

# Rechnerorganisation und Systemsoftware (WS20)

30. März 2021

Seitenanzahl: 34  
Bearbeitungszeit: 120 Minuten  
Gesamtpunktzahl: 164

Bitte beachten Sie:

- Überprüfen Sie die Vollständigkeit der Klausur anhand der Seitennummern.
- Tragen Sie auf der Titelseite Ihre Matrikelnummer und Ihren Namen ein.
- Lose Blätter und unleserliche Angaben werden nicht korrigiert.
- Legen Sie Ihren Studentenausweis auf den Tisch.
- Nutzen Sie nur dokumentenechte Stifte (kein Rot, keine Blei- oder Buntstifte).
- Begründen Sie Ihre Antworten und geben Sie ggf. den Rechenweg an.
- Die folgenden Hilfsmittel sind zugelassen:  
Persönliche Notizen auf einem beidseitig beschriebenen DIN A4 Blatt,  
Nicht programmierbarer Taschenrechner, der keine Graphen anzeigen und keine Daten speichern kann,
- **Die Verwendung von nicht explizit zugelassenen Hilfsmitteln wird als Täuschungsversuch gewertet.**

**Nachname:**

**Vorname:**

**Matrikelnummer:**

**Unterschrift:**

Aufgabe	1	2	3	4	5	6	7	Summe
Mögliche Punkte	17	19	35	29	18	10	36	164
Erreichte Punkte								



[48-471-2]

## ERKLÄRUNG ZUR PRÜFUNGSFÄHIGKEIT – WINTERSEMESTER 2020/21

---

Stand: 11. Februar 2021

### Erklärung

Jede(r) Prüfling im Raum erklärt zusätzlich durch ihre/seine Teilnahme das Folgende:

- frei von respiratorischen Infektionssymptomen (Husten, Schnupfen, Halsschmerzen), Geruchs- und Geschmacksverlust in Zusammenhang mit Fieber ( $\geq 38,0^\circ\text{C}$ ) zu sein,
- dass sie/er sich im Vorfeld über die Zulässigkeit ihrer/seiner Teilnahme selbstständig informiert hat und
- nicht unter (insb. behördlich angeordneter) häuslicher Quarantäne zu stehen

Jeder Prüfling erklärt des Weiteren, dass ihr/ihm bewusst ist, dass im Falle einer Quarantänepflicht die Teilnahme an der Prüfung solange untersagt wäre, bis die Quarantänezeit abgelaufen ist.

Jeder Prüfling hat zur Kenntnis genommen und versichert durch ihre/seine Teilnahme, dass sie/er

- mindestens eine eigene OP-Maske, FFP-2 oder anderes von der TUK akzeptiertes Modell (keine reine Mund-Nase-Bedeckung) mitgebracht und spätestens ab der Einlasskontrolle, insbesondere auf dem zugewiesenen Platz und bei jedem Betreten und Verlassen des Platzes sowie nach erforderlicher Aufforderung trägt; maßgeblich ist dabei das aktuelle Sicherheitskonzept der TUK;
- einen Mindestabstand von 1,5 m zu anderen Personen jederzeit (auch im Freien! Auf dem Campusgelände sowie im kontrollierbaren Umfeld des Prüfungsraums) einhält;
- die üblichen hygienischen Empfehlungen (z.B. Niesetikette) befolgt.

Jeder Prüfling erklärt weiter, dass ihr/ihm bewusst ist, dass im Falle der oben beschriebenen Symptome eine Teilnahme von den Aufsichtspersonen für diesen Termin untersagt werden kann. Darüber hinaus kann ein Verstoß gegen diese Regelungen und eine Missachtung von Anweisungen des Aufsichtspersonals zu einem jederzeitigen Ausschluss von der Prüfung führen.

