SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

Systemarchitektur SS 2021

Präsenzblatt 8 (Lösungsvorschläge)

Hinweis: **Dieses Aufgabenblatt wurde von Tutoren erstellt. Die Aufgaben sind für die Klausur weder relevant noch irrelevant, die Lösungsvorschlage weder korrekt noch inkorrekt.**

Aufgabe 8.1: Hazards im Pipelining-Datenpfad

Betrachten Sie folgende MIPS-Instruktionssequenz:

```
  lw    r1, 0(r0)
  lw    r2, 4(r0)
  lw    r4, 8(r0)
  add   r3, r1, r2
  beq   r3, r4 <Target>
  sub   r1, r3, r2
  ...
Target:
  lw    r2, 12(r0)
  add   r3, r1, r2
```

1. Identifizieren und nummerieren Sie Kontrollabhängigkeiten, sowie RAW-, WAR-, WAW-Datenabhängigkeiten.
2. Wir betrachten den MIPS-Prozessor mit einer fünfstufigen Pipeline aus der Vorlesung in verschiedenen Varianten:
   a) mit *Ablaufstalling*
   b) mit *Ablaufstalling* und *frühzeitiger Sprungentscheidung*
   c) mit *frühzeitiger Sprungentscheidung* und *branch delay slots*
   d) mit *frühzeitiger Sprungentscheidung*, *branch delay slots* und *forwarding*
   e) mit *frühzeitiger Sprungentscheidung*, *branch delay slots*, *forwarding* und *stalling*

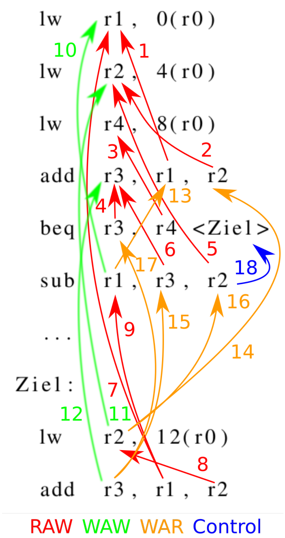
   Geben Sie für jeden Fall an, welche der Abhängigkeiten aus 1 einen *Hazard* verursachen und an welchen Stellen `nop`-Instruktionen eingefügt werden müssen, um auf dem jeweiligen Prozessor korrekt ausgeführt werden zu können. Geben Sie außerdem den Speicherverbrauch für die Instruktionen, sowie die benötigte Anzahl an Prozessorzyklen unter der Annahme, dass die Verzweigung gewählt wird, an.

	a)	b)	c)	d)	e)
Hazards					

modifiziertes Programm	lw r1, 0(r0)
	lw r2, 4(r0)
	lw r4, 8(r0)
	add r3, r1, r2
	beq
	r3, r4 <Target>
	sub r1, r3, r2
	...
	Target:
	lw r2, 12(r0)
	add r3, r1, r2
		lw r1, 0(r0)
		lw r2, 4(r0)
		lw r4, 8(r0)
		add r3, r1, r2
		beq
		r3, r4 <Target>
		sub r1, r3, r2
		...
		Target:
		lw r2, 12(r0)
		add r3, r1, r2
			lw r1, 0(r0)
			lw r2, 4(r0)
			lw r4, 8(r0)
			add r3, r1, r2
			beq
			r3, r4 <Target>
			sub r1, r3, r2
			...
			Target:
			lw r2, 12(r0)
			add r3, r1, r2
				lw r1, 0(r0)
				lw r2, 4(r0)
				lw r4, 8(r0)
				add r3, r1, r2
				beq
				r3, r4 <Target>
				sub r1, r3, r2
				...
				Target:
				lw r2, 12(r0)
				add r3, r1, r2
					lw r1, 0(r0)
					lw r2, 4(r0)
					lw r4, 8(r0)
					add r3, r1, r2
					beq
					r3, r4 <Target>
					sub r1, r3, r2
					...
					Target:
					lw r2, 12(r0)
					add r3, r1, r2

Speicherverbrauch					
Anzahl Prozessorzyklen					

Lösungsvorschlag:

1. 

2. Durch Pipelining können nur RAW- und Kontrollabhängigkeiten verletzt werden, da die Reihenfolge von schreibenden Zugriffen nicht verändert wird.

   a) Ohne Zusatzhardware können die nächsten drei Takte nach einem Verzweigungsbefehl nicht für relevante Berechnungen genutzt werden (Ablaufstalling) und zwischen einem schreibenden und einem nachfolgenden lesenden Zugriff auf ein Register müssen mindestens zwei Instruktionen liegen.

   b) Durch *frühzeitige Sprungentscheidung* wird das Ablaufstalling auf nur noch einen Takt reduziert, was die benötigten Prozessorzyklen um zwei erniedrigt. Es bleiben die gleichen *Hazards* erhalten und wir benötigen entsprechend genauso viele `nop`-Instruktionen wie in Fall a).

   c) Durch die Verwendung des *Branch Delay Slots* wird die Instruktion nach einem Verzweigungsbefehl immer ausgeführt. Da dies in unserem Fall die Semantik des Programms ändern würde, müssen wir eine zusätzliche `nop`-Instruktion einfügen, wodurch vier Byte mehr an Speicher benötigt werden. Die Anzahl an Prozessorzyklen ändert sich allerdings nicht.

