CS-470: Homework Set #2

A Cycle-by-Cycle Scheduler for a VLIW Processor

1 Introduction

In this homework, you will write a scheduler for a simple VLIW processor called VLIW470. In the first homework, you investigated how an out-of-order processor can dynamically extract parallelism. In this homework, you will adopt a completely different approach and rely on compiler algorithms to statically analyze code and extract parallelism. The processor will once again run a minimal subset of the RISC-V instruction set, this time with support for loads and stores, enriched with additional hypothetical support for a subset of the Itanium architectural capabilities. We provide you with a simulator for that processor, reading VLIW code and producing a cycle-by-cycle dump of the architectural data structures. You will be able to use a similar visualizer as in Homework 1 for debugging purposes.

You can use any language you wish to program the compiler, provided that the teaching assistants can run it with reasonable ease and that the inputs and output formats are strictly respected.

2 Processor Architecture

We first present the architecture of the VLIW470 processor. It includes a small subset of the Intel Itanium architectural capabilities—if you want to learn more about the Itanium architecture, feel free to have a look at the first two reference papers [1, 2].

2.1 Instruction Set

Our processor supports the subset of the RISC-V instruction set listed in table 1. With respect to the previous homework, we added a `load` and a `store` instruction, a `nop` instructions, two loop instructions, and four mov instructions. The `loop` instruction, depicted in Figure 1, has a single immediate argument

Mnemonic	Semantic	Latency
`add dest, opA, opB`	dest = opA + opB	1
`addi dest, opA, imm`	dest = opA + (signed) imm	1
`sub dest, opA, opB`	dest = opA − opB	1
`mulu dest, opA, opB`	dest = (unsigned) opA * (unsigned) opB	3
`ld dest, imm(addr)`	dest = **MEM**[addr + (signed)imm]	1
`st source, imm(addr)`	**MEM**[addr + (signed)imm] = source	1
`loop loopStart`	Behaves as in Figure 1. loopStart is an immediate.	1
`loop.pip loopStart`	loop instruction, see section 2.5. loopStart is an immediate.	1
`nop`	Does nothing	1
`mov pX, true/false`	Sets the value of the predicate register pX	1
`mov LC/EC, imm`	Sets the value of special registers with the given immediate	1
`mov dest, imm`	Sets the value of the destination register with the given immediate	1
`mov dest, source`	Sets the value of the destination register from the source register	1

Table 1: Instruction Set.

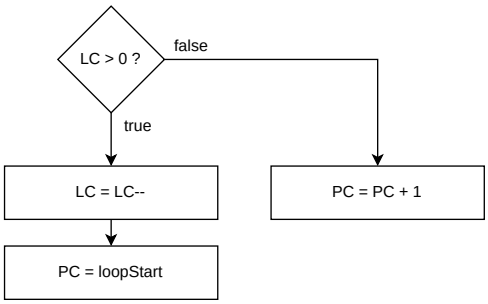


Figure 1: Simple Loop Instruction.

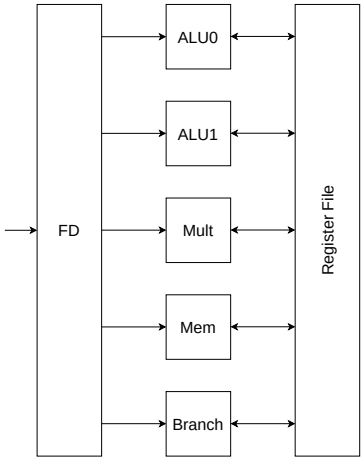


Figure 2: Pipeline Overview.

representing the target of the branch and denoted by `loopStart`. `loop` jumps to `loopStart` if and only if the content of the LC register is greater than 0, where LC is a special register standing for Loop Count and exclusively used to hold loop counter values. `loop.pip` does additional work to enable efficient software pipelining and will be described in much more depth in Section 2.5. Finally, all instructions except the `loop` and `loop.pip` instructions can be prefixed with (pX) to indicate that the instructions dependes on the value of predicate pX. We describe predication in more details in Section 2.3.

As with the previous homework, the input and output of your program will use the JSON format. We give a detailed example of both the input and output syntax in Section 4.

2.2 Processor Pipeline

The pipeline of the VLIW470 processor is depicted in Figure 2. The processor has two ALUs, which can execute any of the arithmetic instructions (`add`, `addi`, `sub`, and `mov`), a multiplication unit, which can execute `mulu` operations, a memory unit, which can execute the `load` and `store` operations, and a branch unit, able to execute either of the `loop` or `loop.pip`. For simplicity, the latency of all operations except `mulu` is 1 cycle, meaning that the results of all but the `mulu` instruction are available on the next cycle. The latency of `mulu` is 3 cycles.

