer's memory, this is not the case for most languages that compile into algebraic circuits. As we will see in the following paragraphs, common primitives like booleans or unsigned integers require many constraints and memory. Primitives different from the underlying field elements can be expensive.

The base-field type

Since both algebraic circuits and their associated rank-1 constraint systems are defined over a finite field, elements from that field are the atomic informational units in those models. In this sense, field elements $x \in \mathbb{F}$ are for algebraic circuits what bits are for computers.

In PAPER, we write F for this type and specify the actual field instance for every statement in curly brackets after the name of that statement. Two functions are associated to this type, which are induced by the **addition** and **multiplication** law in the field F . We write

$$\text{MUL} : F \times F \rightarrow F ; (x, y) \mapsto \text{MUL}(x, y) \quad (7.1)$$

$$\text{ADD} : F \times F \rightarrow F ; (x, y) \mapsto \text{ADD}(x, y) \quad (7.2)$$

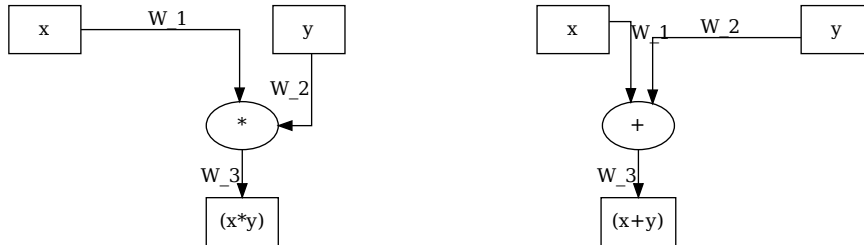
Circuit compilers have to compile these functions into algebraic gates, as explained in 6.2.2.2. Every other function has to be expressed in terms of them and proper wiring.

To represent addition and multiplication in the PAPER language, we define the following two functions:

```
fn MUL(x : F, y : F) -> (MUL(x, y) : F) { }
```

```
fn ADD(x : F, y : F) -> (ADD(x, y) : F) { }
```

The compiler then compiles every occurrence of the MUL or the ADD function into the following circuits:



Example 127 (Basic gates). To give an intuition of how a real-world compiler might transform addition and multiplication in algebraic expressions into a circuit, consider the following PAPER statement:

```
statement basic_ops {F:F_13} {
  fn main(in_1 : F, pub in_2 : F) -> (out_1:F, out_2:F) {
    out_1 <== MUL(in_1, in_2) ;
    out_2 <== ADD(in_1, in_2) ;
  }
}
```

To compile this into an algebraic circuit, we start with an empty circuit and evaluate the function `main`, which is the only function in this statement.

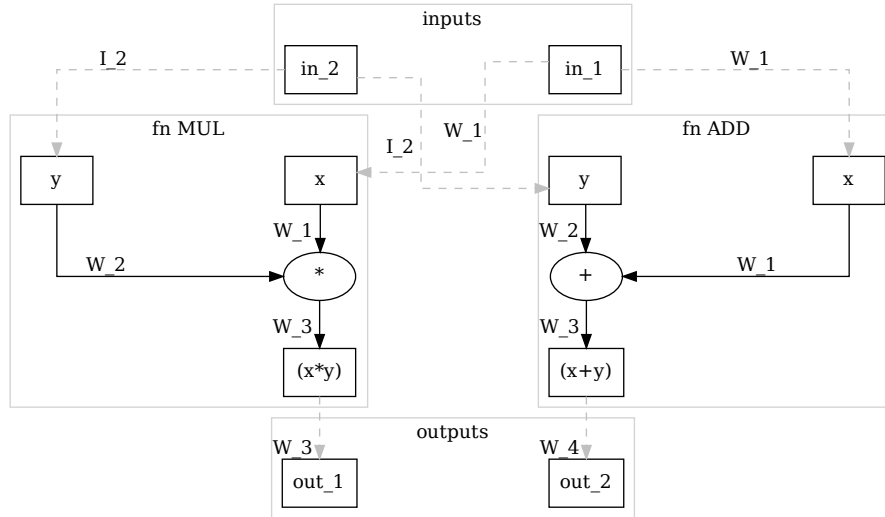
We draw an inputs subgraph containing box-nodes for every argument of the function, and an outputs subgraph containing box-nodes for every factor in the return value. Since all of these nodes represent variables of the `field` type, we don't have to add any type constraints to the circuit.

check
reference

We check the validity of every expression on the right side of every `<==` operator including a type check. In our case, every variable is of the `field` type and hence the types match the types of the `MUL` as well as the `ADD` function and the type of the left sides of `<==` operators.

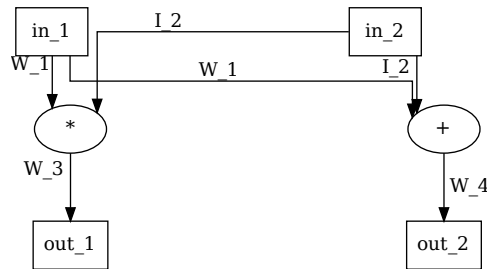
We evaluate the expressions on the right side of every `<==` operator inductively, replacing every occurrence of a function with a subgraph that represents its associated circuit.

According to `PAPER`, every occurrence of the `public` specifier overwrites the associate `private` default value. Using the appropriate edge labels we get:



Any real-world compiler might process its associated high-level language in a similar way, replacing functions, or gadgets by predefined associated circuits. This process is often followed by various optimization steps that try to reduce the number of constraints as much as possible.

In `PAPER`, we optimize this circuit by collapsing all box nodes that are directly connected to other box nodes, adhering to the rule that a variable's `public` specifier overwrites any `private` specifier. Reindexing edge labels, we get the following circuit as our pen and pencil compiler output:



Example 128 (3-factorization). Consider our 3-factorization problem from example 106 and the associated circuit $C_{3.fac_zk}(\mathbb{F}_{13})$ we provided in equation 6.8. To understand the process of replacing high-level functions by their associated circuits inductively, we want define a `PAPER` statement that we brain-compile into an algebraic circuit equivalent to $C_{3.fac_zk}(\mathbb{F}_{13})$:

check
reference

check
reference


```

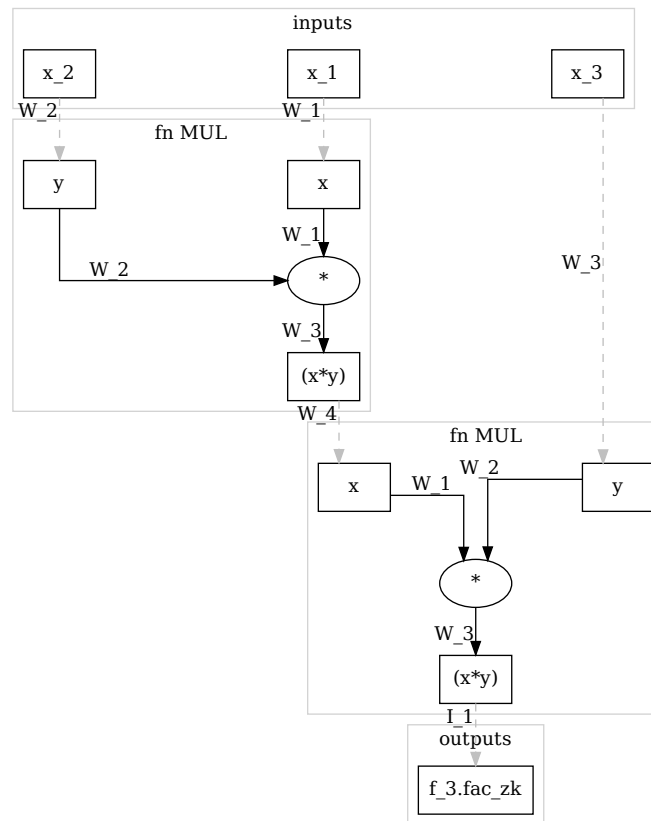
5062 statement 3_fac_zk {F:F_13} {
5063   fn main(x_1 : F, x_2 : F, x_3 : F) -> (pub 3_fac_zk : F){
5064     f_3.fac_zk <== MUL( MUL( x_1 , x_2 ) , x_3 ) ;
5065   }
5066 }

```

Using PAPER, we start with an empty circuit and then add 3 input nodes to the input subgraph as well as 1 output node to the output subgraph. All these nodes are decorated with the associated variable names. Since all of these nodes represent variables of the `field` type, we don't have to add any type constraints to the circuit.

We check the validity of every expression on the right side of the single `<==` operator including a type check.