---

Nachname,

Vorname,

Unterschrift



[48-471-3]

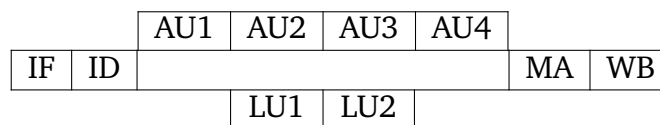
### Aufgabe 1. *Pipelining 1*

(17P)

- (a) **4P** Bestimmen Sie *alle (!)* Datenabhängigkeiten in dem folgenden Abacus-Programm. Geben Sie für jede Befehlszeile  $i$  die von  $i$  abhängigen Anweisungen in der entsprechenden Spalte für die Datenabhängigkeit an.

i	Befehl	RAW	WAR	WAW
0	<b>mov</b> \$1,0			
1	<b>ldi</b> \$2,\$1,0			
2	<b>mul</b> \$3,\$2,\$2			
3	<b>muli</b> \$4,\$2,3			
4	<b>add</b> \$3,\$3,\$4			
5	<b>subi</b> \$2,\$3,8			
6	<b>sti</b> \$2,\$1,0			

- (b) **4P** Arithmetische Anweisungen benötigen mehr Schaltungsaufwand als logische Anweisungen. In der folgenden Pipeline eines Abacus-Prozessors wurde daher die Ausführungsphase (EX) in zwei parallel arbeitende Pipelines aufgetrennt: Arithmetische Anweisungen durchlaufen dabei die 4 Stufen AU1, AU2, AU3, AU4 während logische Anweisungen nur die zwei Stufen LU1, LU2 zur Ausführung benötigen. Die Pipeline verwendet Forwarding, um RAW-Konflikte soweit möglich aufzulösen. Der Prozessor ist allerdings nicht korrekt! Geben Sie ein Abacus-Programm an, dessen Ausführung auf diesem Prozessor zu einem falschen Ergebnis führt. Hinweis: Es reicht ein Programm mit weniger als 5 Befehlen aus.





- (c) **5P** Bestimmen Sie den SpeedUp eines Abacus-Prozessors mit **6-stufiger** Pipeline gegenüber einem Prozessor ohne Pipeline. Der Abacus-Prozessor ohne Pipeline wird mit 200 MHz getaktet. Der Abacus-Prozessor mit Pipeline benötigt in jeder Pipelinestufe 0,1 ns zusätzliche Zeit für den Aufwand der Pipelineorganisation (Forwarding, Pipeline-Register, etc.). Die durchschnittlichen Ausführungshäufigkeiten der verschiedenen Befehlstypen durch den Pipeline-Prozessor sind unten angegeben.

Befehlstyp	Häufigkeit
Speicherbefehl	45%
Sprungbefehl	15%
sonstige Befehle	40%

Um Pipeline-Konflikte aufzulösen, muss der Prozessor mit Pipeline die Pipeline nach jedem Sprungbefehl für einen Takt und nach 50% der Speicherbefehle für 3 Takte anhalten. Berechnen Sie den SpeedUp durch Pipelining. Berechnen Sie auch, wie viele Millionen Anweisungen pro Sekunde (MIPS) vom Pipeline-Prozessor ausgeführt werden.





- (d) 4P Ein Abacus-Prozessor verwendet die unten dargestellte Pipeline mit 13 Stufen und nutzt dabei Forwarding und Register-Bypassing. Nun sollen Sprungbefehle bereits in der Decode-Stufe ID vollständig ausgeführt werden, um die Leistung des Prozessors zu erhöhen.

1	2	3	4	5	6	7	8	9	10	11	12	13
IF	ID	EX1	EX2	EX3	EX4	EX5	MA1	MA2	MA3	MA4	MA5	WB

Welche zusätzlichen Schaltungsmodulare werden benötigt, um die Ausführung der Sprungbefehle in der ID-Stufe zu bewerkstelligen? Wie viele Pipeline-Stalls werden mit dieser Variante des Prozessors bei einem Sprungbefehl vermieden?