d) Durch *Forwarding* treten keine Datenhazards mehr zwischen ALU-Operationen auf und zwischen einer `lw`- und einer davon lesenden Instruktion ist nur noch eine weitere Instruktion nötig. Durch die *frühzeitige Sprungentscheidung* und die damit neu eingeführte RAW-Abhängigkeit muss zusätzlich noch eine weitere Instruktion zwischen der schreibenden Instruktion und der Verzweigung sein.

e) Durch Daten-*Stalling* kann die Hardware selbst mit sämtlichen Datenhazards umgehen, hier wird nur noch die `nop`-Instruktion für den *Branch Delay Slot* benötigt. Dadurch verringert sich wiederum der Speicherverbrauch, während die benötigten Prozessorzyklen gleich bleiben.

	a)	b)	c)	d)	e)
Hazards	RAW: 2, 3, 4, 8	RAW: 2, 3, 4, 8	RAW: 2, 3, 4, 8 Control: 18	RAW: 4, 8 Control: 18	Control: 18

modifiziertes Programm	`lw r1, 0(r0)`
	`lw r2, 4(r0)`
	`lw r4, 8(r0)`
	`nop`
	`add r3, r1, r2`
	`nop`
	`nop`
	`beq r3, r4 <Target>`
	`sub r1, r3, r2`
	`...`
	`Target:`
	`lw r2, 12(r0)`
	`nop`
	`nop`
	`add r3, r1, r2`
		`lw r1, 0(r0)`
		`lw r2, 4(r0)`
		`lw r4, 8(r0)`
		`nop`
		`add r3, r1, r2`
		`nop`
		`nop`
		`beq r3, r4 <Target>`
		`sub r1, r3, r2`
		`...`
		`Target:`
		`lw r2, 12(r0)`
		`nop`
		`nop`
		`add r3, r1, r2`
			`lw r1, 0(r0)`
			`lw r2, 4(r0)`
			`lw r4, 8(r0)`
			`nop`
			`add r3, r1, r2`
			`nop`
			`nop`
			`beq r3, r4 <Target>`
			`nop`
			`sub r1, r3, r2`
			`...`
			`Target:`
			`lw r2, 12(r0)`
			`nop`
			`nop`
			`add r3, r1, r2`
				`lw r1, 0(r0)`
				`lw r2, 4(r0)`
				`lw r4, 8(r0)`
				`add r3, r1, r2`
				`nop`
				`beq r3, r4 <Target>`
				`nop`
				`sub r1, r3, r2`
				`...`
				`Target:`
				`lw r2, 12(r0)`
				`nop`
				`add r3, r1, r2`
					`lw r1, 0(r0)`
					`lw r2, 4(r0)`
					`lw r4, 8(r0)`
					`add r3, r1, r2`
					`beq r3, r4 <Target>`
					`nop`
					`sub r1, r3, r2`
					`...`
					`Target:`
					`lw r2, 12(r0)`
					`add r3, r1, r2`

Speicherverbrauch	$(13+x)\cdot 4$	$(13+x)\cdot 4$	$(14+x)\cdot 4$	$(11+x)\cdot 4$	$(9+x)\cdot 4$
Anzahl Prozessorzyklen	$12+3+4=19$	$12+1+4=17$	$13+4=17$	$10+4=14$	$10+4=14$

Aufgabe 8.2: Hazard Unit

In dieser Aufgabe möchten wir uns näher mit der Implementierung der Hazard Unit für die Fließbandverarbeitung beschäftigen. Um die Aufgabe einfacher zu halten, dürfen Sie annehmen, dass keine Ablaufhazards auftreten, d.h. wir interessieren uns nur für Daten-Hazards. Wir implementieren den Forwarding Mechanismus (Folie 26, Foliensatz 14). Dazu bekommen wir als Eingaben:

1. Aus der Execute Phase: Die Registernummern der Operanden A (rs) und B (rt).
2. Aus der Memory Phase: Die Registernummer des Zielregisters (WriteRegM) und das Registerschreibsignal (RegWriteM).
3. Aus der Write-Back Phase: Die Registernummer des Zielregisters (WriteRegW) und das Registerschreibsignal (RegWriteW).

Die Ausgaben ForwardAE und ForwardBE geben an ob – und wenn ja von wo – wir Daten forwarden. Zeichnen Sie den zugehörigen Schaltkreis.

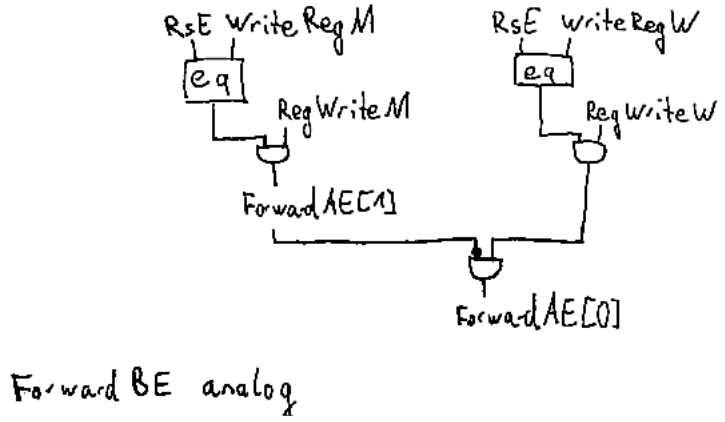
Lösungsvorschlag:

Aufgabe 8.3: Bonus: Dynamische 2-Bit-Sprungvorhersage

In der Vorlesung wurde das Konzept der 2-Bit-Sprungvorhersage vorgestellt: Um die Zahl der Stall-Zyklen bei Sprüngen zu verringern, wird anhand der Vergangenheit schon in der Instruction-Decode-Phase vorhergesagt, ob ein Sprung gemacht wird oder nicht. Wenn die Vorhersage zweimal falsch ist, ändert sich die Entscheidung. Dieser Mechanismus kann als Moore-Automat modelliert werden (siehe Foliensatz 14, Folie 35).