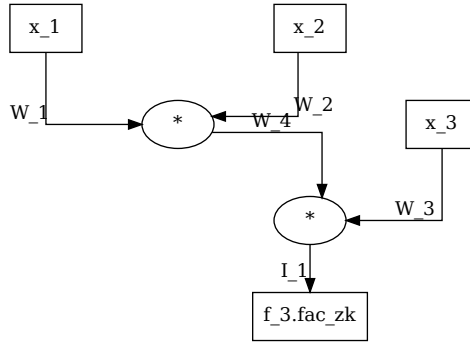
We evaluate the expressions on the right side of every `<==` operator inductively. We have two nested multiplication functions and we replace them by the associated multiplication circuits, starting with the most outer function. We get:



5076

In a final optimization step, we collapse all box nodes directly connected to other box nodes, adhering to the rule that a variables `public` specifier overwrites any `private` specifier. Reindexing edge labels we get the following circuit:

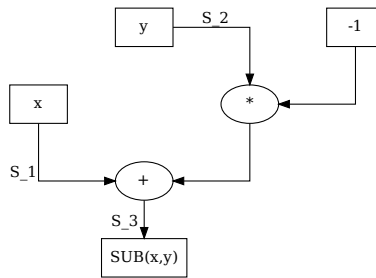
5079



5080

5081 **The Subtraction Constraint System** By definition, algebraic circuits only contain addition
 5082 and multiplication gates, and it follows that there is no single gate for field subtraction, despite
 5083 the fact that subtraction is a native operation in every field.

5084 High-level languages and their associated circuit compilers, therefore, need another way to
 5085 deal with subtraction. To see how this can be achieved, recall that subtraction is defined by addi-
 5086 tion with the additive inverse, and that the inverse can be computed efficiently by multiplication
 5087 with -1 . A circuit for field subtraction is therefore given by



5088

5089 Using the general method from 6.2.1.1, the circuits associated rank-1 constraint system is given
 5090 by:

$$(S_1 + (-1) \cdot S_2) \cdot 1 = S_3 \quad (7.3)$$

5091 Any valid assignment $\{S_1, S_2, S_3\}$ to this circuit therefore enforces the value S_3 to be the differ-
 5092 ence $S_1 - S_2$.

5093 Real-world compilers usually provide a gadget or a function to abstract over this circuit
 5094 such that programers can use subtraction as if it were native to circuits. In PAPER, we define
 5095 the following subtraction function that compiles to the previous circuit:

```
5096 fn SUB(x : F, y : F) -> (SUB(x, y) : F) {
5097   constant c : F = -1 ;
5098   SUB <== ADD(x , MUL( y , c ) );
5099 }
```

5100 In the setup phase of a statement, we compile every occurrence of the SUB function into an
 5101 instance of its associated subtraction circuit, and edge labels are generated according to the
 5102 rules from 6.2.2.1.

check
referencecheck
reference

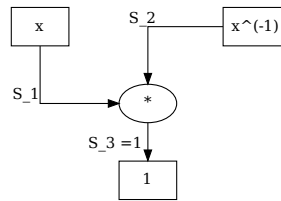
The Inversion Constraint System By definition, algebraic circuits only contain addition and multiplication gates, and it follows that there is no single gate for field inversion, despite the fact that inversion is a native operation in every field.

If the underlying field is a prime field, one approach would be to use Fermat’s little theorem 3.3 to compute the multiplicative inverse inside the circuit. To see how this works, let \mathbb{F}_p be the prime field. The multiplicative inverse x^{-1} of a field element $x \in \mathbb{F}$ with $x \neq 0$ is then given by $x^{-1} = x^{p-2}$, and computing x^{p-2} in the circuit therefore computes the multiplicative inverse.

Unfortunately, real-world primes p are large and computing x^{p-2} by repeated multiplication of x with itself is infeasible. A “double and multiply” approach (as described in XXX) is faster, as it computes the power in roughly $\log_2(p)$ steps, but still adds a lot of constraints to the circuit.

Computing inverses in the circuit makes no use of the fact that inversion is a native operation in any field. A more constraints friendly approach is therefore to compute the multiplicative inverse outside of the circuit and then only enforce correctness of the computation in the circuit.

To understand how this can be achieved, observe that a field element $y \in \mathbb{F}$ is the multiplicative inverse of a field element $x \in \mathbb{F}$ if and only if $x \cdot y = 1$ in \mathbb{F} . We can use this, and define a circuit that has two inputs, x and y , and enforces $x \cdot y = 1$. It is then guaranteed that y is the multiplicative inverse of x . The price we pay is that we can not compute y by circuit execution, but auxiliary data is needed to tell any prover which value of y is needed for a valid circuit assignment. The following circuit defines the constraint



Using the general method from 6.2.1.1, the circuit is transformed into the following rank-1 constraint system:

$$S_1 \cdot S_2 = 1 \quad (7.4)$$

Any valid assignment $\{S_1, S_2\}$ to this circuit enforces that S_2 is the multiplicative inverse of S_1 , and, since there is no field element S_2 such that $0 \cdot S_2 = 1$, it also handles the fact that the multiplicative inverse of 0 is not defined in any field.

Real-world compilers usually provide a gadget or a function to abstract over this circuit, and those functions compute the inverse x^{-1} as part of their witness generation process. Programmers then don’t have to care about providing the inverse as auxiliary data to the circuit. In PAPER, we define the following inversion function that compiles to the previous circuit:

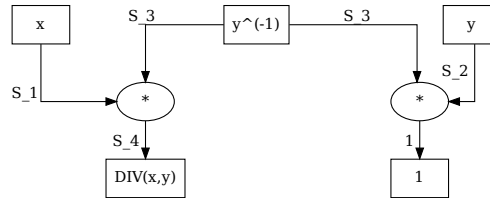
```
fn INV(x : F, y : F) -> (x_inv : F) {
  constant c : F = 1 ;
  c <== MUL( x , y ) ;
  x_inv <== y ;
}
```

As we see, this functions takes two inputs, the field value and its inverse. It therefore does not handle the computation of the inverse by itself. This is to keep PAPER as simple as possible.

In the setup phase, we compile every occurrence of the INV function into an instance of the inversion circuit XXX, and edge labels are generated according to the rules from XXX.

The Division Constraint System By definition, algebraic circuits only contain addition and multiplication gates, and it follows that there is no single gate for field division, despite the fact that division is a native operation in every field.

Implementing division as a circuit, we use the fact that division is multiplication with the multiplicative inverse. We therefore define division as a circuit using the inversion circuit and constraint system from the previous paragraph. Expensive inversion is computed outside of the circuit and then provided as circuit input. We get



Using the method from 6.2.1.1, we transform this circuit into the following rank-1 constraint system:

$$\begin{aligned} S_2 \cdot S_3 &= 1 \\ S_1 \cdot S_3 &= S_4 \end{aligned}$$

Any valid assignment $\{S_1, S_2, S_3, S_4\}$ to this circuit enforces S_4 to be the field division of S_1 by S_2 . It handles the fact that division by 0 is not defined, since there is no valid assignment in case $S_2 = 0$.

In PAPER, we define the following division function that compiles to the previous circuit:

```
fn DIV(x : F, y : F, y_inv : F) -> (DIV : F) {
  DIV <== MUL( x , INV( y, y_inv ) ) ;
}
```

In the setup phase, we compile every occurrence of the binary INV operator into an instance of the inversion circuit.

Exercise 44. Let F be the field \mathbb{F}_5 of modular 5 arithmetics from example 13. Brain-compile the following PAPER statement into an algebraic circuit:

```
statement STUPID_CIRC {F: F_5} {
  fn foo(in_1 : F, in_2 : F)->(out_1 : F, out_2 : F){
    constant c_1 : F = 3 ;
    out_1<== ADD( MUL( c_1 , in_1 ) , in_1 ) ;
    out_2<== INV( c_1 , in_2 ) ;
  } ;

  fn main(in_1 : F, in_2 : F)->(out_1 : F, out_2 : TYPE_2){
    constant (c_1,c_2) : (F,F) = (3,2) ;
    (out_1,out_2) <== foo(in_1, in_2) ;
  } ;
}
```

Exercise 45. Consider the tiny-jubjub curve from example 66 and its associated circuit 125. Write a statement in PAPER that brain-compiles the statement into a circuit equivalent to the one derived in XXX, assuming that curve points are instances and every other assignment is a witness.

5176 *Exercise 46.* Let $F = \mathbb{F}_{13}$ be the modular 13 prime field and $x \in F$ some field element. Define a
 5177 statement in PAPER such that given instance x a field element $y \in F$ is a witness for the statement
 5178 if and only if y is the square root of x .

5179 Brain-compile the statement into a circuit and derive its associated rank-1 constraint system.
 5180 Consider the instance $x = 9$ and compute a constructive proof for the statement.

5181 The boolean Type

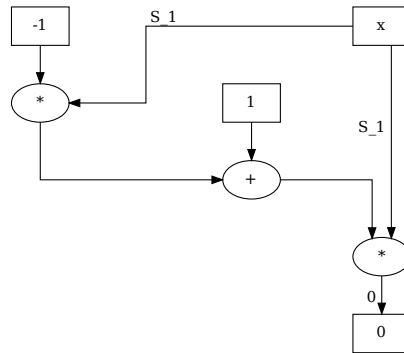
5182 Booleans are a classical primitive type, implemented by virtually every higher programing lan-
 5183 guage. It is therefore important to implement booleans in circuits. One of the most common
 5184 ways to do this is by interpreting the additive and multiplicative neutral element $\{0, 1\} \subset \mathbb{F}$ as
 5185 the two boolean values such that 0 represents *false* and 1 represents *true*. boolean operators
 5186 like *and*, *or*, or *xor* are then expressible as algebraic computations inside \mathbb{F} .