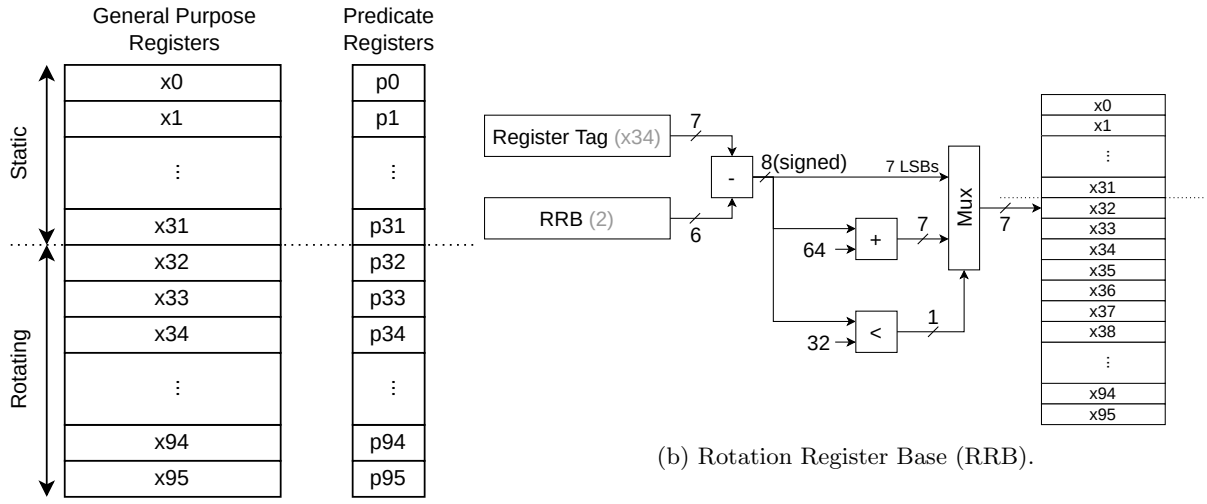
In VLIW processors, the compiler is responsible for finding and exploiting ILP (Instruction Level Parallelism). Therefore, the Fetch and Decode unit fetches bundles of instructions from memory. A bundle is a set of instructions the same size as the number of execution units; each element of the bundle executes on a distinct and predetermined execution unit. We always represent bundles as in Figure 3 with execution units in the depicted order (ALU0, ALU1, Mult, Mem, and Branch). If an execution unit is not used in a particular cycle, a `nop` instruction must occupy the associated bundle slot.

2.3 Register Files

Our processor has 96 general-purpose registers named `x0` to `x95` and 96 predicate registers named `p0` to `p95`, depicted in Figure 4a. Predicates are 1-bit registers holding the result of conditional expression evaluation and can be associated with any instruction. A predicated instruction can modify the state of the processor if and only if its predicate is one. Otherwise, the destination register is untouched, and the commit stage will discard the instruction. Loop instructions cannot be predicated, but any other one can. Please note that predicating `nop` instructions has no effect. While, in this homework, we only use

ALU 0	ALU 1	Mult	Mem	Branch

Figure 3: Bundle.



(a) Register Files.

(b) Rotation Register Base (RRB).

Figure 4: Rotating Register Support.

predication to support efficient software pipelining, it is also a way VLIW processors use to speculatively execute instructions. If you want to learn more, feel free to check the Intel Itanium compiler paper [3].

Similar to the Intel Itanium architecture, the processor provides architectural support for software pipelining through rotating registers. General-purpose registers `x32` to `x95` rotate, as well as predicate registers `p32` to `p95`. Rotation is achieved thanks to the Register Rotation Base (RRB) register and querying the value of any rotating register requires subtracting its content from the register number (see Figure 4b; if the subtraction results in a value below 32, the register selection wraps around, of course). If the value of RRB is changed across loop iterations, we can overlap loop iterations while re-using the exact same code and avoiding resource conflicts. Managing RRB is the job of the instruction `loop.pip` (Section 2.5) and accounting for changing RRB values will impact the register allocation algorithm (Section 3.3).

2.4 Data Memory

The data memory system is not a critical topic in VLIW470. The memory is addressed by word (64-bit), which means accessing it with address `1` and address `2` leads to different word. As a result, you don't have to consider the alignment problem. Meanwhile, since the latency of the load & store unit is 1 cycle, you can simply treat that all memory instructions are atomic and strictly follow the program order, which means their result is immediately available after being committed.

You don't have to consider the memory storing instructions.

2.5 Architectural Support for Loops

Generally, the vast majority of the execution time of programs is spent iterating over some data using loops. However, the instructions in a loop body generally depend on each other and may not expose enough parallelism to fully utilize the processor resources. A solution to this problem is software pipelining, where various loop iterations are executed concurrently. While conceptually simple, software pipelining involves fairly convoluted program transformations and hardware data structures. We describe in this section the hardware support provided by the `loop.pip` instruction.

VLIW470, similarly to Itanium, has several resources to simplify loop optimization through software pipelining. These resources are manages by `loop.pip` as follows:

1. The first is a way to maintain the loop count and thus decide if a new iteration should start. This state is stored in a special register called LC (Loop Count) and is initialized by the software before the loop with the `mov` instruction. Instructions `loop` and `loop.pip` take care of decrementing LC

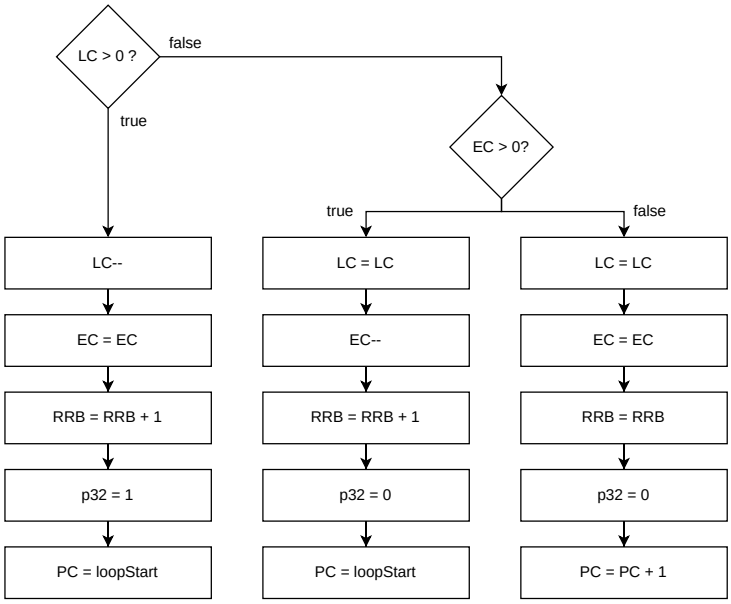


Figure 5: `loop.pip` Instruction.

and either branching to the start of the loop, when new iterations start, or continue the execution past the loop, when this is over.