In dieser Aufgabe soll anstatt der in der Vorlesung implementierten Mechanismen Early Branching und Branch Delay Slots eine 2-Bit-Prädiktion implementiert werden. Starten Sie dazu mit dem um Datenhazards bereinigten Pipelining-Datenpfad (Folie 29). Gehen Sie wie folgt vor:

1. Es müssen die vergangenen Entscheidungen gespeichert werden. Die einfachste Möglichkeit ist, im Instruction Memory zwei zusätzliche Bits für jede Instruktion zu speichern. Überlegen Sie sich, welche zusätzlichen Ports der IM braucht.



2. Wie setzen Sie den Moore-Automaten um? Denken Sie daran, dass der Zustandsspeicher des Moore-Automaten jetzt im IM liegt. In welcher Phase der Pipeline können Sie die Ausgabefunktion, in welcher die Übergangsfunktion umsetzen? Wäre es auch möglich, die Sprungvorhersage als Mealy-Automat zu realisieren?

3. Nun können Sie die eigentliche Prädiktion umsetzen. Fällen Sie die Sprungentscheidung mithilfe der Ausgabe des Moore-Schaltwerkes bereits in der ID-Phase. Müssen Sie einen Stall-Zyklus einführen, wenn gemäß der Vorhersage gesprungen werden soll?

4. Es kann immernoch passieren, dass die Vorhersage falsch ist. Dies wird erst in der MEM-Phase bekannt, nachdem die ALU die Verzweigungsbedingung geprüft hat. Was muss die Hazard Unit machen, um diesen Ablaufhazard zu beseitigen?

5. Setzen Sie Ihre obigen Ideen um!

Die in dieser Aufgabe designte Sprungvorhersage hat im einfachen MIPS-Datenpfad gegenüber Early Branching keinen Vorteil, da Early Branching ebenfalls in der ID-Phase eine (immer korrekte) Entscheidung trifft. In komplexeren Datenpfaden mit 15 bis 20 Pipeline-Stufen sieht dies anders aus, dort hat Dynamische Sprungvorhersage einen realen Vorteil.
In der Praxis wird aus Effizienzgründen nicht für jede Instruktion die Sprunggeschichte gespeichert. Bei der Sprungzielvorhersage kann das Sprungziel bereits vor der Dekodierung der Instruktion vorhergesagt werden, sodass der eine nötige Stall-Zyklus entfällt. Eine Übersicht über verschiedene erweiterte Techniken finden Sie z.B. auf Wikipedia[1].

Lösungsvorschlag:

1. Der Instruction Memory muss es ermöglichen, die Branch-Bits der aktuellen Instruktion zu lesen, als auch die Branch-Bits einer bestimmten Instruktion zu aktualisieren, nachdem die tatsächliche Sprungentscheidung bekannt wird.
   - Ausgang `bbits[1:0]`, der die beiden Branch-Bits der aktuellen Instruktion ausgibt.
   - Eingang `bbitsWe`, der das Schreiben der Branch-Bits aktiviert.
   - Eingang `bbitsWa`, der die Adresse (den PC) des IM angibt, an den die Branch-Bits geschrieben werden sollen.
   - Eingang `bbitsWd[1:0]`, der die Werte der Branch-Bits angibt, die geschrieben werden sollen.

2. Die Ausgabefunktion sollte sinnvollerweise in der Instruction-Decode-Phase umgesetzt werden, damit mithilfe der Prädiktion eine frühe Sprungentscheidung möglich wird.
   Die Eingabe des Moore-Automaten ist die tatsächliche Sprungentscheidung. Aus diesem Grund ist es logischerweise nicht sinnvoll, die Vorhersage als Mealy-Automaten umzusetzen: Die Vorhersage sollte nicht

[1] `https://en.wikipedia.org/wiki/Branch_predictor`

von der aktuellen Eingabe abhängen - anderenfalls wäre die Vorhersage sinnlos. Dementsprechend kann die Übergangsfunktion in der WB-Phase umgesetzt werden, also nachdem die ALU die tatsächliche Sprungentscheidung berechnet hat. In dem Fall müssen die aktualisierten Branch-Bits in den Instruktionsspeicher an die Stelle der Branch-Instruktion geschrieben werden. Daher muss auch der PC der Instruktion bis in die WB-Phase durchgereicht werden.

3. Das Moore-Schaltwerk liefert die Branch-Prädiktion. Liefert der Decoder die Information, dass die aktuelle Instruktion eine Branch-Instruktion ist, und sagt das Schaltwerk einen Sprung vorher, dann muss der PC auf das Sprungziel gesetzt werden. Weil der Sprung aber mit einem Zyklus Verzögerung bekannt wird (eben erst in der ID-Phase), muss im Falle eines Sprungs aber ein Stall-Zyklus eingeführt werden, d.h. der Puffer zwischen IF- und ID-Phase muss geflusht werden. Das kann analog zum Vorgehen beim Early Branching (Folie 38) ohne Involvierung der Hazard Unit passieren.