5187 Representing booleans this way is convenient, because the elements 0 and 1 are defined in
 5188 any field. The representation is therefore independent of the actual field in consideration.

5189 To fix boolean algebra notation, we write 0 to represent *false* and 1 to represent *true*, and
 5190 we write \wedge to represent the boolean AND as well as \vee to represent the boolean OR operator.
 5191 The boolean NOT operator is written as \neg .

5192 **The boolean Constraint System** To represent booleans by the additive and multiplicative
 5193 neutral elements of a field, a constraint is required to actually enforce variables of boolean type
 5194 to be either 1 or 0. In fact, many of the following circuits that represent boolean functions are
 5195 only correct under the assumption that their input variables are constrained to be either 0 or 1.
 5196 Not constraining boolean variables is a common problem in circuit design.

5197 In order to constrain an arbitrary field element $x \in \mathbb{F}$ to be 1 or 0, the key observation is
 5198 that the equation $x \cdot (1 - x) = 0$ has only two solutions 0 and 1 in any field. Implementing this
 5199 equation as a circuit therefore generates the correct constraint:



5200

Using the method from 6.2.1.1, we transform this circuit into the following rank-1 constraint system:

$$S_1 \cdot (1 - S_1) = 0$$

check
reference

5201 Any valid assignment $\{S_1\}$ to this circuit enforces S_1 to be either 0 or 1.

5202 Some real-world circuit compilers (like ZOKRATES or BELLMAN) are typed, while others
 5203 (like circom) are not. However, all of them have their way of dealing with the binary con-
 5204 straint. In PAPER, we define the following boolean type that compiles to the previous circuit:

```

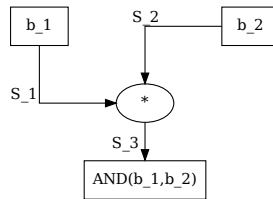
5205 type BOOL(b : BOOL) -> (x : F) {
5206     constant c1 : F = 0 ;
5207     constant c2 : F = 1 ;
5208     constant c3 : F = -1 ;
5209     c1 <== MUL( x , ADD( c2 , MUL( x , c3 ) ) ) ;
5210     x <== b ;
5211 }

```

5212 In the setup phase of a statement, we compile every occurrence of a variable of boolean type
 5213 into an instance of its associated boolean circuit.

5214 **The AND operator constraint system** Given two field elements b_1 and b_2 from \mathbb{F} that are
 5215 constrained to represent boolean variables, we want to find a circuit that computes the logical
 5216 **and** operator $AND(b_1, b_2)$ as well as its associated R1CS that enforces $b_1, b_2, AND(b_1, b_2)$ to
 5217 satisfy the constraint system if and only if $b_1 \wedge b_2 = AND(b_1, b_2)$ holds true.

5218 The key insight here is that, given three boolean constraint variables b_1, b_2 and b_3 , the
 5219 equation $b_1 \cdot b_2 = b_3$ is satisfied in \mathbb{F} if and only if the equation $b_1 \wedge b_2 = b_3$ is satisfied in
 5220 boolean algebra. The logical operator \wedge is therefore implementable in \mathbb{F} by field multiplication
 5221 of its arguments and the following circuit computes the \wedge operator in \mathbb{F} , assuming all inputs are
 5222 restricted to be 0 or 1:



5223

5224 The associated rank-1 constraint system can be deduced from the general process 6.2.1.1 and
 5225 consists of the following constraint:

$$S_1 \cdot S_2 = S_3 \quad (7.5)$$

 check
reference

5226 Common circuit languages typically provide a gadget or a function to abstract over this circuit
 5227 such that programers can use the \wedge operator without caring about the associated circuit. In
 5228 PAPER, we define the following function that compiles to the \wedge -operator's circuit:

```

5229 fn AND(b_1 : BOOL, b_2 : BOOL) -> AND(b_1, b_2) : BOOL{
5230     AND(b_1, b_2) <== MUL( b_1 , b_2 ) ;
5231 }

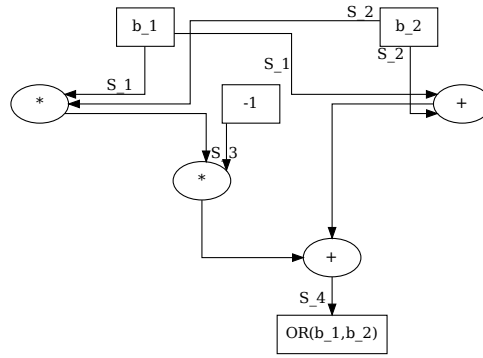
```

5232 In the setup phase of a statement, we compile every occurrence of the AND function into an
 5233 instance of its associated \wedge -operator's circuit.

5234 **The OR operator constraint system** Given two field elements b_1 and b_2 from \mathbb{F} that are
 5235 constrained to represent boolean variables, we want to find a circuit that computes the logical
 5236 **or** operator $OR(b_1, b_2)$ as well as its associated R1CS that enforces $b_1, b_2, OR(b_1, b_2)$ to satisfy
 5237 the constraint system if and only if $b_1 \vee b_2 = OR(b_1, b_2)$ holds true.

5238 Assuming that three variables b_1, b_2 and b_3 are boolean constraint, the equation $b_1 + b_2 - b_1 \cdot$
 5239 $b_2 = b_3$ is satisfied in \mathbb{F} if and only if the equation $b_1 \vee b_2 = b_3$ is satisfied in boolean algebra.
 5240 The logical operator \vee is therefore implementable in \mathbb{F} by the following circuit, assuming all
 5241 inputs are restricted to be 0 or 1:

 "constraints"
or "con-
strained"?



5242

The associated rank-1 constraint system can be deduced from the general process 6.2.1.1 and consists of the following constraints:

check
reference

$$S_1 \cdot S_2 = S_3$$

$$(S_1 + S_2 - S_3) \cdot 1 = S_4$$

Common circuit languages typically provide a gadget or a function to abstract over this circuit such that programers can use the \vee operator without caring about the associated circuit. In PAPER, we define the following function that compiles to the \vee -operator's circuit:

```
5246 fn OR(b_1 : BOOL, b_2 : BOOL) -> OR(b_1,b_2) : BOOL{
5247   constant c1 : F = -1 ;
5248   OR(b_1,b_2) <== ADD(ADD(b_1,b_2),MUL(c1,MUL(b_1,b_2))) ;
5249 }
```

In the setup phase of a statement, we compile every occurrence of the OR function into an instance of its associated \vee -operator's circuit.

Exercise 47. Let \mathbb{F} be a finite field and let b_1 as well as b_2 two boolean constraint variables from \mathbb{F} . Show that the equation $OR(b_1, b_2) = 1 - (1 - b_1) \cdot (1 - b_2)$ holds true.

Use this equation to derive an algebraic circuit with ingoing variables b_1 and b_2 and outgoing variable $OR(b_1, b_2)$ such that b_1 and b_2 are boolean **constraint** and the circuit has a valid assignment, if and only if $OR(b_1, b_2) = b_1 \vee b_2$.

"constraints"
or "con-
strained"?

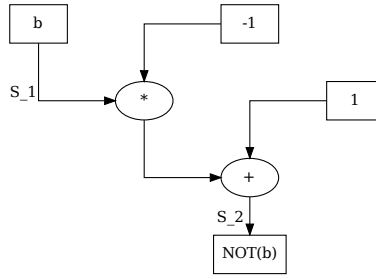
Use the technique from XXX to transform this circuit into a rank-1 constraint system and find its full solution set. Define a PAPER function that brain-compiles into the circuit.

add refer-
ence

The NOT operator constraint system Given a field element b from \mathbb{F} that is constrained to represent a boolean variable, we want to find a circuit that computes the logical **NOT** operator $NOT(b)$ as well as its associated RICS that enforces $b, NOT(b)$ to satisfy the constraint system if and only if $\neg b = NOT(b)$ holds true.

Assuming that two variables b_1 and b_2 are boolean **constraint**, the equation $(1 - b_1) = b_2$ is satisfied in \mathbb{F} if and only if the equation $\neg b_1 = b_2$ is satisfied in boolean algebra. The logical operator \neg is therefore implementable in \mathbb{F} by the following circuit, assuming all inputs are restricted to be 0 or 1:

"constraints"
or "con-
strained"?