## Aufgabe 2. *Pipelining 2*

(19P)

- (a) **5P** Betrachten Sie die Ausführung eines Abacus-Programms, das die Elemente eines Vektors aufsummiert, auf die über einen Indexvektor zugegriffen werden. Die Ausführung zeigt Anweisungen bis (und einschließlich) der ersten Anweisung in der zweiten Iteration der Schleife. Das Programm wird auf einem Abacus-Prozessor mit einer klassischen 5-stufigen Pipeline **mit** Register-Bypassing ausgeführt (Branch-in-Decode wird **nicht** unterstützt, d.h. Sprungbefehle werden normal behandelt und nicht in der Decode-Stufe ID ausgeführt):

1	2	3	4	5
IF	ID	EX	MA	WB

Ergänzen Sie in den unten aufgeführten Befehlssequenzen mit und ohne Forwarding die Zahl der **nop**-Befehle bzw. Pipeline-Stalls, die notwendig sind, um Konflikte in der Pipeline zu vermeiden. *Hinweis: Sie können hierfür eine Kurzschreibweise verwenden:  $i$  nops, wobei  $i$  die Anzahl notwendiger Pipeline-Stalls angibt.*

5 Stufen <b>ohne</b> Forwarding	5 Stufen <b>mit</b> Forwarding
<b>mov</b> \$1,0	<b>mov</b> \$1,0
<b>mov</b> \$2,8	<b>mov</b> \$2,8
<b>mov</b> \$3,0	<b>mov</b> \$3,0
<b>mov</b> \$7,0	<b>mov</b> \$7,0
<b>ld</b> \$4,\$7,\$3	<b>ld</b> \$4,\$7,\$3
<b>mov</b> \$7,8	<b>mov</b> \$7,8
<b>ld</b> \$4,\$7,\$4	<b>ld</b> \$4,\$7,\$4
<b>addu</b> \$1,\$1,\$4	<b>addu</b> \$1,\$1,\$4
<b>addiu</b> \$3,\$3,1	<b>addiu</b> \$3,\$3,1
<b>sltu</b> \$4,\$3,\$2	<b>sltu</b> \$4,\$3,\$2
<b>bnz</b> \$4,-7	<b>bnz</b> \$4,-7
<b>mov</b> \$7,0	<b>mov</b> \$7,0



- (b) 8P Das Programm wird nun auf einem Abacus-Prozessor mit der folgenden 10-stufigen Pipeline (ebenfalls mit Register-Bypassing und ohne Branch-in-Decode) ausgeführt, d.h. die Ausführungsphase ist nun in 3 Stufen und die Speicherzugriffsphase in 4 Stufen aufgeteilt:

1	2	3	4	5	6	7	8	9	10
IF	ID	EX1	EX2	EX3	MA1	MA2	MA3	MA4	WB

Ergänzen Sie wiederum in den unten aufgeführten Befehlssequenzen mit und ohne Forwarding die Zahl der **nop**-Befehle bzw. Pipeline-Stalls, die notwendig sind, um Konflikte in der Pipeline zu vermeiden. *Hinweis: Sie können hierfür eine Kurzschreibweise verwenden:  $i$  nops, wobei  $i$  die Anzahl notwendiger Pipeline-Stalls angibt.*

10 Stufen <b>ohne</b> Forwarding	10 Stufen <b>mit</b> Forwarding
<b>mov</b> \$1,0	<b>mov</b> \$1,0
<b>mov</b> \$2,8	<b>mov</b> \$2,8
<b>mov</b> \$3,0	<b>mov</b> \$3,0
<b>mov</b> \$7,0	<b>mov</b> \$7,0
<b>ld</b> \$4,\$7,\$3	<b>ld</b> \$4,\$7,\$3
<b>mov</b> \$7,8	<b>mov</b> \$7,8
<b>ld</b> \$4,\$7,\$4	<b>ld</b> \$4,\$7,\$4
<b>addu</b> \$1,\$1,\$4	<b>addu</b> \$1,\$1,\$4
<b>addiu</b> \$3,\$3,1	<b>addiu</b> \$3,\$3,1
<b>sltu</b> \$4,\$3,\$2	<b>sltu</b> \$4,\$3,\$2
<b>bnz</b> \$4,-7	<b>bnz</b> \$4,-7
<b>mov</b> \$7,0	<b>mov</b> \$7,0