4. Falls die Sprungvorhersage falsch lag, wurde die Ausführung dreier Instruktionen fälschlich begonnen und die Pipeline-Stufen IF, ID und EXE müssen geflusht werden. Ähnlich zu der Behandlung bei Datenhazards werden Stalls durch die Hazard Unit eingefügt. Problematisch ist noch, dass nun der nächste PC korrekt gesetzt werden muss: Entweder zum Sprungziel oder auf die Instruktion nach dem Sprung. Dazu können das potentielle Sprungziel und der potentielle Nachfolge-PC bis in die WB-Phase durchgereicht werden, damit diese dann je nach Entscheidung in den PC geschrieben werden können.

5. Für die Implementierung wird folgende Kodierung der beiden Zustandsbits der Sprungvorhersage verwendet:

Strongly not taken	00
Not taken	01
Taken	10
Strongly taken	11

Damit kann anhand des obersten Zustandsbits entschieden werden, ob gesprungen werden soll oder nicht. Folgende Tabelle beschreibt die Übergangsfunktion (für den Ausgangsautomaten siehe Folie 37). Die Eingabe zero kommt von der ALU und ist 1 gdw der Branch hätte genommen werden sollen.

bbits	zero	bbits'
00	0	00
00	1	01
01	0	00
01	1	11
10	0	00
10	1	11
11	0	10
11	1	11

Die Abbildungen 3 und 4 zeigen eine mögliche Umsetzung. Die unwichtigen Teile, die wie auf Folie 28 bleiben, sind größtenteils nicht noch einmal dargestellt. Die für die Aufgabe wichtigen Bestandsteile sind blau hinterlegt. Der Schaltkreis *bbitsTransition* kann einfach mithilfe von Quine/McCluskey aus der obigen Übergangsfunktion synthetisiert werden und wird daher nicht explizit angegeben.