2. The hardware renames registers on the fly to guarantee that the identical register names used in consecutive loop iterations do not conflict. The renaming is performed thanks to the RRB (Rotating Register Base) register, incremented when `loop.pip` is executed.

3. Whenever the loop starts, the appropriate operations of the iteration must be progressively enabled. To this end, each instruction within the loop body must be associated with a predicate: the stage predicate. The `loop.pip` instruction automatically enables or disable stage predicates depending on which stages are active.

4. Finally, when the loop ends, the pipeline still needs to be drained, meaning that no new iteration is launched, but any ongoing iteration in the pipeline must successfully terminate. As a result, the new predicates are set to 0, LC no longer changes as it is now 0, and the special register EC (Epilogue Count) tracks the draining progress. EC must be set to the number of loop stages (discussed in Section 3.1) before the loop starts, at the same time as LC. PC and RRB management does not change during the epilogue.

Once the pipeline is empty, execution continues past the loop instruction. We give a summary of the loop instruction functionality in Figure 5.

The next section explains how the compiler schedules instructions for software pipelining, making use of the `loop.pip` capabilities—and this may help clarify the rationale of many of the actions described above.

3 Software Pipelining

To exploit the previously described data structures and behaviour, we rely on modulo scheduling, a loop pipelining strategy introduced in the 90s [4].

3.1 Initiation Interval

We define the Initiation Interval (II) as the number of issue cycles between the initiation of consecutive loop iterations in the schedule. We also define a loop stage as a group of II consecutive bundles. Loop

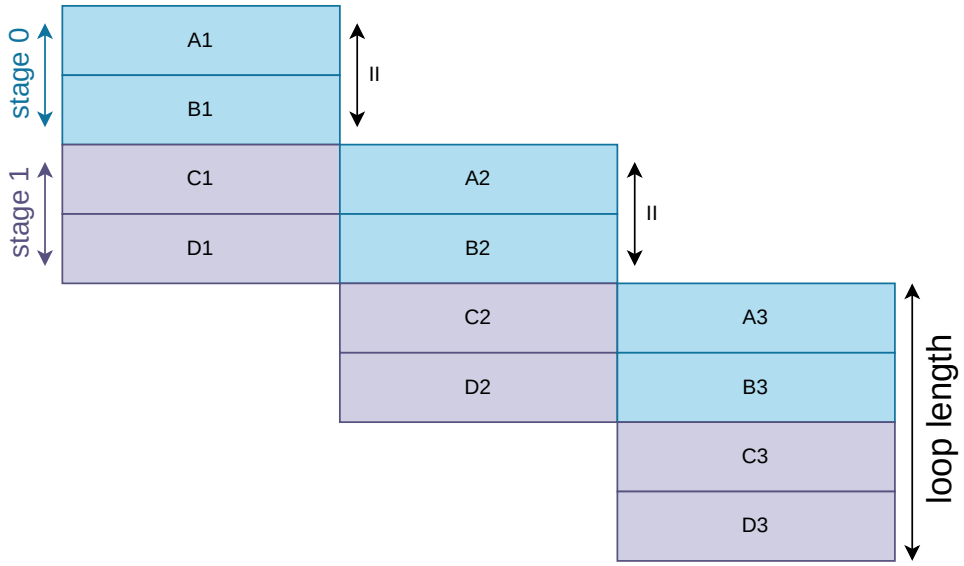


Figure 6: Initiation Interval.

stages cannot overlap and the first stage always starts with the first bundle of the loop body. Each loop stage contains exactly one of the loop instructions. For example, in Figure 6 we give an example of a loop with a body of four instructions, denoted by A, B, C, and D, with an II of 2. We annotate each stage with the loop iteration it belongs. The first stage consists of instructions A and B, and the second stage consists of instructions C and D. In this case, stages A and C are concurrently executed, as well as stages B and D. Concurrent execution of loop stages can only occur if there is no resource contention or dependencies between them. The II is critical as it defines the number of resources available to run the loop body, and the bigger the II, the more resources are available to the loop iterations (e.g., if II were 4, all resources in the processor would be devoted to the instructions of a single iteration). On the other hand, the bigger the II, the less pipelined the loop is, and one wants to minimize the II to maximize the execution units utilization and improve performance. Of course, II is also lower bounded by loop carried dependencies such as accumulating values. Finding the II is an iterative process since each II results in a different schedule.

While it would be possible to iterate over all II values starting from one until one finds the minimal one leading to a valid schedule, it is possible to lower bound the II with a simple calculation. This bound II_{res} is computed as follows. Firstly, we separate the instructions supported by the processor into classes, where instructions belonging to the same class use the same execution unit (arithmetic operations, mult operations, memory operations, and branch operations). Denoting by N_i the number of operation belonging to class i in the loop body and U_i the number of execution units executing operations of class i in the processor, II_{res} is computed as:

$$II_{\text{res}} = \max_i \left(\left\lceil \frac{N_i}{U_i} \right\rceil \right) \tag{1}$$

For example, imagine we try to map a simple program containing 5 arithmetic operations, 1 mult operation, 4 memory operations, and a branch operation on our VLIW470. We define, without loss of generality, class 1 the class of arithmetic operations, class 2 the class of mult operations, class 3 the class of memory operations and class 4 the class of branch operations. Applying the above formula with $N_1 = 5, U_1 = 2, N_2 = 1, U_2 = 1, N_3 = 4, U_3 = 1, N_4 = 1, U_4 = 1$, we find that $II_{\text{res}} = \max(3, 1, 4, 1) = 4$. With an II of 4, the memory unit is maximally utilized, and clearly it would not be possible to schedule the code with a smaller II.

Starting from II_{res}, we can iteratively try increasing II values until we find a valid schedule.

```
0: mov LC, 100
1: mov x2, 0x1000
2: mov x3, 1
3: mov x4, 25
----
4: ld x5, 0(x2)
5: mulu x6, x5, x4
6: mulu x3, x3, x5
7: st x6, 0(x2)
8: addi x2, x2, 1
9: loop 4
----
10: st x3, 0(x2)
```

Figure 7: Simple Loop.