5267

The associated rank-1 constraint system can be deduced from the general process XXX and consists of the following constraints

add reference

$$(1 - S_1) \cdot 1 = S_2$$

5268 Common circuit languages typically provide a gadget or a function to abstract over this circuit
 5269 such that programers can use the \neg operator without caring about the associated circuit. In
 5270 PAPER, we define the following function that compiles to the \neg -operator's circuit:

```
5271 fn NOT(b : BOOL -> NOT(b) : BOOL{
5272   constant c1 = 1 ;
5273   constant c2 = -1 ;
5274   NOT(b_1) <== ADD( c1 , MUL( c2 , b ) ) ;
5275 }
```

5276 In the setup phase of a statement, we compile every occurrence of the NOT function into an
 5277 instance of its associated \neg -operator's circuit.

5278 *Exercise 48.* Let \mathbb{F} be a finite field. Derive the algebraic circuit and associated rank-1 constraint
 5279 system for the following operators: NOR, XOR, NAND, EQU.

5280 **Modularity** As we have seen in chapter 6,, both algebraic circuits and R1CS have a modular-
 5281 ity property, and as we have seen in this section, all basic boolean functions are expressible in
 5282 circuits. Combining those two properties, show that it is possible to express arbitrary boolean
 5283 functions as algebraic circuits.

check references

5284 This shows that the expressiveness of algebraic circuits and therefore rank-1 constraint sys-
 5285 tems is as general as the expressiveness of boolean circuits. An important implication is that
 5286 the languages $L_{R1CS-SAT}$ and $L_{Circuit-SAT}$ as defined in 2, are as general as the famous language
 5287 L_{3-SAT} , which is known to be \mathcal{NP} -complete.

check reference

5288 *Example 129.* To give an example of how a compiler might construct complex boolean expres-
 5289 sions in algebraic circuits from simple ones and how we derive their associated rank-1 constraint
 5290 systems, let's look at the following PAPER statement:

```
5291 statement BOOLEAN_STAT {F: F_p} {
5292   fn main(b_1:BOOL,b_2:BOOL,b_3:BOOL,b_4:BOOL )-> pub b_5:BOOL {
5293     b_5 <== AND( OR( b_1 , b_2 ) , AND( b_3 , NOT( b_4 ) ) ) ;
5294   } ;
5295 }
```

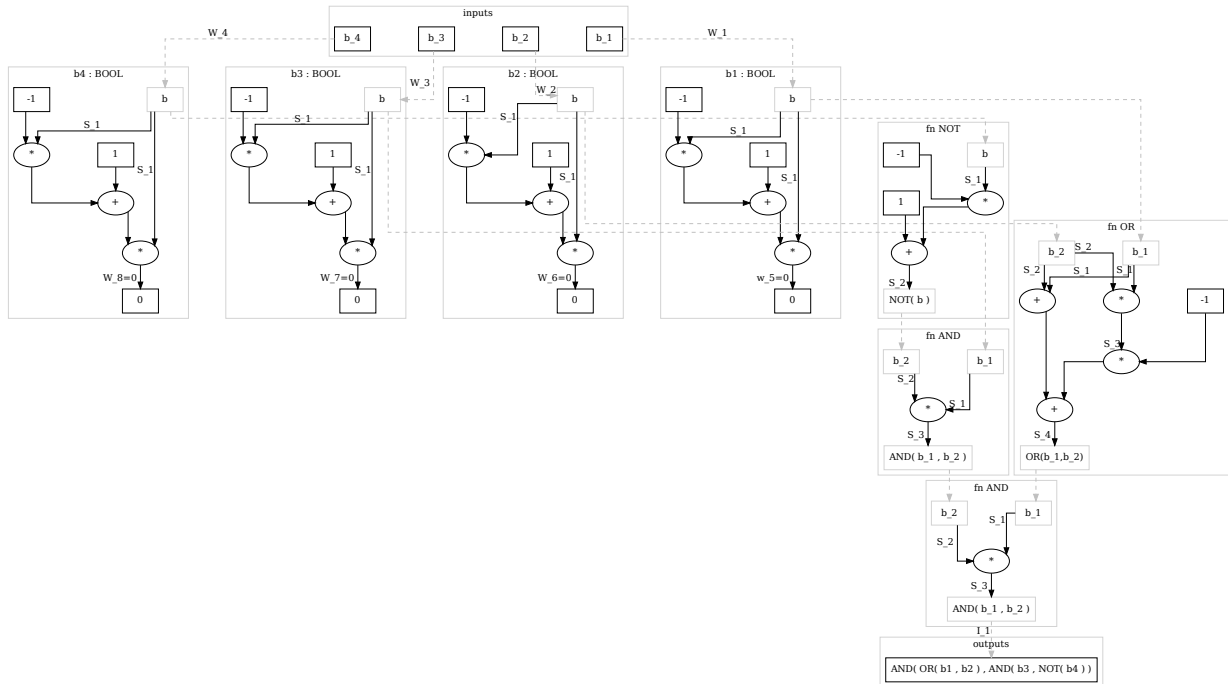
The code describes a circuit that takes four private inputs b_1, b_2, b_3 and b_4 of boolean type and computes a public output b_5 such that the following boolean expression holds true:

$$(b_1 \vee b_2) \wedge (b_3 \wedge \neg b_4) = b_5$$

During a setup-phase, a circuit compiler transforms this high-level language statement into a circuit and associated rank-1 constraint systems and hence defines a language $L_{BOOLEAN_STAT}$.

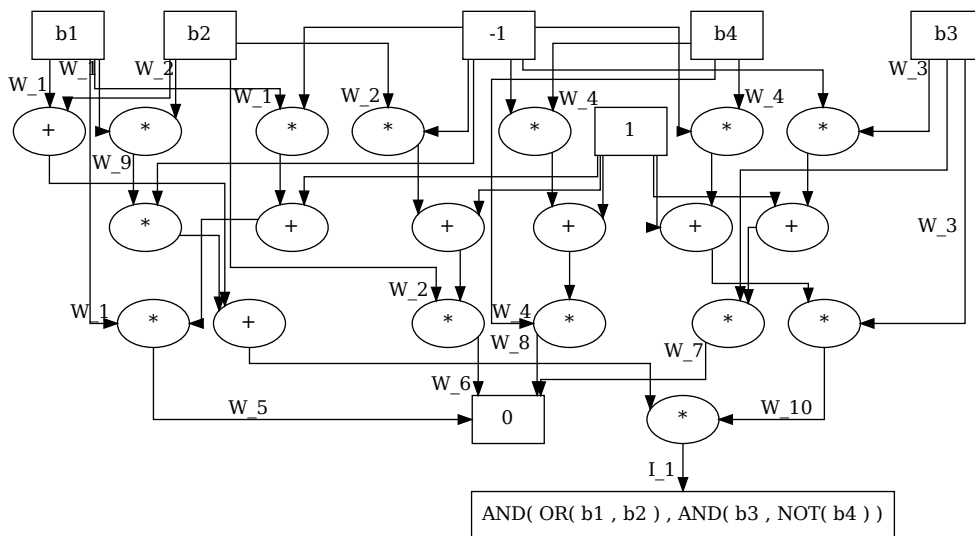
To see how this might be achieved, we use PAPER as an example to execute the setup-phase and compile `BOOLEAN_STAT` into a circuit. Taking the definition of the boolean constraint `XXX` as well as the definitions of the appropriate boolean operators into account, we get the following circuit:

add reference



Simple optimization then collapses all box-nodes that are directly linked and all box nodes that represent the same constants. After relabeling the edges, the following circuit represents the circuit associated to the `BOOLEAN_STAT` statement:

can we rotate this by 90°?



5310 Given some public input I_1 from \mathbb{F}_{13} , a valid assignment to this circuit consists of private inputs
 5311 W_1, W_2, W_3, W_4 from \mathbb{F}_{13} such that the equation $I_1 = (W_1 \vee W_2) \wedge (W_3 \wedge \neg W_4)$ holds true. In
 5312 addition, a valid assignment also has to contain private inputs W_5, W_6, W_7, W_8, W_9 and W_{10} ,
 5313 which can be derived from circuit execution. The inputs W_5, \dots, W_8 ensure that the first four
 5314 private inputs are either 0 or 1 but not any other field element, and the others enforce the boolean
 5315 operations in the expression.