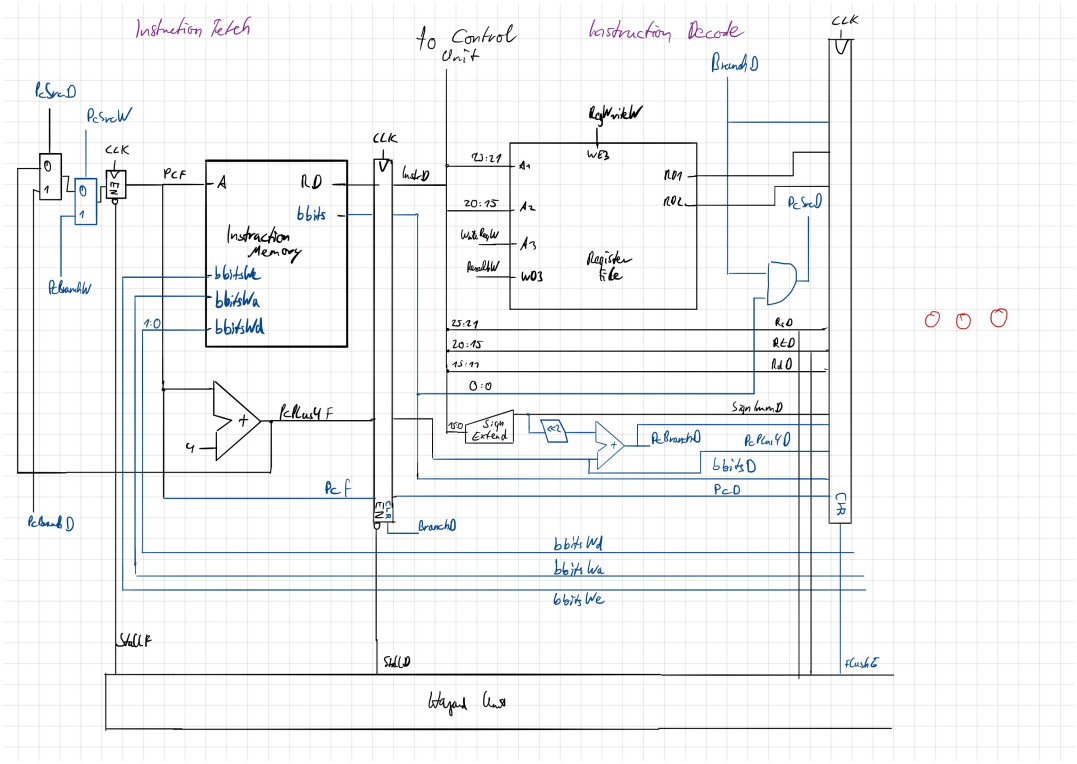


Abbildung 1: 2-Bit-Prädiktion: Implementierung, erster Akt

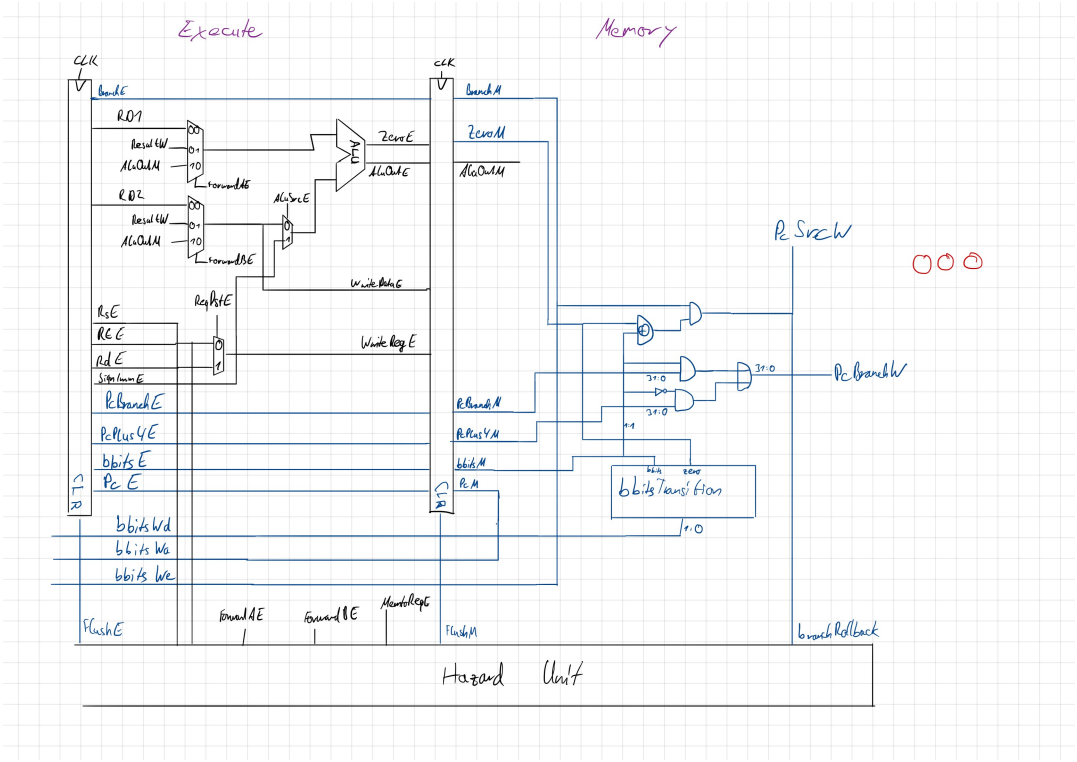


Abbildung 2: 2-Bit-Prädiktion: Implementierung, zweiter Akt

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

System Architecture SS 2021

Tutorial Sheet 8 (Suggested Solutions)

Note: **This task sheet was created by tutors. The tasks are neither relevant nor irrelevant for the exam, the suggested solutions are neither correct nor incorrect.**

Problem 8.1: Hazards

Consider the following MIPS instruction sequence:

```
    lw    r1, 0(r0)
    lw    r2, 4(r0)
    lw    r4, 8(r0)
    add   r3, r1, r2
    beq   r3, r4 <Target>
    sub   r1, r3, r2
    ...
Target:
    lw    r2, 12(r0)
    add   r3, r1, r2
```

1. Identify and number control dependencies, and RAW, WAR, WAW data dependencies.
2. We consider the MIPS processor with a five-stage pipeline from the lecture in different variants:
   a) with *stalling*
   b) with *stalling* and *early branch resolution*
   c) with *early branch resolution* and *branch delay slots*
   d) with *early branch resolution*, *branch delay slots* and *forwarding*
   e) with *early branch resolution*, *branch delay slots*, *forwarding* and *stalling*

   For each case, indicate which of the dependencies from 1 cause a *hazard* and where `nop`-instructions must be inserted in order to execute correctly on the respective processor. Afterwards, specify the memory consumption for the instructions, as well as the required number of processor cycles assuming that branching is chosen.

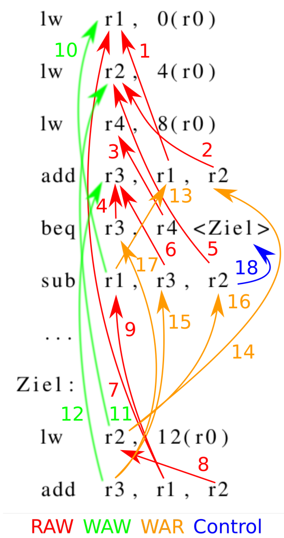
	a)	b)	c)	d)	e)
Hazards					

modified program	lw r1, 0(r0)
	lw r2, 4(r0)
	lw r4, 8(r0)
	add r3, r1, r2
	beq
	r3, r4 <Target>
	sub r1, r3, r2
	...
	Target:
	lw r2, 12(r0)
	add r3, r1, r2
		lw r1, 0(r0)
		lw r2, 4(r0)
		lw r4, 8(r0)
		add r3, r1, r2
		beq
		r3, r4 <Target>
		sub r1, r3, r2
		...
		Target:
		lw r2, 12(r0)
		add r3, r1, r2
			lw r1, 0(r0)
			lw r2, 4(r0)
			lw r4, 8(r0)
			add r3, r1, r2
			beq
			r3, r4 <Target>
			sub r1, r3, r2
			...
			Target:
			lw r2, 12(r0)
			add r3, r1, r2
				lw r1, 0(r0)
				lw r2, 4(r0)
				lw r4, 8(r0)
				add r3, r1, r2
				beq
				r3, r4 <Target>
				sub r1, r3, r2
				...
				Target:
				lw r2, 12(r0)
				add r3, r1, r2
					lw r1, 0(r0)
					lw r2, 4(r0)
					lw r4, 8(r0)
					add r3, r1, r2
					beq
					r3, r4 <Target>
					sub r1, r3, r2
					...
					Target:
					lw r2, 12(r0)
					add r3, r1, r2

memory usage					
Number of processor cycles					

Suggested solution:

1.