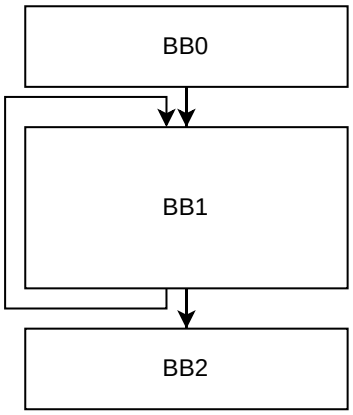


Figure 8: General Loop Structure.

3.2 Scheduling

In this section, we describe the scheduling heuristic we will use in this homework: As Soon As Possible (ASAP) scheduling. We will assume in this homework that the code always consists of three basic blocks, as depicted in Figure 8, where a basic block is a block of code with no branches. BB0 is the initialization code and is executed only once before the loop body begins, BB2 is the finalization code and is executed only once when the loop ends, and BB1 is the loop body and is executed as many times as there are loop iterations. This very simple structure means in particular that we do not allow nested loops. We will use the code snippet depicted in Figure 7 as a running example.

As Soon As Possible Scheduling (ASAP scheduling) is a greedy scheduling algorithm that considers each instruction in program order and schedules it at the earliest possible slot in the schedule. The earliest slot is the one when all instructions it depends on, from the same loop iteration or the loop initialization, have been scheduled and completed, and when there is an appropriate slot free. Tracking instruction dependencies is the job of the dependency analysis and is described afterward.

We start by defining instruction dependencies. There are three ways instructions can depend on each other:

1. **True dependence** (or Read after Write dependency) occurs when some instruction i_2 reads the register i_1 writes to and i_2 is after i_1 in the sequential description of the program. Definitely, scheduling i_2 before i_1 would result in incorrect program behavior. As a result, such dependencies must be represented explicitly by the dependency analysis and correspond to data dependencies.
2. **Anti dependence** (or Write after Read dependency) occurs when some instruction i_2 write to the same location i_1 reads from and i_2 is after i_1 in the sequential description of the program. In this case, it is possible to re-order i_2 before i_1 as long as we modify (rename) the location i_2 writes to (and all subsequent uses). Such dependencies are alleviated with appropriate register allocation (see Section 3.3), and are not tracked by the dependency analysis.
3. **Output dependence** (or Write after Write dependency) occurs when some instruction i_2 writes to the same location as i_1, and i_2 is after i_1 in the sequential description of the program. Clearly, if the two instructions are stores, they need to remain in the same order, as otherwise, the target memory address might hold a wrong value. However, if the two instructions write to the same register, writing the result of the two instructions to different ones (and renaming all subsequent uses) would remove the dependency. In this homework, we will guarantee you that all memory instructions can be considered independent. Therefore, since such dependencies are also alleviated with appropriate register allocation (see Section 3.3), they are not tracked by dependency analysis.

Therefore, dependency analysis accounts for all true (or Read after Write) dependencies of a code snippet. We account for such dependencies in a data structure similar to Table 2. In the table, instructions are identified both with their address in program memory and a capital letter as its identifier. The instruction address refers to instructions in the original code, while the identifier refers to the instruction itself when being scheduled, as the address will eventually change when rescheduling. The instruction column stores

Instr. Addr.	Id.	Instr.	Destination Register	Local Dependencies	interloop Dependencies	Loop Invariant Dependencies	Post Loop Dependencies
0	A	mov	LC	–	–	–	–
1	B	mov	x2	–	–	–	–
2	C	mov	x3	–	–	–	–
3	D	mov	x4	–	–	–	–
4	E	ld	x5	–	x2: (B or I')	–	–
5	F	mul	x6	x5: E	–	x4: D	–
6	G	mul	x3	x5: E	x3: (C or G')	–	–
7	H	st	–	x6: F	x2: (B or I')	–	–
8	I	addi	x2	–	x2: (B or I')	–	–
9	J	loop	–	–	–	–	–
10	K	st	–	–	–	–	x3: G, x2: I

Table 2: Dependency Analysis.

the instruction type. It is not necessary for ASAP scheduling but helps visualize things. The last five columns track instruction dependencies: the destination register column tracks produced registers, and the last four consumed registers. We classify instruction dependencies into four categories. If the producer and the consumer are in the same basic block, it is a *local dependency*. Please note that two loop iterations count as two different basic blocks as going back to the top of the loop involves traversing a branch. If the producer and consumer are in different basic blocks, and the consumer is in the loop body, it is an *interloop dependency*. If the producer is in BB0, and consumers are in BB1, and optionally in BB0 and BB2, it is a *loop invariant*. Finally, if the producer is in BB1 and the consumer in BB2, it is a *post-loop dependency*. The reasons behind such classification will become clear in Section 3.3.

The dependency analysis result of the code snippet given in Figure 8 is in Table 2. We note that any register has at most two producers. Indeed, with the simple loop assumption of this homework and reflected in Figure 8, the only basic block where a register can have two producers is BB1, with a producer within BB1 (from the previous iteration) and another one in BB0.

With the dependency analysis done, we now have the required information for ASAP scheduling. The algorithm picks instructions in sequential order, checks the dependencies as listed in Table 2, and schedules the instruction in the earliest possible slot. We note that checking an interloop dependency between instruction P with latency $\lambda(P)$ and scheduled at the bundle with address in instruction code $S(P)$, and instruction C scheduled at the bundle with address in instruction code $S(C)$ requires checking the following relation:

$$S(P) + \lambda(P) \leq S(C) + II. \tag{2}$$

It simply says that P must be completed before C starts. If it is violated, the policy in this homework in to simply retry to schedule with a higher II value.