To compute the associated R1CS, we can use the general method from 6.2.1.1 and look at every labeled outgoing edge not coming from a source node. We declare the edges coming from input nodes as well as the edge going to the single output node as public, and every other edge as private input. In this case we get:

check
reference

$$\begin{aligned}
 W_5 : W_1 \cdot (1 - W_1) &= 0 && \text{boolean constraints} \\
 W_6 : W_2 \cdot (1 - W_2) &= 0 \\
 W_7 : W_3 \cdot (1 - W_3) &= 0 \\
 W_8 : W_4 \cdot (1 - W_4) &= 0 \\
 W_9 : W_1 \cdot W_2 &= W_9 && \text{first OR-operator constraint} \\
 W_{10} : W_3 \cdot (1 - W_4) &= W_{10} && \text{AND(.,NOT(.))-operator constraints} \\
 I_1 : (W_1 + W_2 - W_9) \cdot W_{10} &= I_1 && \text{AND-operator constraints}
 \end{aligned}$$

5316 The reason why this R1CS only contains a single constraint for the multiplication gate in the
 5317 OR-circuit, while the general definition XXX requires two constraints, is that the second con-
 5318 straint in XXX only appears because the final addition gate is connected to an output node. In
 5319 this case, however, the final addition gate from the OR-circuit is enforced in the left factor of
 5320 the I_1 constraint. Something similar holds true for the negation circuit.

add refer-
ence

add refer-
ence

During a prover-phase, some public instance I_5 must be given. To compute a constructive proof for the statement of the associated languages with respect to instance I_5 , a prover has to find four boolean values W_1, W_2, W_3 and W_4 such that

$$(W_1 \vee W_2) \wedge (W_3 \wedge \neg W_4) = I_5$$

5321 holds true. In our case neither the circuit nor the PAPER statement specifies how to find those
 5322 values, and it is a problem that any prover has to solve outside of the circuit. This might or
 5323 might not be true for other problems, too. In any case, once the prover found those values, they
 5324 can execute the circuit to find a valid assignment.

To give a concrete example, let $I_1 = 1$ and assume $W_1 = 1, W_2 = 0, W_3 = 1$ and $W_4 = 0$. Since $(1 \vee 0) \wedge (1 \wedge \neg 0) = 1$, those values satisfy the problem and we can use them to execute the circuit. We get

$$\begin{aligned}
 W_5 &= W_1 \cdot (1 - W_1) = 0 \\
 W_6 &= W_2 \cdot (1 - W_2) = 0 \\
 W_7 &= W_3 \cdot (1 - W_3) = 0 \\
 W_8 &= W_4 \cdot (1 - W_4) = 0 \\
 W_9 &= W_1 \cdot W_2 = 0 \\
 W_{10} &= W_3 \cdot (1 - W_4) = 1 \\
 I_1 &= (W_1 + W_2 - W_9) \cdot W_{10} = 1
 \end{aligned}$$

5325 A constructive proof of knowledge of a witness, for instance, $I_1 = 1$, is therefore given by the
 5326 tuple $P = (W_5, W_6, W_7, W_8, W_9, W_{10}) = (0, 0, 0, 0, 0, 1)$.

Arrays

The `array` type represents a fixed-size collection of elements of equal type, each selectable by one or more indices that can be computed at run time during program execution.

Arrays are a classical type, implemented by many higher programming languages that compile to circuits or rank-1 constraint systems. However, most high-level circuit languages support **static** arrays, i.e., arrays whose length is known at compile time only.

The most common way to compile arrays to circuits is to transform any array of a given type τ and size N into N circuit variables of type τ . Arrays are therefore **syntactic sugar**, that is, parts of the formal language that makes the code easier for humans to read, which the compiler transforms into input nodes, much like any other variable. In PAPER, we define the following array type:

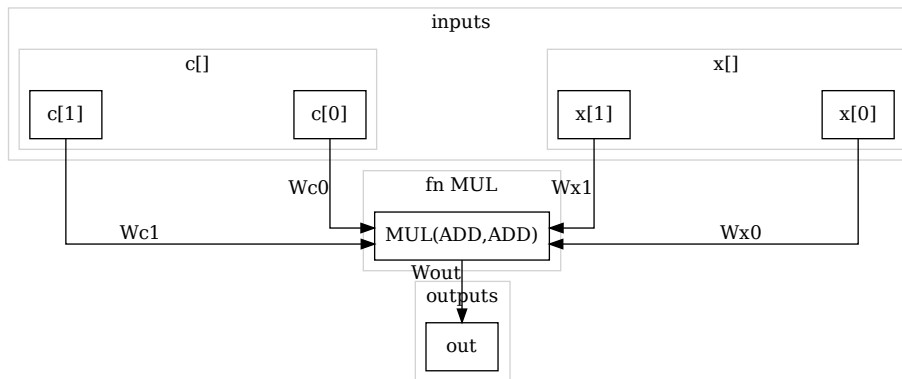
```
type <Name>: <Type>[N : unsigned] -> (Type,...) {
  return (<Name>[0],...)
}
```

In the setup phase of a statement, we compile every occurrence of an array of size N that contains elements of type `Type` into N variables of type `Type`.

Example 130. To give an intuition of how a real-world compiler might transform arrays into circuit variables, consider the following PAPER statement:

```
statement ARRAY_TYPE {F: F_5} {
  fn main(x: F[2]) -> F {
    let constant c: F[2] = [2,4] ;
    let out:F <== MUL(ADD(x[1],c[0]),ADD(x[0],c[1])) ;
    return out ;
  } ;
}
```

During a setup phase, a circuit compiler might then replace any occurrence of the array type by a tuple of variables of the underlying type, and then use those variables in the circuit synthesis process instead. To see how this can be achieved, we use PAPER as an example. Abstracting over the sub-circuit of the computation, we get the following circuit:



The Unsigned Integer Type

Unsigned integers of size N , where N is usually a power of two, represent non-negative integers in the range $0 \dots 2^N - 1$. They have a notion of addition, subtraction and multiplication, defined

5360 by modular 2^N arithmetics. If some N is given, we write uN for the associated type.

5361 **The uN Constraint System** Many high-level circuit languages define the the various uN types
 5362 as arrays of size N , where each element is of boolean type. This is similar to their representa-
 5363 tion on common computer hardware and allows for efficient and straightforward definition of
 5364 common operators, like the various **shift**, or logical operators.

shift

5365 If some unsigned integer N is known at compile time in PAPER, we define the following uN
 5366 type:

```
5367 type uN -> BOOL[N] {
5368   let base2 : BOOL[N] <== BASE_2 (uN) ;
5369   return base2 ;
5370 }
```

To enforce an N -tuple of field elements (b_0, \dots, b_{N-1}) to represent an element of type uN we therefore need N boolean constraints

$$\begin{aligned} S_0 \cdot (1 - S_0) &= 0 \\ S_1 \cdot (1 - S_1) &= 0 \\ &\dots \\ S_{N-1} \cdot (1 - S_{N-1}) &= 0 \end{aligned}$$

5371 In the setup phase of a statement, we compile every occurrence of the uN type by a size N array
 5372 of boolean type. During a=the prover phase, actual elements of the uN type are first transformed
 5373 into binary representation and then this binary representation is assigned to the boolean array
 5374 that represents the uN type.

5375 *Remark 4.* Representing the uN type as boolean arrays is conceptually clean and works over
 5376 generic base fields. However, representing unsigned integers in this way requires a lot of space
 5377 as every bit is represented as a field element and if the base field is large, those field elements
 5378 require considerable space in hardware.

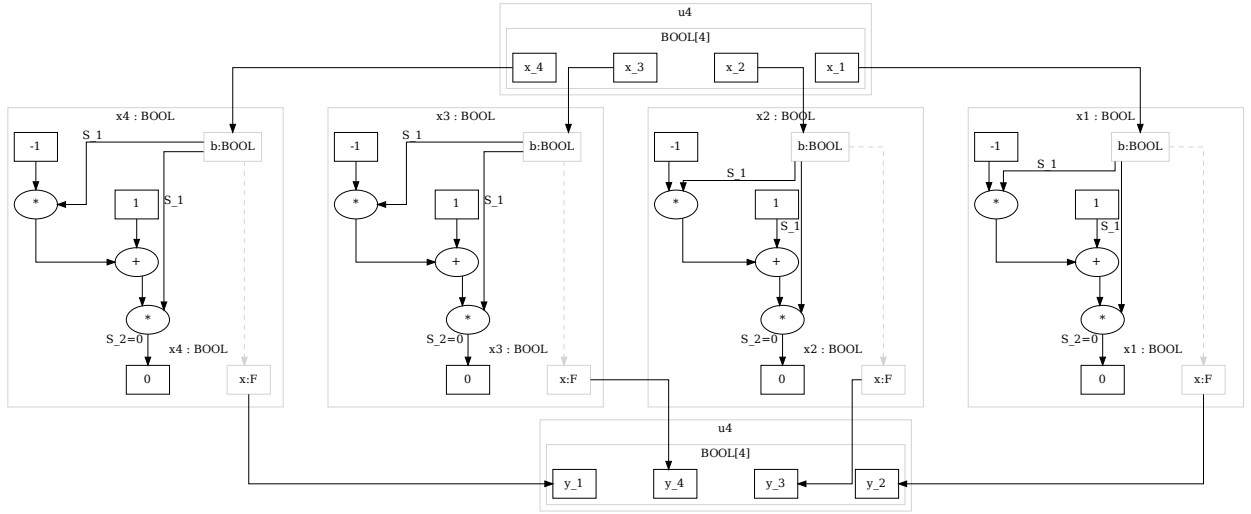
5379 It should be noted that, in some cases, there is another, more space- and constraint-efficient
 5380 approach for representing unsigned integers that can be used whenever the underlying base field
 5381 is sufficiently large. To understand this, recall that addition and multiplication in a prime field
 5382 \mathbb{F}_p is equal to addition and multiplication of integers, as long as the sum or the product does
 5383 not exceed the modulus p . It is therefore possible to represent the uN type inside the base-field
 5384 type whenever N is small enough. In this case, however, care has to be taken to never overflow
 5385 the modulus. It is also important to make sure that, in the case of subtraction, the subtrahend is
 5386 never larger than the minuend.