- (c) **6P** Die Taktfrequenz des Abacus-Prozessors ohne Pipeline, von dem die Pipeline-Implementierungen in den Teilen (a) und (b) abgeleitet sind, beträgt 280 MHz. Beide Implementierungen benötigen pro Pipeline-Stufe zusätzlich 0,05 ns Zeit für den zusätzlichen Aufwand in der Pipeline (Forwarding, Pipeline-Register, etc.). Berechnen Sie die Ausführungszeiten (in Nanosekunden) des Programms aus den Teilen (a) und (b) auf den jeweiligen Pipeline-Prozessoren mit Forwarding. Das Programm terminiert nach der letzten Schleifeniteration mit einem **sync** Befehl.







### Aufgabe 3. Cache Architektur (35P)

Betrachten Sie das folgende Abacus-Programm, welches für die Elemente  $a[i]$ ,  $b[i]$ ,  $c[i]$  der Vektoren  $a, b, c$  die Zuweisung  $c[i] = c[i] + a[i] * b[i]$  ausführt. Die Elemente  $a[i]$ ,  $b[i]$ ,  $c[i]$  der Vektoren  $a, b, c$  befinden sich dabei an den Adressen  $4i$ ,  $6i$  und  $8i$  und benötigen jeweils 1 Byte. *Hinweis: Stören Sie sich nicht an den Überschneidungen der Vektoren.*

```

0:      mov $1,0           // Schleifenvariable i
1:      mov $7,8           // Vektorlaenge in $7
2:  L:   muliu $2,$1,8      // lade c[i] in $4
3:      ldi $4,$2,0
4:      muliu $3,$1,4      // lade a[i] in $5
5:      ldi $5,$3,0
6:      muliu $3,$1,6      // lade b[i] in $6
7:      ldi $6,$3,0
8:      mul $5,$5,$6       // berechne a[i]*b[i]
9:      add $4,$4,$5       // c[i] = c[i] + a[i]*b[i]
10:     sti $4,$2,0        // speichere c[i]
11:     addiu $1,$1,1      // i = i+1
12:     slt $2,$1,$7       // i < n
13:     bnz $2,L

```

Gehen Sie von einem byteadressierten (d.h. 1 Wort = 1 Byte) Hauptspeicher von 512 Byte aus sowie von den folgenden Cache-Konfigurationen aus:

DM	direkt adressiert
FA	vollassoziativ
A2	2-fach satzassoziativ
A4	4-fach satzassoziativ




Die Größe jedes Caches beträgt 16 Byte, aufgeteilt in 2 Byte große Cache-Zeilen (d.h. Blockgröße = 2 Byte). Alle Caches verwenden die **least recently used (LRU)** Ersetzungsstrategie und die **write back** Strategie, um Werte in den Hauptspeicher zu schreiben. Außerdem wird angenommen, dass alle Caches zu Beginn der Programmausführung leer sind (Valid-Bits sind auf 0 gesetzt).

(a) 4P Bestimmen Sie die folgenden Zahlen für die o.g. Cache-Konfigurationen:

Cache	Sätze	Blöcke/Satz	Tag-Bits	Satz-Bits
DM				
FA				
A2				
A4				

[illegible]

- (b) 2P Ordnen Sie die Cache-Konfigurationen in absteigender Reihenfolge ihrer Hardware-Komplexität (höchste Komplexität links bis niedrigste Komplexität rechts).

		
---	---	---

- (c) **10P** Betrachten Sie zunächst den direktadressierten (**DM**) Cache. Tragen Sie für die Ausführung der Abacus-Programmzeilen 3, 5 und 7 (Ladebefehle) während jeder Iteration  $i$  der Schleife in die untenstehende Tabelle die Adresse (Dezimalwert) der zugewiesenen Speicherstelle (Adr.), den Tag-Wert (Tag) und die Satzadresse (Satz) im DM-Cache ein.