Additionally, we note that long latency instructions in the loop initialization, or scheduled at the end of the loop body may incur unnecessary delay slots within the loop body itself. For example, naively

```
0: mov LC, 100
1: mov x2, 5
2: mulu x2, x2, x2
----
3: add x2, x2, x2
4: loop 3
----
5: st x2, 0x1000(x0)
```

(a) Code.

PC	ALU0	ALU1	Mult	Mem	Branch
0	mov LC, 100	mov x2, 5			
1	mulu x2, x2, x2				
2					
3					
4	add x2, x2, x2				loop 4
5				st x2, 0x1000(x0)	

(b) Schedule.

Figure 9: Loop with Bubble.

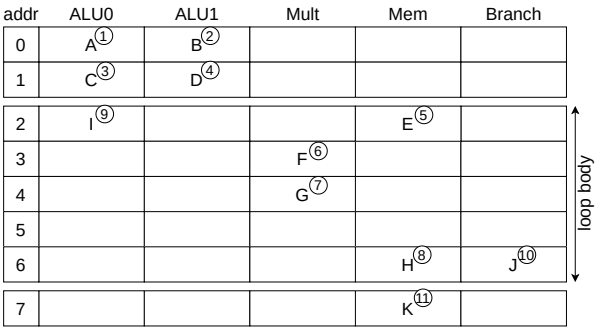


Figure 10: ASAP Scheduling with the `loop` Instruction

scheduling the code in Figure 9a would result in a loop with 3 empty bundles preceding the one actually doing the work. As a result, as depicted in Figure 9b, we always make sure that we add the empty delay bundles before/after the loop, but never within the loop body.

The scheduling heuristic presented in this section is simple. In general, as with most scheduling problems, finding the best pipeline schedule is NP-Complete and people have come up with much more complex heuristics. If interested, feel free to have a look at the original modulo scheduling paper [5] (note that this paper has been very influential and got the prestigious IEEE Micro test of time award). You can also try to improve on the ASAP scheduling by designing your own algorithm, or by adding some additional analysis if you wish. As long as your implementation does not perform significantly worse than simple ASAP scheduling, your homework will be valid.

3.2.1 Scheduling with the *loop* Instruction

ASAP scheduling the code snippet from Figure 7, given the dependency analysis from Table 2, we get the schedule in Figure 10 for a processor without the `loop.pip` instruction.

We decompose the schedule in three portions, corresponding to each of the basic blocks of the loop. Each line in the schedule corresponds to a bundle of instructions, shown with its associated PC. Without the `loop.pip` instruction, the length of the loop body equals the II and that the loop has only one stage. As a result, equation 2 can only be evaluated once all instructions have been scheduled. If equation 2 is violated, rescheduling simply consists in increasing the II by moving the loop instruction down in the schedule—and adapting the code of the third basic block accordingly.

We now explain, as an example, how to schedule instruction G. It depends on instruction E, C, and instruction G' (the result of instruction G from the previous iteration). Instruction C is scheduled at cycle 1 and E at cycle 2, both with a latency of 1. As a result, G needs to be scheduled at least at cycle 3. G also depends on G', but we need the complete loop body schedule to evaluate equation 2 as in our case the II equals the length of the loop body. Hence, we simply try to schedule G from cycle 3 and will evaluate equation 2 when all instructions are scheduled. Then, we try to schedule G in bundle 3; as the instruction F occupies the Mult slot, we try bundle 4, which works. When all instructions have been scheduled, we can check that equation 2 is indeed satisfied for all interloop dependencies. If it had not, we would move instruction J to the next bundle until equation 2 is satisfied.

3.2.2 Scheduling With *loop.pip*

ASAP scheduling the code snippet from Figure 7, given the dependency analysis from Table 2, we get the schedule in Figure 11.

Again, the schedule is decomposed into three portions, corresponding to each of the basic blocks of the loop. However, the loop now has multiple stages of II bundles each, and all stages run concurrently since a loop iteration is launched every II cycles. As a result, we must ensure that there is no resource contention between different loop stages. We denote by $S[s][i][j]$ the slot in schedule S, stage s, bundle with address (in program memory) i, and corresponding to the execution unit j. Then, if we allocate instruction X in the slot at location $S[s][i][j] = \text{X}$, we need to prevent any stage s' such that $s' \neq s$

to schedule another instruction Y at location $S[s'][i][j] = \text{Y}$. To do so, we simply mark any such slot as reserved: $\forall s' : s \neq s', S[s'][i][j] = \text{reserved}$. We indicate this with `--` in Figure 11. Please note that, again, we cannot know the number of loop stages before scheduling all instructions and that stages can be added to the data structure while scheduling. In this case, the reservation information needs to be propagated when stages are added to the schedule. This information is either recomputed by going through the schedule and checking all reserved locations, or by adding a level of indirection and maintaining a separate table for the reservation information.

Applying equation 1 on the code snippet gives an II lower bound of 2; therefore, we start to schedule with an II of 2, as shown in Figure 11a. Other than the slot reservation mechanism, the scheduling process is very similar to the one without `loop.pip`. ASAP Scheduling with an II of 2, we get to the schedule in Figure 11a. However, we see that instruction G violates equation 2, meaning that the II is too small. Indeed, G has an interloop dependency, so its result must be ready before the same instruction G' gets executed in the next iteration. In our processor, it takes 3 cycles for G to generate its result, so apparently $II = 2$ breaks the dependency constraint. As a result, we schedule a second time the loop, this time with an II of 3, and obtain the schedule of Figure 11b. Please note that since the II is known when we start scheduling, equation 2 can be evaluated when instructions are scheduled, and the scheduling process can be aborted before it is completed.