5387 *Example 131.* To give an intuition of how a real-world compiler might transform unsigned
 5388 integers into circuit variables, consider the following PAPER statement:

```
5389 statement RING_SHIFT{F: F_p, N=4} {
5390   fn main(x: uN)-> uN {
5391     let y:uN <== [x[1],x[2],x[3],x[0]] ;
5392     return y ;
5393   } ;
5394 }
```

5395 During the setup-phase, a circuit compiler might then replace any occurrence of the uN type
 5396 by N variables of `boolean` type. Using the definition of booleans, each of these variables is

then transformed into the `field` type and a boolean constraint system. To see how this can be achieved, we use PAPER as an example and get the following circuit:



During the prover phase, the function `main` is called with an actual input of `u4` type, say $x=14$. The high-level language then has to transform the decimal value 14 into its 4-bit binary representation $14_2 = (0, 1, 1, 1)$ outside of the circuit. Then the array of field values $x[4] = [0, 1, 1, 1]$ is used as an input to the circuit. Since all 4 field elements are either 0 or 1, the four boolean constraints are satisfiable and the output is an array of the four field elements $[1, 1, 1, 0]$, which represents the `u4` element 7.

The Unigned Integer Operators Since elements of `uN` type are represented as boolean arrays, shift operators are implemented in circuits simply by rewiring the boolean input variables to the output variables accordingly.

Logical operators, like AND, OR, or NOT are defined on the `uN` type by invoking the appropriate boolean operators bitwise to every bit in the boolean array that represents the `uN` element.

Addition and multiplication can be represented similarly to how machines represent those operations. Addition can be implemented by first defining the **full adder** circuit and then combining N of these circuits into a circuit that adds two elements from the `uN` type.

Exercise 49. Let $F = \mathbb{F}_{13}$ and $N=4$ be fixed. Define circuits and associated R1CS for the left and right **bishift** operators $x \ll 2$ as well as $x \gg 2$ that operate on the `uN` type. Execute the associated circuit for $x : u4 = 11$.

Exercise 50. Let $F = \mathbb{F}_{13}$ and $N=2$ be fixed. Define a circuit and associated R1CS for the addition operator $\text{ADD} : F \times F \rightarrow F$. Execute the associated circuit to compute $\text{ADD}(2, 7)$.

Exercise 51. Brain-compile the following PAPER code into a circuit and derive the associated R1CS.

```
statement MASK_MERGE {F:F_5, N=4} {
  fn main(pub a : uN, pub b : uN) -> F {
    let constant mask : uN = 10 ;
    let r : uN <== XOR(a, AND(XOR(a, b), mask)) ;
    return r ;
  }
}
```

```
5428     }
5429 }
```

5430 Let L_{mask_merge} be the language defined by the circuit. Provide a constructive knowledge proof
5431 in L_{mask_merge} for the instance $I = (I_a, I_b) = (14, 7)$.

5432 7.2.2 Control Flow

5433 Most programming languages of the imperative or functional style have some notion of basic
5434 control structures to direct the order in which instructions are evaluated. Contemporary circuit
5435 compilers usually provide a single thread of execution and provide basic flow constructs that
5436 implement control flow in circuits.

5437 The Conditional Assignment

5438 Writing high-level code that compiles to circuits, it is often necessary to have a way for condi-
5439 tional assignment of values or computational output to variables.

5440 One way to realize this in many programming languages is in terms of the conditional
5441 ternary assignment operator $?:$ that branches the control flow of a program according to some
5442 condition and then assigns the output of the computed branch to some variable.

```
5443 variable = condition ? value_if_true : value_if_false
```

5444 In this description, `condition` is a boolean expression and `value_if_true` as well as
5445 `value_if_false` are expressions that evaluate to the same type as `variable`.

5446 In programming languages like Rust, another way to write the conditional assignment oper-
5447 ator that is more familiar to many programmers is given by

```
5448 variable = if condition then {
5449     value_if_true
5450 } else {
5451     value_if_false
5452 }
```

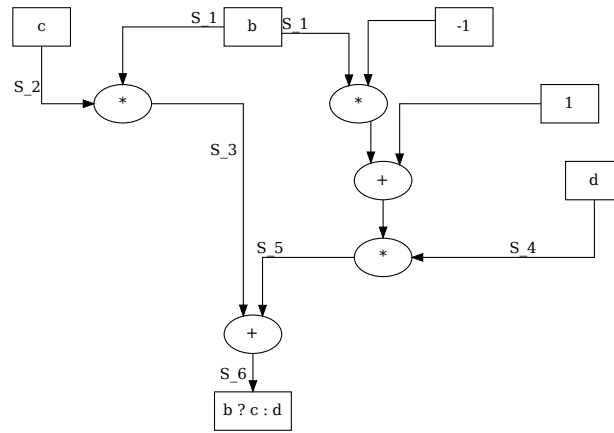
5453 In most programming languages, it is a key property of the ternary assignment operator that
5454 the expression `value_if_true` is only evaluated if `condition` evaluates to true and the
5455 expression `value_if_false` is only evaluated if `condition` evaluates to false. In fact,
5456 computer programs would turn out to be very inefficient if the ternary operator would evaluate
5457 both expressions regardless of the value of `condition`.

5458 A simple way to implement conditional assignment operator as a circuit can be achieved
5459 if the requirement that only one branch of the conditional operator is executed is dropped. To
5460 see that, let b , c and d be field elements such that b is a boolean constraint. In this case, the
5461 following equation enforces a field element x to be the result of the conditional assignment
5462 operator:

$$x = b \cdot c + (1 - b) \cdot d \quad (7.6)$$

5463 Expressing this equation in terms of the addition and multiplication operators from XXX, we
5464 can flatten it into the following algebraic circuit:

add refer-
ence



5465

5466 Note that, in order to compute a valid assignment to this circuit, both S_2 as well as S_4 are
 5467 necessary. If the inputs to the nodes c and d are circuits themselves, both circuits need valid
 5468 assignments and therefore have to be executed. As a consequence, this implementation of
 5469 the conditional assignment operator has to execute all branches of all circuits, which is very
 5470 different from the execution of common computer programs and contributes to the increased
 5471 computational effort any prover has to invest, in contrast to the execution in other programming
 5472 models.

We can use the general technique from 6.2.1.1 to derive the associated rank-1 constraint system of the conditional assignment operator. We get the following:

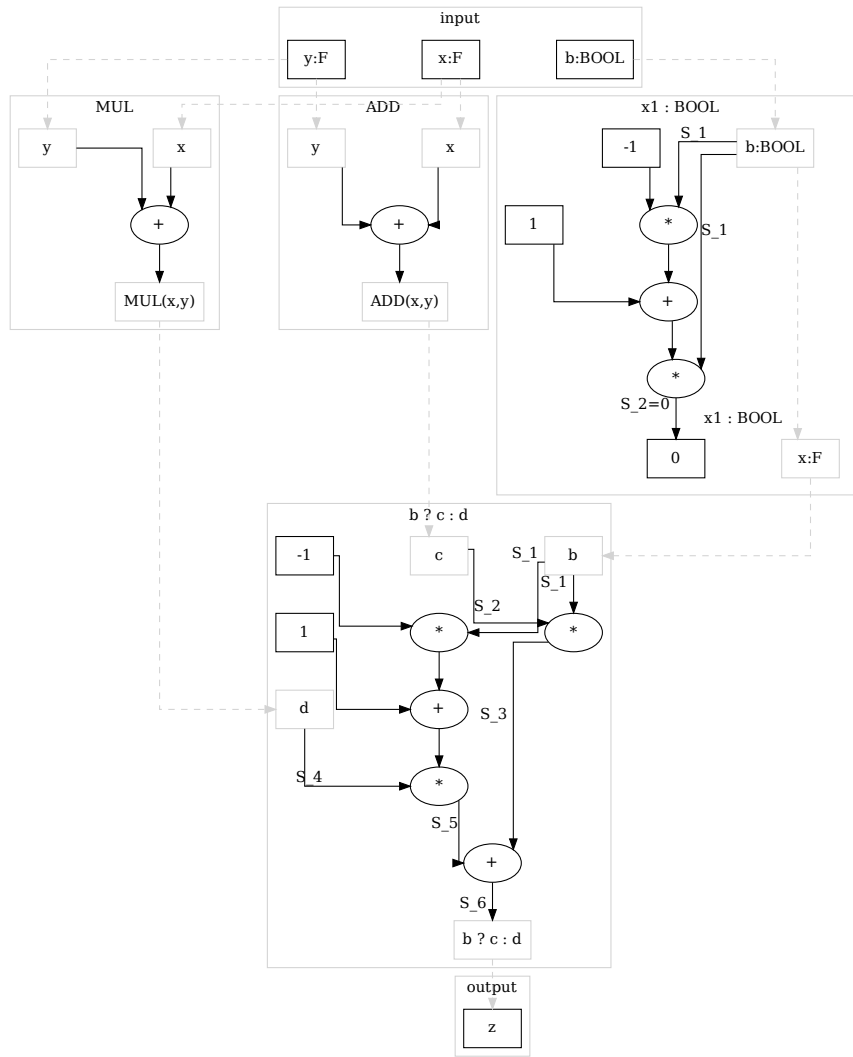
check
reference

$$\begin{aligned} S_1 \cdot S_2 &= S_3 \\ (1 - S_1) \cdot S_4 &= S_5 \\ (S_3 + S_5) \cdot 1 &= S_6 \end{aligned}$$