[illegible]



- (d) **8P** Betrachten Sie erneut den direktadressierten (**DM**) Cache. Tragen Sie nun für die Ausführung der Abacus-Programmzeilen 3, 5, 7 (Ladebefehle) und 10 (Speicherbefehle) während jeder Iteration  $i$  der Schleife in die untenstehende Tabelle den Status des Cache-Zugriffs ein (Hit oder Miss in Spalte H/M) und vermerken Sie zusätzlich in Spalte WB, ob der Cache-Zugriff zu einem Rückschreiben in den Hauptspeicher führt (Ja oder Nein).

Iter. i	Zeile 3		Zeile 5		Zeile 7		Zeile 10	
	H/M	WB	H/M	WB	H/M	WB	H/M	WB
1								
2								
3								
4								
5								
6								
7								
8								

- (e) **8P** Füllen Sie nun die untenstehende Tabelle für den vollassoziativen **FA-Cache** analog zu Teilaufgabe (d) aus.

Iter. i	Zeile 3		Zeile 5		Zeile 7		Zeile 10	
	H/M	WB	H/M	WB	H/M	WB	H/M	WB
1								
2								
3								
4								
5								
6								
7								
8								

- (f) **3P** Berechnen Sie die Cache-Trefferrate für den DM- und den FA-Cache. Wie hoch ist die Gesamtzahl der für beide Caches erforderlichen Write-Backs?

**Aufgabe 4. Compiler Frontend – Parser 1****(29P)**

- (a) 3P Bestimmen Sie eine kontextfreie Grammatik für die unten angegebene Sprache  $L$ :

$$L = \{a^n d c^{2n} \cup a^n b^m c^{3m} \mid m > 0, n \geq 0\}$$

- (b) 4P Bestimmen Sie eine äquivalente LL(1)-Grammatik für die Grammatik  $G = (\{S\}, \{a, +, (, )\}, P, S)$  mit den folgenden Produktionsregeln  $P$ :

$$S \rightarrow a \mid (S) \mid S + S$$

- $$B \rightarrow BA0 \mid \epsilon$$



- (d) 5P Betrachten Sie die Grammatik  $G = (N, T, P, S)$  mit den Nichtterminalsymbolen  $N = \{S, L, R, M, F\}$ , den Terminalsymbolen  $T = \{n, v, *, =, \odot\}$  sowie den folgenden Produktionsregeln  $P$ :

$$F \rightarrow \odot R \mid \epsilon$$

Bestimmen Sie die First-Mengen aller Nichtterminalsymbole.

$F$	
$L$	
$M$	
$R$	
$S$	

- (e) 5P Bestimmen Sie die Follow-Mengen aller Nichtterminalsymbole.

$F$	
$L$	
$M$	
$R$	
$S$	

(f) **6P** Bestimmen Sie die Parsertabelle für einen LL(1) Top-Down-Parser.

	$*$	$=$	$\odot$	$n$	$v$	$\$$
$F$						
$L$						
$M$						
$R$						
$S$						

(g) **3P** Geben Sie den Aktionen des Parsers beim Parsen des Wortes  $*v = n$  an. Füllen Sie dazu die unten angegebene Tabelle aus, bei der in jeder Zeile der Stackinhalt, das restliche Wort und die ausgewählte Aktion eingetragen werden sollen. *Hinweis: Die Tabelle enthält mehr Zeilen als notwendig sind.*

[illegible]



**Aufgabe 5. Compiler Frontend – Parser 2****(18P)**

Betrachten Sie die kontextfreie Grammatik  $G = (\{A, B\}, \{a, b\}, P, A)$  mit den folgenden Produktionsregeln  $P$ :

$$A \rightarrow aB$$

$$B \rightarrow bAB \mid a$$

- (a) 8P Konstruieren Sie den deterministischen endlichen Zustandsautomaten aus LR(0)-Elementen. Geben Sie dazu zunächst die LR(0)-Elemente jedes Zustands (beschriftet mit einer ganzen Zahl) in einem separaten Feld unten an. Zeichnen Sie dann den Automaten, der die Übergänge zwischen diesen Zustandsbezeichnungen zeigt. *Hinweis: Möglicherweise gibt es mehr Kästchen als nötig.*

s0	s1	s2	s3	s4

s5	s6	s7	s8	s9





- (b) 6P Bestimmen Sie die Aktionstabelle für einen SLR(1) Bottom-up-Parser. *Hinweis: Die Tabelle kann mehr Zeilen als nötig enthalten.*