2. Pipelining can only violate RAW and control dependencies, because the order of write accesses is not changed.
   a) Without additional hardware, the next three clocks after a branch instruction cannot be used for relevant computations (stalling), and there must be at least two instructions between a write and a subsequent read access to a register.
   b) With *early branch resolution* the stalling is reduced to only one clock cycle which reduces the required processor cycles by two. The same *hazards* remain and we need correspondingly as `nop`-instructions as in case a).
   c) By using the *branch delay slots* instruction after a branching instruction is always executed. Since this would change the semantics of the program in our case, we have to insert an additional `nop`-instruction, which requires four bytes more memory. However, the number of processor cycles does not change.

d) With *forwarding*, data hazards no longer occur between ALU operations and between a `lw` instruction and an instruction reading from it only one more instruction is needed. Due to the *early branch resolution* and the RAW dependence introduced with it, there must be one more instructions between the writing instruction and the branch.

e) Data *stalling* allows the hardware itself to handle all data hazards, here only the `nop`-instruction for the *Branch Delay Slot* is needed. This in turn reduces the memory consumption, while the required processor cycles remain the same.

	a)	b)	c)	d)	e)
Hazards	RAW: 2, 3, 4, 8	RAW: 2, 3, 4, 8	RAW: 2, 3, 4, 8 Control: 18	RAW: 4, 8 Control: 18	Control: 18

modified program	`lw r1, 0(r0)`
	`lw r2, 4(r0)`
	`lw r4, 8(r0)`
	`nop`
	`add r3, r1, r2`
	`nop`
	`nop`
	`beq r3, r4 <Target>`
	`sub r1, r3, r2`
	`...`
	`Target:`
	`lw r2, 12(r0)`
	`nop`
	`nop`
	`add r3, r1, r2`
		`lw r1, 0(r0)`
		`lw r2, 4(r0)`
		`lw r4, 8(r0)`
		`nop`
		`add r3, r1, r2`
		`nop`
		`nop`
		`beq r3, r4 <Target>`
		`sub r1, r3, r2`
		`...`
		`Target:`
		`lw r2, 12(r0)`
		`nop`
		`nop`
		`add r3, r1, r2`
			`lw r1, 0(r0)`
			`lw r2, 4(r0)`
			`lw r4, 8(r0)`
			`nop`
			`add r3, r1, r2`
			`nop`
			`nop`
			`beq r3, r4 <Target>`
			`nop`
			`sub r1, r3, r2`
			`...`
			`Target:`
			`lw r2, 12(r0)`
			`nop`
			`nop`
			`add r3, r1, r2`
				`lw r1, 0(r0)`
				`lw r2, 4(r0)`
				`lw r4, 8(r0)`
				`add r3, r1, r2`
				`nop`
				`beq r3, r4 <Target>`
				`nop`
				`sub r1, r3, r2`
				`...`
				`Target:`
				`lw r2, 12(r0)`
				`nop`
				`add r3, r1, r2`
					`lw r1, 0(r0)`
					`lw r2, 4(r0)`
					`lw r4, 8(r0)`
					`add r3, r1, r2`
					`beq r3, r4 <Target>`
					`nop`
					`sub r1, r3, r2`
					`...`
					`Target:`
					`lw r2, 12(r0)`
					`add r3, r1, r2`

Memory consumption	$(13+x)\cdot 4$	$(13+x)\cdot 4$	$(14+x)\cdot 4$	$(11+x)\cdot 4$	$(9+x)\cdot 4$
Number processor cycles	$12+3+4=19$	$12+1+4=17$	$13+4=17$	$10+4=14$	$10+4=14$

Problem 8.2: Hazard Unit

In this task we would like to take a closer look at the implementation of the Hazard Unit for pipelining. To keep the task simpler, you may assume that no control hazards occur, i.e. we are only interested in data hazards. We implement the forwarding mechanism (slide 26, slide set 14). For this, we get as inputs:

1. From the Execute Phase: The register numbers of operands A (rs) and B (rt).
2. From the Memory Phase: The register number of the destination register (WriteRegM) and the register write signal (RegWriteM).
3. From Write-Back Phase: The register number of the destination register (WriteRegW) and the register write signal (RegWriteW).

The ForwardAE and ForwardBE outputs indicate whether - and if so, from where - we are forwarding data. Draw the corresponding circuit.