5473 *Example 132.* To give an intuition of how a real-world circuit compiler might transform any
 5474 high-level description of the conditional assignment operator into a circuit, consider the follow-
 5475 ing PAPER code:

```
5476 statement CONDITIONAL_OP {F:F_p} {
5477   fn main(x : F, y : F, b : BOOL) -> F {
5478     let z : F <== if b then {
5479       ADD(x,y)
5480     } else {
5481       MUL(x,y)
5482     } ;
5483     return z ;
5484   }
5485 }
```

5486 Brain-compiling this code into a circuit, we first draw box nodes for all input and output vari-
 5487 ables, and then transform the boolean type into the field type together with its associated con-
 5488 straint. Then we evaluate the assignments to the output variables. Since the conditional assign-
 5489 ment operator is the top level function, we draw its circuit and then draw the circuits for both
 5490 conditional expressions. We get the following:



5491

5492 **Loops**

5493 In many programming languages, various loop control structures are defined that allow devel-
 5494 opers to execute expressions with a specified number of repetitions or arguments. In particular,
 5495 it is often possible to implement unbounded loops like the loop structure give below, where the
 5496 number of executions depends on execution inputs and is therefore unknown at compile time:

5497 **while** true do { }

5498 In contrast to this, it should be noted that algebraic circuits and rank-1 constraint systems
 5499 are not general enough to express arbitrary computation, but bounded computation only. As a
 5500 consequence, it is not possible to implement unbounded loops, or loops with bounds that are
 5501 unknown at compile time in those models. This can be easily seen since circuits are acyclic
 5502 by definition, and implementing an unbounded loop as an acyclic graph requires a circuits of
 5503 unbounded size.

5504 However, circuits are general enough to express bounded loops, where the upper bound on
 5505 its execution is known at compile time. Those loop can be implemented in circuits by enrolling
 5506 the loop.

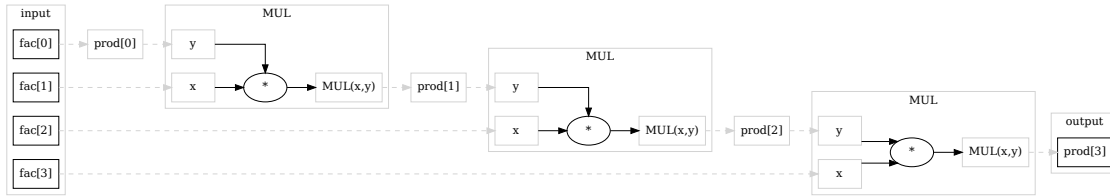
As a consequence, any programming language that compiles to algebraic circuits can only provide loop structures where the bound is a constant known at compile time. This implies that loops cannot depend on execution inputs, but on compile time parameters only.

Example 133. To give an intuition of how a real-world circuit compiler might transform any high-level description of a bounded `for` loop into a circuit, consider the following PAPER code:

```
statement FOR_LOOP {F:F_p, N: unsigned = 4} {
  fn main(fac : F[N]) -> F {
    let prod[N] : F ;
    prod[0] <== fac[0] ;
    for unsigned i in 1..N do [{
      prod[i] <== MUL(fac[i], prod[i-1]) ;
    }]
    return prod[N] ;
  }
}
```

Note that, in a program like this, the loop counter `i` has no expression in the derived circuit. It is pure syntactic sugar, telling the compiler how to unroll the loop.

Brain-compiling this code into a circuit, we first draw box nodes for all input and output variables, noting that the loop counter is not represented in the circuit. Since all variables are of `field` type, we don't have to compile any type constraints. Then we evaluate the assignments to the output variables by unrolling the loop into 3 individual assignment operators. We get:



7.2.3 Binary Field Representations

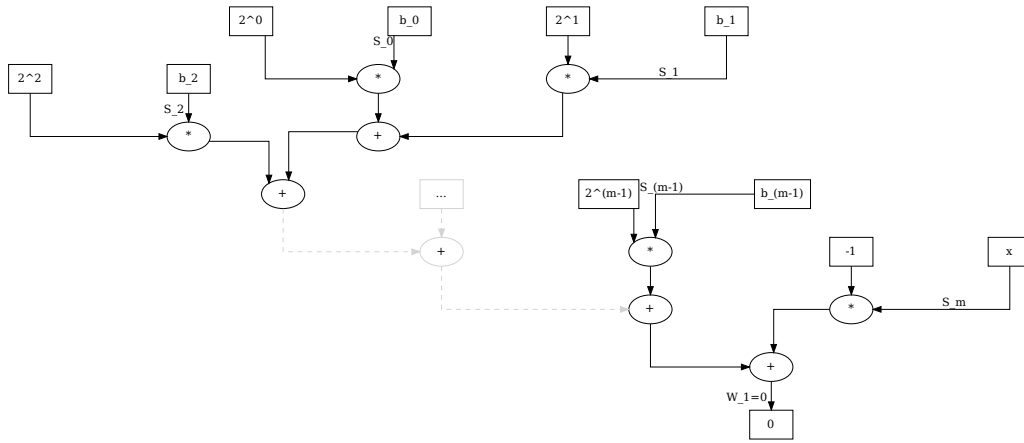
In applications, it is often necessary to enforce a binary representation of elements from the `field` type. To derive an appropriate circuit over a prime field \mathbb{F}_p , let $m = \lceil \log_2 p \rceil$ be the smallest number of bits necessary to represent the prime modulus p . Then a bitstring $(b_0, \dots, b_{m-1}) \in \{0, 1\}^m$ is a binary representation of a field element $x \in \mathbb{F}_p$, if and only if

$$x = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \dots + b_{m-1} \cdot 2^{m-1}$$

In this expression, addition and exponentiation is considered to be executed in \mathbb{F}_p , which is well defined since all terms 2^j for $0 \leq j < m$ are elements of \mathbb{F}_p . Note, however, that in contrast to the binary representation of unsigned integers $n \in \mathbb{N}$, this representation is not unique in general, since the modular p equivalence class might contain more than one binary representative.

Considering that the underlying prime field is fixed and the most significant bit of the prime modulus is m , the following circuit flattens equation XXX, assuming all inputs b_1, \dots, b_m are of boolean type.

add reference



5538

Applying the general transformation rule to compute the associated rank-1 constraint systems, we see that we actually only need a single constraint to enforce some binary representation of any field element. We get

$$(S_0 \cdot 2^0 + S_1 \cdot 2^1 + \dots + S_{m-1} \cdot 2^{m-1} - S_m) \cdot 1 = 0$$

5539 Given an array `BOOL[N]` of N boolean constraint field elements and another field element x ,
 5540 the circuit enforces `BOOL[N]` to be one of the binary representations of x . If `BOOL[N]` is not
 5541 a binary representation of x , no valid assignment and hence no solution to the associated R1CS
 5542 can exist.

5543 *Example 134.* Consider the prime field \mathbb{F}_{13} . To compute binary representations of elements
 5544 from that field, we start with the binary representation of the prime modulus 13, which is $|13|_2 =$
 5545 $(1, 0, 1, 1)$ since $13 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$. So $m = 4$ and we need up to 4 bits to represent
 5546 any element $x \in \mathbb{F}_{13}$.

To see that binary representations are not unique in general, consider the element $2 \in \mathbb{F}_{13}$. It has the binary representations $|2|_2 = (0, 1, 0, 0)$ and $|2|_2 = (1, 1, 1, 1)$, since in \mathbb{F}_{13} we have

$$2 = \begin{cases} 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 \\ 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 \end{cases}$$

5547 This is because the unsigned integers 2 and 15 are both in the modular 13 remainder class of 2
 5548 and hence are both representatives of 2 in \mathbb{F}_{13} .