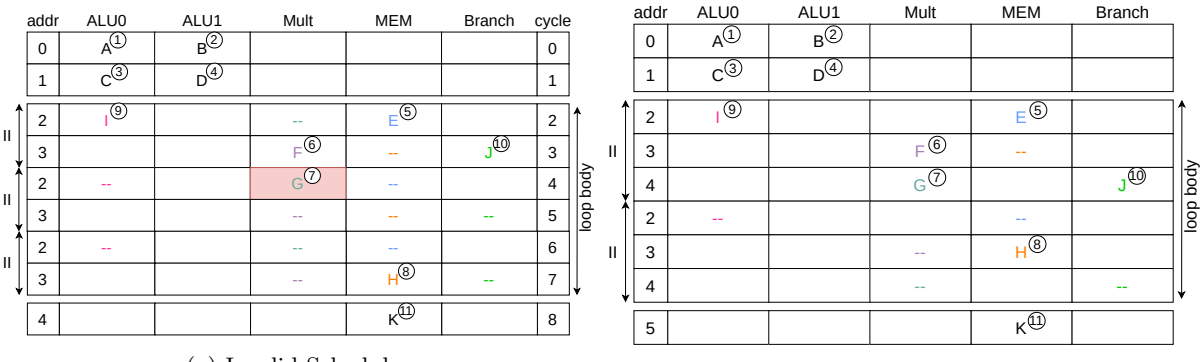
3.3 Register Allocation

3.3.1 Register Allocation with the *loop* Instruction

With a valid schedule computed, we can now perform register allocation. Without `loop.pip`, the register allocation problem is the traditional one. We describe an extremely simple algorithm first and later extend it to support rotating registers. For this homework, we will assume, for simplicity, that you always have enough registers to perform your allocation (and hence you never need to spill registers in memory). While unrealistic in practice, with this assumption we do not need to implement a graph-coloring heuristic, which is outside of the scope of the course. We call the register allocation algorithm without rotating register $alloc_b$; it works in three phases described below.

Firstly, we allocate a fresh unique register to each instruction producing a new value. Since we will be allocating registers from scratch, our allocation will have nothing to do with the original choice of registers. Please note that special registers such as LC and EC are unique and are not considered in the allocation phase. We allocate registers starting from register `x1` and process instructions in the scheduling order. After the first phase, all destination registers in Figure 12 will be specified. Assigning to each register a unique name allows us to handle both the **anti dependence** and **output dependence** presented in Section 3.2.

Secondly, $alloc_b$ links each operand to the register newly allocated in the previous phase. For each instruction X requiring an operand OP, $alloc_b$ finds the producing instruction using the four dependency columns of Table 2, reads the new destination register and replaces the operand of instruction X with it. If the operand value comes from two different producers, one will be in BB1 and another one in BB0



(a) Invalid Schedule.

(b) Valid Schedule.

Figure 11: ASAP Scheduling with `loop.pip`

(as already noted in Section 3.2). In this case, the operand is assigned to the destination register of the producer in BB0, and we ensure that values are correctly propagated between iterations in phase 3. For example, in Figure 7 the load with at address 4 (instruction E) reads the address operand either from the mov in BB0 (instruction B) or from the addi in BB1 (instruction I). In this case, the operand is set to register `x1` as it is the one produced in BB0.

After the second phase, all destination and operand registers have been set in Figure 12, but the two `mov` instructions are not yet there.

Thirdly, $alloc_b$ needs to fix the interloop dependencies. We can see the loop body as a function, and interloop dependencies as function arguments. Then, by choosing the destination register r produced within BB0 as operand of consumers within BB1, we effectively make that register a function argument. Hence, at the end of the loop body, the value produced within BB1 needs to be moved to r to respect calling conventions (and doing so we can effectively see the jump to the top of the loop as a recursive function call). We do this utilizing the `mov` operations, and insert them at the last available bundle of the loop body. If the last bundle is full, or if inserting a `mov` would break an instruction dependency, the loop instruction is simply pushed down, without touching the rest of the schedule, to make enough space for correct execution. For example, we can see that in Figure 12 the `loop` instruction was pushed down (with respect to Figure 10) by one bundle because of the latency of 3 of instruction G.

Fourth, if an instruction reads from a register that no other instruction wrote to before, it is assigned an unused register. Unused registers are assigned traversing instruction in scheduling order.

After this step, the register allocation is done and the loop is ready to be executed. The result is in Figure 12.

3.3.2 Register Allocation with the *loop.pip* Instruction

We now describe the register allocation algorithm when using the `loop.pip` instruction. We call this algorithm $alloc_r$. It will work in four phases. In this section a register written in the form `xZ(+a, +b)` is equivalent to `x(Z + a + b)`. With $Z = 32$, $a = 1$ and $b = 1$ we get `x32(+1, +1)` is equivalent to `x34`. We will denote a the iteration offset, and b the stage offset.

First, $alloc_r$ allocates fresh unique rotating registers to each instruction producing a new value in BB1. Registers rotate when the `loop.pip` instruction is executed, at the end of each loop stage, when the instruction changes the content of the RRB register. Because of this feature, $alloc_r$ must keep track not only of each assigned registers, but also of all the renamed values the same physical register will take, because of the rotations. For this, we proceed as in the $alloc_b$ allocation technique, but only assign one register every # stages + 1 (e.g., `x32`, `x36`, `x40`, etc.. with 3 stages). Indeed, any particular value (e.g., one saved in `x32`) can be accessed at most during the same loop iteration or the next one. Since we have no if then else construct, any assigned register is either read in the current or next iteration. As a result, the maximum lifetime for any register is # stages + 1. During this time, at each stage the value (e.g, the one saved in `x32`) will renamed and accessible through successive registers (e.g., if `x32` is assigned in stage 0, it will be accessible by `x33` in stage 1, `x34` in stage 2, etc..). The dependency of a register r assigned in stage k iteration N − 1 of a loop with K stages is depicted in Figure 14. The longest dependency is

addr	ALU0	ALU1	Mult	Mem	Branch
0	A: mov LC	B: mov x1, 0x1000			
1	C: mov x2, 1	D: mov x3, 25			
2	I: addi x4, x1, 1			E: ld x5, x1	
3			F: mulu x6, x5, x3		
4			G: mulu x7, x5,x2		
5					
6	mov x1, x4			H: st x6, x1	
7	mov x2, x7				J: branch 2
8				K: st x7, x1	

Figure 12: Register Allocation with `loop`

simply the one obtained with $k = 0$.