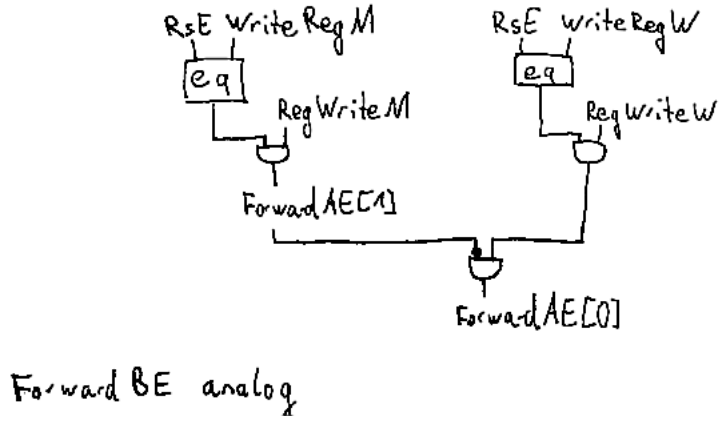
Suggested solution:

Problem 8.3: Bonus: Dynamic 2-bit Branch Prediction

In the lecture, the concept of 2-bit branch prediction was introduced: To reduce the number of stall cycles for jumps, past history is used to predict whether a branch will be taken or not in the decode stage. If the prediction is wrong twice, the decision changes. This mechanism can be modeled as a Moore automaton (see slide set 14, slide 35).

In this task, instead of the Early Branching and Branch Delay Slots mechanisms implemented in the lecture, we will implement a 2-bit prediction. To do this, start with the pipelining datapath without data hazards (slide 29). Proceed as follows:

1. Past decisions need to be saved. The simplest way is to store two additional bits in Instruction Memory for each instruction. Consider which additional ports the IM needs.



2. How do you implement the Moore automaton? Remember that the state memory of the Moore automaton is now in the IM. In which stage of the pipeline can you implement the output function, and in which stage can you implement the transition function? Would it also be possible to implement the branch prediction as a Mealy automaton?

3. Now you can implement the actual prediction. Make the branch decision using the output of the Moore switch already in the ID phase. Do you need to introduce a stall cycle if branching is to be done according to the prediction?

4. It can still happen that the prediction is wrong. This only becomes known in the MEM phase after the ALU has checked the branch condition. What does the hazard unit need to do to eliminate this flow hazard?

5. Implement your ideas above!

The branch prediction designed in this task has no advantage over early branching in the simple MIPS datapath, since early branching also makes a decision (always correct) in the ID phase. In more complex data paths with 15 to 20 pipeline stages, this is different, where dynamic branch prediction has a real advantage. In practice, jump history is not stored for each instruction for efficiency reasons. With branch target prediction, the branch target can be predicted before the instruction is decoded, eliminating the one necessary stall cycle. An overview of various advanced techniques can be found, for example, on Wikipedia[2].

Suggested solution:

1. The instruction memory must allow the branch bits of the current instruction to be read, as well as the branch bits of a particular instruction to be updated when the actual branch decision is known.
   - output `bbits[1:0]`, which outputs the two branch bits of the current instruction.
   - input `bbitsWe` which enables writing of the branch bits.
   - input `bbitsWa` which specifies the address (the PC) of the IM to which the branch bits should be written.
   - input `bbitsWd[1:0]` which specifies the values of the branch bits to be written.

2. The output function should be implemented in the instruction decode phase, so that an early branch decision can be made with the help of prediction.
   The input of the Moore automaton is the actual branch decision. For this reason, it is logically not reasonable to implement the prediction as a Mealy automaton: The prediction should not depend on the actual input - otherwise the prediction would be meaningless. Accordingly, the transition function can be implemented in the WB phase, that is, after the ALU has calculated the actual jump decision. In that case, the updated branch bits must be written to the instruction memory in the place of the branch instruction. Therefore, the PC of the instruction must also be passed through to the WB phase.

[2] `https://en.wikipedia.org/wiki/Branch_predictor`

3. The Moore implementation provides the branch prediction. If the decoder provides the information that the current instruction is a branch instruction, and the switch unit predicts a branch, then the PC must be set to the branch target. However, because the branch is known with a one cycle delay (just in the ID phase), a stall cycle must be introduced in case of a jump, i.e. the buffer between IF and ID phase must be flushed. This can happen analogously to the procedure for early branching (slide 38) without involving the hazard unit.

4. If the branch prediction was wrong, the execution of three instructions was started incorrectly and the pipeline stages IF, ID and EXE must be flushed. Similar to the handling for data hazards, stalls are inserted by the hazard unit. It is still problematic that now the next PC must be set correctly: Either to the branch target or to the instruction after the branch. For this purpose, the potential branch target and the potential successor PC can be passed through to the WB phase so that they can then be written to the PC depending on the decision.

5. The following encoding of the two state bits of the jump prediction is used for the implementation:

Strongly not taken	00
Not taken	01
Taken	10
Strongly taken	11

This allows the first state bit to be used to decide whether to branch or not. The following table describes the transition function (for the output automaton see slide 37). The input zero comes from the ALU and is 1 iff the branch should have been taken.

bbits	zero	bbits'
00	0	00
00	1	01
01	0	00
01	1	11
10	0	00
10	1	11
11	0	10
11	1	11

Figures 3 and 4 show a possible implementation. The unimportant parts that remain as on slide 28 are for mostly not shown again. The parts that are important for the task are highlighted in blue. The circuit *bbitsTransition* can be easily synthesized from the above transition function using Quine/McCluskey and is therefore not explicitly given.