To see how circuit XXX works, we want to enforce the binary representation of $7 \in \mathbb{F}_{13}$. Since $m = 4$ we have to enforce a 4-bit representation for 7, which is $(1, 1, 1, 0)$, since $7 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3$. A valid circuit assignment is therefore given by $(S_0, S_1, S_2, S_3, S_4) = (1, 1, 1, 0, 7)$ and, indeed, the assignment satisfies the required 5 constraints including the 4 boolean constraints for S_0, \dots, S_3 :

$$\begin{aligned} 1 \cdot (1 - 1) &= 0 & // \text{boolean constraints} \\ 1 \cdot (1 - 1) &= 0 \\ 1 \cdot (1 - 1) &= 0 \\ 0 \cdot (1 - 0) &= 0 \\ (1 + 2 + 4 + 0 - 7) \cdot 1 &= 0 & // \text{binary rep. constraint} \end{aligned}$$

 add refer-
ence

7.2.4 Cryptographic Primitives

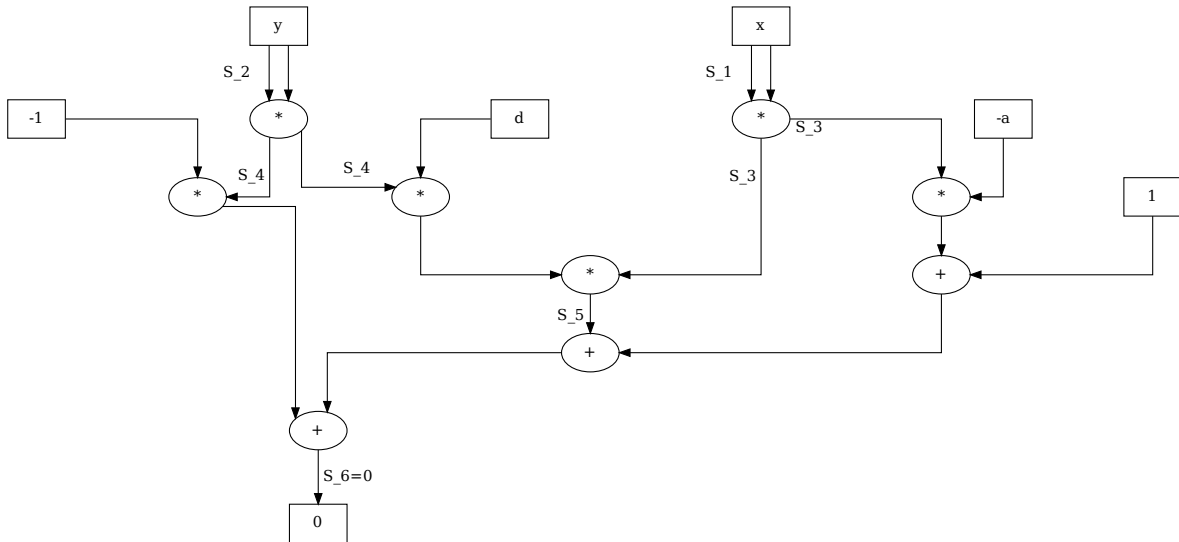
In applications, it is often required to do cryptography in a circuit. To do this, basic cryptographic primitives like hash functions or elliptic curve cryptography needs to be implemented as circuits. In this section, we give a few basic examples of how to implement such primitives.

Twisted Edwards curves

Implementing elliptic curve cryptography in circuits means to implement the field equation as well as the algebraic operations of an elliptic curve as circuits. To do this efficiently, the curve must be defined over the same base field as the field that is used in the circuit.

For efficiency reasons, it is advantageous to choose an elliptic curve such that that all required constraints and operations can be implement with as few gates as possible. Twisted Edwards curves are particularly useful for that matter, since their addition law is particularly simple and the same equation can be used for all curve points including the point at infinity. This simplifies the circuit a lot.

Twisted Edwards curve constraints As we have seen in section 5.1.3, a twisted Edwards curve over a finite field F is defined as the set of all pairs of points $(x, y) \in \mathbb{F} \times \mathbb{F}$ such that x and y satisfy the equation $a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2$. As we have seen in example XXX, we can transform this equation into the following circuit:



The circuit enforces the two inputs of `field` type to satisfy the twisted Edwards curve equation and, as we know from example XXX, the associated rank-1 constraint system is given by:

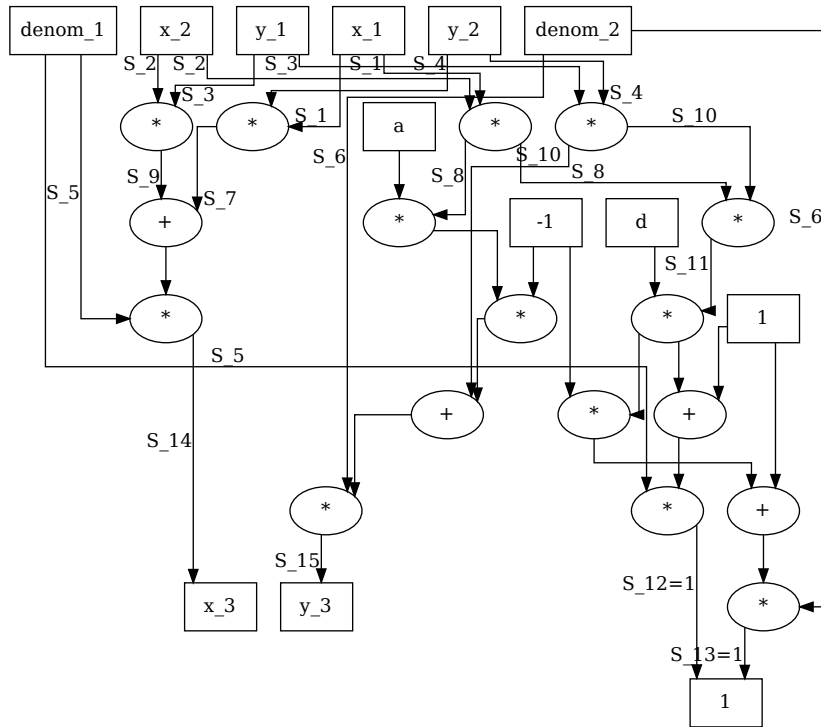
$$\begin{aligned}
 S_1 \cdot S_1 &= S_3 \\
 S_2 \cdot S_2 &= S_4 \\
 (S_4 \cdot 8) \cdot S_3 &= S_5 \\
 (12 \cdot S_4 + S_5 + 10 \cdot S_3 + 1) \cdot 1 &= 0
 \end{aligned}$$

Exercise 52. Write the circuit and associated rank-1 constraint system for a Weierstraß curve of a given field \mathbb{F} .

Twisted Edwards curve addition As we have seen in 5.1.3, a major advantage of twisted Edwards curves is the existence of an addition law that contains no branching and is valid for all curve points. Moreover, the neutral element is not “at infinity” but the actual curve point $(0, 1)$. In fact, given two points (x_1, y_1) and (x_2, y_2) on a twisted Edwards curve, their sum is defined as

$$(x_3, y_3) = \left(\frac{x_1 y_2 + y_1 x_2}{1 + d \cdot x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a \cdot x_1 x_2}{1 - d \cdot x_1 x_2 y_1 y_2} \right)$$

We can use the division circuit from XXX to flatten this equation into an algebraic circuit. Inputs to the circuit are then not only the two curve points (x_1, y_1) and (x_2, y_2) , but also the two denominators $denum_1 = 1 + d \cdot x_1 x_2 y_1 y_2$ as well as $denum_2 = 1 - d \cdot x_1 x_2 y_1 y_2$, which any prover needs to compute outside of the circuit. We get



Using the general technique from XXX to derive the associated rank-1 constraint system, we get the following result:

$$\begin{aligned} S_1 \cdot S_4 &= S_7 \\ S_1 \cdot S_2 &= S_8 \\ S_2 \cdot S_3 &= S_9 \\ S_3 \cdot S_4 &= S_{10} \\ S_8 \cdot S_{10} &= S_{11} \\ S_5 \cdot (1 + d \cdot S_{11}) &= 1 \\ S_6 \cdot (1 - d \cdot S_{11}) &= 1 \\ S_5 \cdot (S_9 + S_7) &= S_{14} \\ S_6 \cdot (S_{10} - a \cdot S_8) &= S_{15} \end{aligned}$$

5576 *Exercise 53.* Let \mathbb{F} be a field. Define a circuit that enforces field inversion for a point of a
5577 twisted Edwards curve over \mathbb{F} .

Bibliography

- Jens Groth. On the size of pairing-based non-interactive arguments. *IACR Cryptol. ePrint Arch.*, 2016:260, 2016. URL <http://eprint.iacr.org/2016/260>.
- P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700.
- David Fifield. The equivalence of the computational diffie–hellman and discrete logarithm problems in certain groups, 2012. URL <https://web.stanford.edu/class/cs259c/finalpapers/dlp-cdh.pdf>.
- Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-46766-3. URL <https://fmouhart.epheme.re/Crypto-1617/TD08.pdf>.
- Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. Cryptology ePrint Archive, Report 2016/492, 2016. <https://ia.cr/2016/492>.