After this phase, all destination registers of the loop body have been allocated. In Figure 13, registers `x32`, `x35`, `x38` and `x41` have been allocated.

Second, $alloc_r$ allocates nonrotating registers for each loop invariant. Indeed, since the content of these registers does not change throughout the loop execution, a simple register is sufficient. We allocate nonrotating registers starting with register `x1`. The loop invariant column of Table 2, $alloc_r$ identifies instructions of BB1 whose results are invariant. After this stage, in Figure 13 the destination register of instruction D has been allocated to `x1`.

Third, $alloc_r$ links each operand within the loop body to its producer, using the last four columns of Table 2. For loop invariant dependencies, the register assigned in phase two is read and assigned to the corresponding operand. For local loop dependencies, the consumed register name needs to be corrected by the number of times RRB changed since the producer wrote to that register. This is obtained by adding the number of stages separating the producer and the consumer to the destination register of the producer, as depicted by the following equation:

$$\mathtt{x}_D = \mathtt{x}_S + (\mathrm{St(D)} - \mathrm{St(S)}), \tag{3}$$

where $\mathtt{x}_D$ is the operand register and D the instruction consuming $\mathtt{x}_D$, $\mathtt{x}_S$ the produced register name and S the instruction producing $\mathtt{x}_S$, and $\mathrm{St}(X)$ the stage where instruction X is scheduled in. Please note that since S is always scheduled before D, the difference is always positive. In Figure 13, this correction is shown in the second offset after each consumed register, in blue, a denoted as the stage offset. The first offset, in red, will always be 0 in this case.

For interloop dependencies, the operand register is corrected similarly to the local dependencies. Indeed, let's assume that a register r is produced during stage k in iteration $\mathrm{N} - 1$ and consumed somewhere in iteration N. The loop has K stages. This is equivalent to the situation depicted in Figure 14. In this case, after a single increment of the RRB register, the situation is again equivalent to the one described by equation 3. Hence, computing the consumed register name of interloop dependencies can be summarized by the following equation:

$$\mathtt{x}_D = \mathtt{x}_S + (\mathrm{St}(D) - \mathrm{St}(S)) + 1 \tag{4}$$

In Figure 13, this correction is shown in both offsets after each consumed register. The iteration offset, in red, says if the register is produced in the same iteration (+0) or if another one (+1). The stage offset, in blue, accounts for the result of equation 3.

Fourth, $alloc_r$ allocates destination registers for each remaining register in BB0 and BB2. We have the following cases:

- If an instruction in BB0 is writing to a register r such that r is an interloop dependent operand of any instruction C of the loop body, and instruction P produces that operand within the loop body,

	PC	ALU0	ALU1	Mult	MEM	Branch	cycle
	0	A: mov LC, 100	B: mov x32(+1, +0), 0x1000				0
	1	C: mov x41(+1, +0), 1	D: mov x1, 25				1
II	2	I: addi x32, x32(+1, +0), 1			E: ld x35, x32(+1, +0)		2
II	3			F: mulu x38, x35(+0, +0), x1	--		3
II	4			G: mulu x41, x41(+1, +0), x35(+0, +0)		J: loop.pip 2	4
II	2	--			--		5
II	3			--	H: st x38(+0, +1), x32(+1, +1)		6
II	4			--		--	7
	5				K: st x41(+0, +1), x32(+0, +1)		8

Figure 13: Register Allocation with `loop.pip`

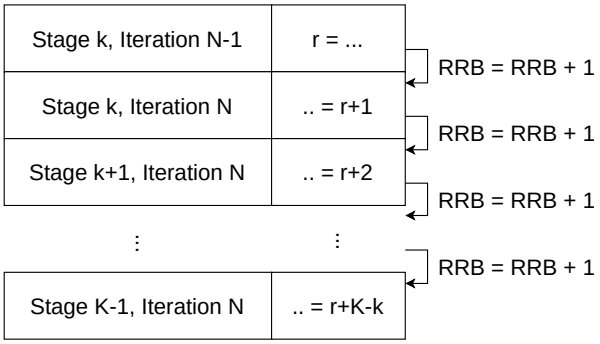


Figure 14: RRB Increments.

then the stage offset is set to $-\mathrm{St}(\mathrm{P})$, the iteration offset is set to 1, and the destination register is the same as the destination register of P.

- If an instruction has a local dependency within BB0 or BB2, register allocation works in the same way as register allocation without `loop.pip` (unless the destination register has already been allocated in Phase 1).
- If an instruction in BB2 has a post dependency, it consumes a register produced within the loop body. This register is produced in the last iteration of the loop. As a result, the iteration offset is always zero, and the stage offset is simply the stage distance between the producer and consumer, where the consumer is assumed to be on the last stage of the loop.
- If an instruction in BB0 or BB2 reads a loop invariant it is simply assigned to the corresponding operand.
- If an instruction reads from a register that no other instruction wrote to before, it is assigned a static, general-purpose, unused register. Unused registers are assigned traversing instructions in scheduling order.

Again, the presented allocation heuristic is simple, and people came up with more complex allocation strategies. If you want to have a look at more advanced scheme, you can have a look at this other paper from Rau et al. [6].