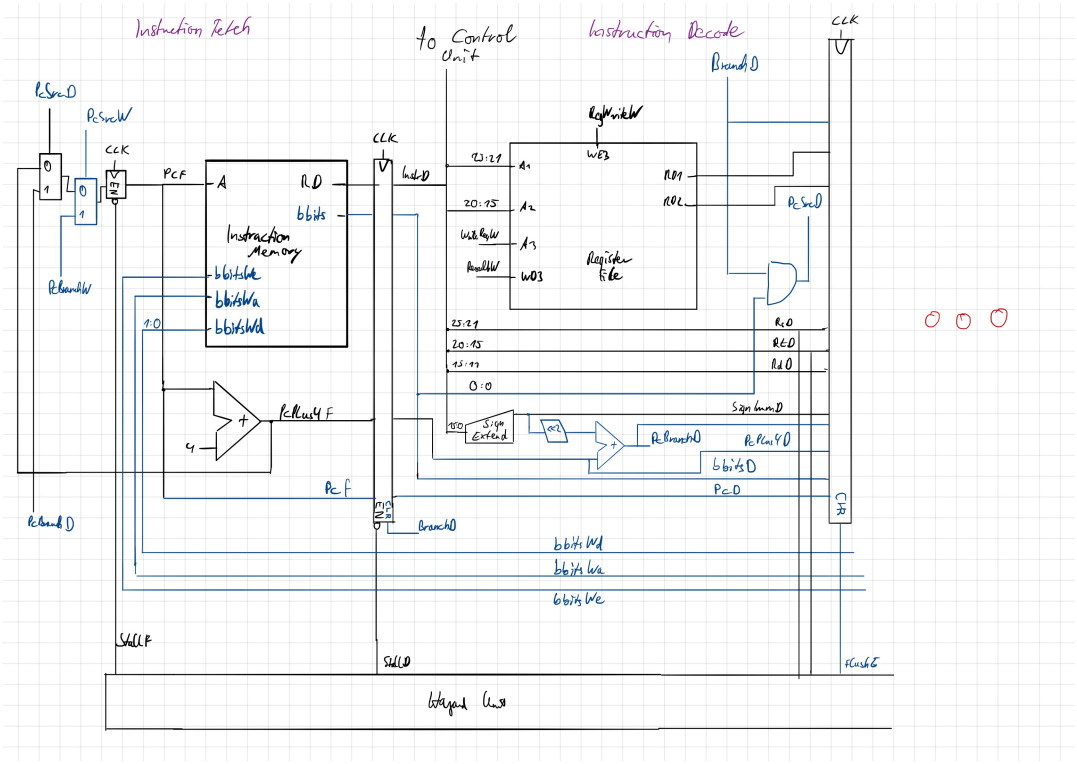


Abbildung 3: 2-bit prediction: implementation, first act

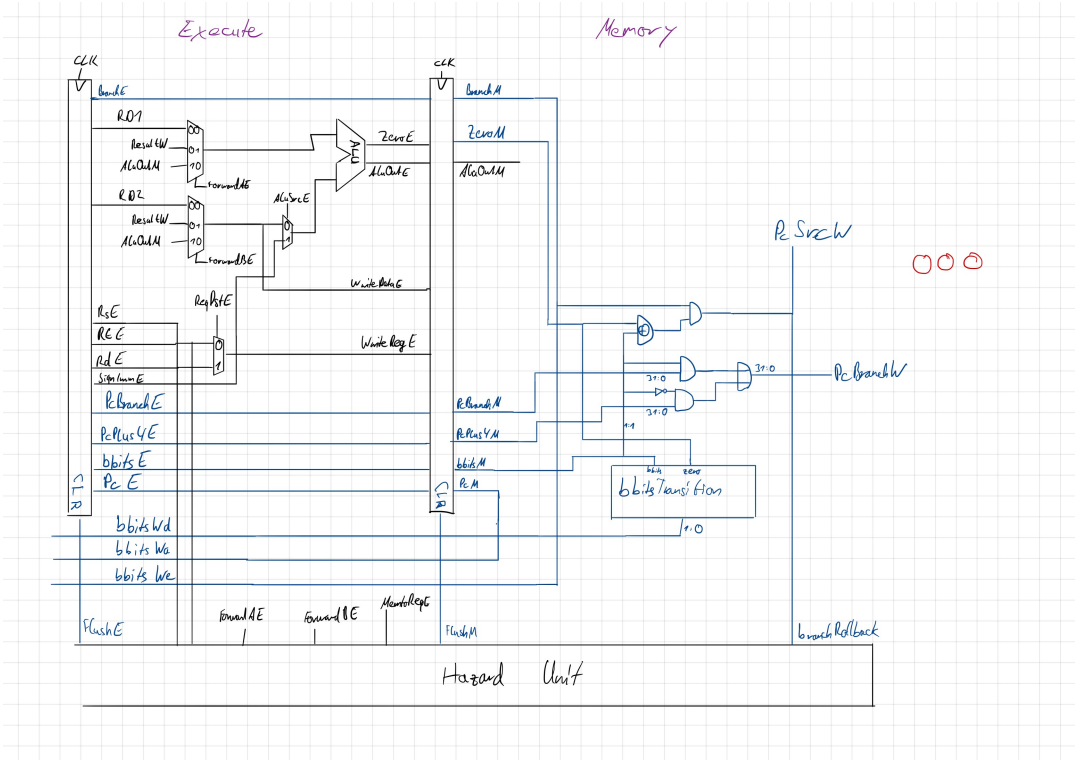


Abbildung 4: 2-bit prediction: implementation, second act