3.4 Preparing the loop

With the scheduling and allocation fixed, the last step is to prepare the loop for execution. These steps are only required for the schedule utilizing the `loop.pip` instruction. First, we can represent the code in a more compact way. Indeed, we already guaranteed that different loop stages can execute concurrently, and as a result, a single block of bundles of length II is sufficient to represent the whole loop body. With

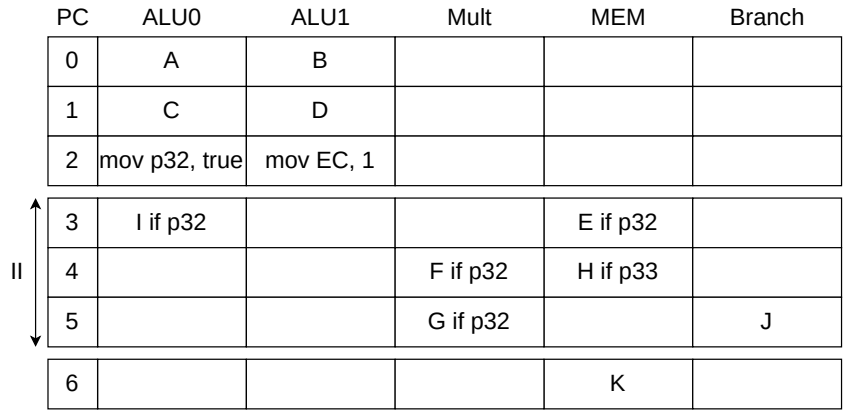


Figure 15: Final Code for the Simple Loop.

```
[
"mov LC, 100",
"mov x2, 0x1000",
"mov x3, 1",
"mov x4, 25",
"ld x5, 0(x2)",
"mulu x6, x5, x4",
"mulu x3, x3, x5",
"st x6, 0(x2)",
"addi x2, x2, 1",
"loop 4",
"st x3, 0(x2)
]
```

(a) Input JSON Corresponding to the Code Snippet of Figure 7

```
[
  ["mov LC, 100", "mov x33, 0x1000", "nop", "nop", "nop"],
  ["mov x42, 1", "mov x1, 25", "nop", "nop", "nop"],
  ["mov p32, true", "mov EC, 1", "nop", "nop", "nop"],
  ["(p32) addi x32, x33, 1", "nop", "nop", "(p32) ld x35, 0(x33)", "nop"],
  ["nop", "nop", "(p32) mulu x38, x35, x1", "(p33) st x39, 0(x34)", "nop"],
  ["nop", "nop", "(p32) mulu x41, x42, x35", "nop", "loop.pip 3"],
  ["nop", "nop", "nop", "st x42 0(x33)", "nop"],
]
```

(b) Output JSON Corresponding to the Schedule of Figure 15.

Figure 16: Reference JSON

this representation, each instruction needs to be predicated with a predicate. All instructions belonging to the same stage are associated with the same predicate, and stage predicates are rotating predicates. In Figure 15, we see that all instruction but H are associated predicate p32, and H is associated with predicate p33. The predicate of the first stages needs to be enabled by the software before the loop, and you can assume that all predicates are initially set to false. Finally, we add another `mov` instruction to initialize the value of EC to the number of stages minus one.

4 Input and Expected Output

The input to your compiler is a simple program in the assembly defined in Table 1. Your goal is to produce two output programs from this description utilizing the algorithms and data structures described in this document:

1. A VLIW schedule utilizing the `loop` instruction.
2. A VLIW schedule utilizing the `loop.pip` instruction.

Both the input code snippet and the VLIW schedules are expressed in JSON. As a reference, you will find in Figure 16a the JSON corresponding to the code snippet in Figure 7, and in Figure 16b the JSON corresponding to the schedule of Figure 15.

5 Submission Guidelines

5.1 Programming Language and Running Environment

We provide you with a Ubuntu 22.04 container where your code will be compiled and run from for grading. You are using the root account in the container (e.g., running `sudo` without providing the password). The container image contains the following programming language toolchains:

- C 17 / C++ 17 (gcc, g++)
- Python 3.10 (python3, pip3)
- Go 1.20 (go)
- Rust 1.67 (cargo, rustc)

- Java (OpenJDK 17)
- Scala 3.2.2 (sbt)
- JavaScript (node 18.14, npm)

You can also use `apt` in `build.sh` (see following section) to install any additional libraries. Please check that your code compiles and runs properly within that environment before the deadline.

If your favorite language is not on the list, you can come and talk to the TAs, and will try to accommodate your case to the extent possible.

5.2 Running Your Code

You must provide a build script with the following name responsible for compiling your simulator and installing the required library dependencies (if any).

```
./build.sh
```

Meanwhile, you must also provide a run script with the following interface responsible for running your simulator using the input program (input.json) and producing the output states for the two loop instructions (loop.json, looppip.json).

```
./run.sh </path/to/input.json> </path/to/loop.json> </path/to/looppip.json>
```

The two scripts and your code must be contained within a single **zip** file (with both scripts at the root of the archive). As this project can be done in pairs, please also include a README.md at the root of the archive with the name and SCIPER of the two people in the pair.

5.3 Test Your Code

We provide basic tests cases for you to test your code, which are under the `test` folder. You can check `desc.txt` under each folder to see their description. We also provide two scripst: `runall.sh` and `testall.sh`. The former script generates output using the input of each test, and the latter one compare your output with the reference output. The same scripts will be used for the real grading.

During grading, we will check the same functionality, but with slightly different code. Your code will also be tested with a set of longer and private tests combining features from all the unit tests you have access to.

References

[1] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the ia-64 architecture. *IEEE Micro*, 20(5):12–23, 2000.

[2] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, 2000.

[3] J. Bharadwaj, W.Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The intel ia-64 compiler code generator. *IEEE Micro*, 20(5):44–53, 2000.

[4] B. Ramakrishna Rau, Michael S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. *SIGMICRO Newsl.*, 23(1–2):158–169, dec 1992.

[5] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, page 63–74, New York, NY, USA, 1994. Association for Computing Machinery.

[6] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, page 283–299, New York, NY, USA, 1992. Association for Computing Machinery.