Zustand	$a$	$b$	$\$$	$A$	$B$

- (c) 4P Geben Sie die einzelnen Schritte für das Parsen des Wortes  $abaa$  bis zu dessen Akzeptanz oder Ablehnung an. Geben Sie in den Zeilen der folgenden Tabelle den Stack-Inhalt, das verbleibende Eingabewort und die nächste auszuführende Aktion an. *Hinweis: Die Tabelle kann mehr Zeilen als nötig enthalten.*

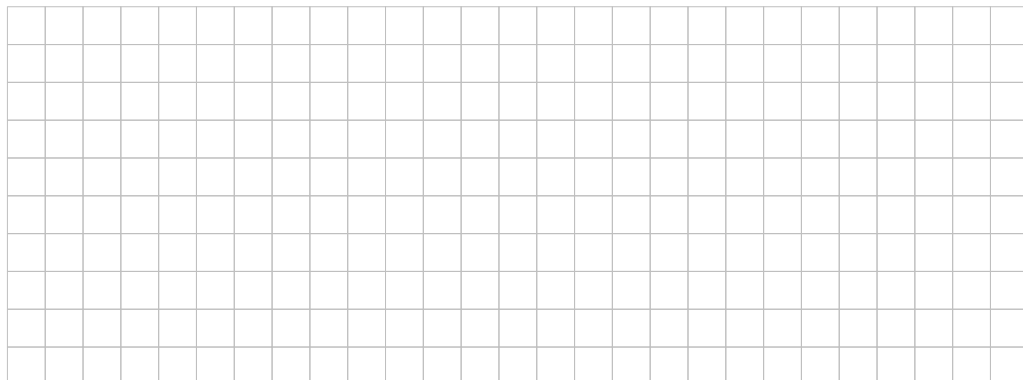
Stack	Wort	Aktion

**Aufgabe 6. Compiler Backend – Codegenerierung 1****(10P)**

*Hinweis: Beachten Sie die Referenzkarten zu MiniC, Cmd, und Abacus im Klausuranhang.*

- (a) 6P Übersetzen Sie das folgende MiniC-Programm, das prüft, ob eine gegebene Zahl  $n$  prim ist, in das entsprechende Cmd-Programm. Sie können so viele temporäre Variablen  $t_0, t_1, t_2, \dots$  verwenden, wie Sie benötigen.

MiniC-Program	Cmd-Program
<pre> thread Prime {   nat n,p,i;    p = 1;   if (n%2 == 0) {     p = 0;   }   else {     i = 3;     do {       if (n%i == 0) {         p = 0;       }       i = i + 2;     } while ((i &lt; n/2) &amp; (p != 0))   } } </pre>	





- (b) 4P Übersetzen Sie das folgende Cmd-Programm, welches das Skalarprodukt 'c' zweier Vektoren 'a' und 'b' berechnet, in das entsprechende Abacus-Programm. Die Vektoren 'a', 'b' und 'c' sind als Sequenz im Hauptspeicher abgelegt, und ihre Startadressen sind jeweils 0, 5 und 10. Verwenden Sie dabei die folgende Registerallokation:

Register	Variables
\$1	m
\$2	i
\$3	t1,t2,t3
\$4	t0

Cmd-Program	Abacus-Program
<pre> 0: m := 5 1: i := 0 2: t0 := a[i] 3: t1 := b[i] 4: t2 := t0 * t1 5: c[i] := t2 6: i := i + 1 7: t3 := i &lt; m 8: if t3 goto 2 </pre>	





**(36P)**

*Hinweis: Beachten Sie die Referenzkarten zu MiniC, Cmd, und Abacus im Klausuranhang.*

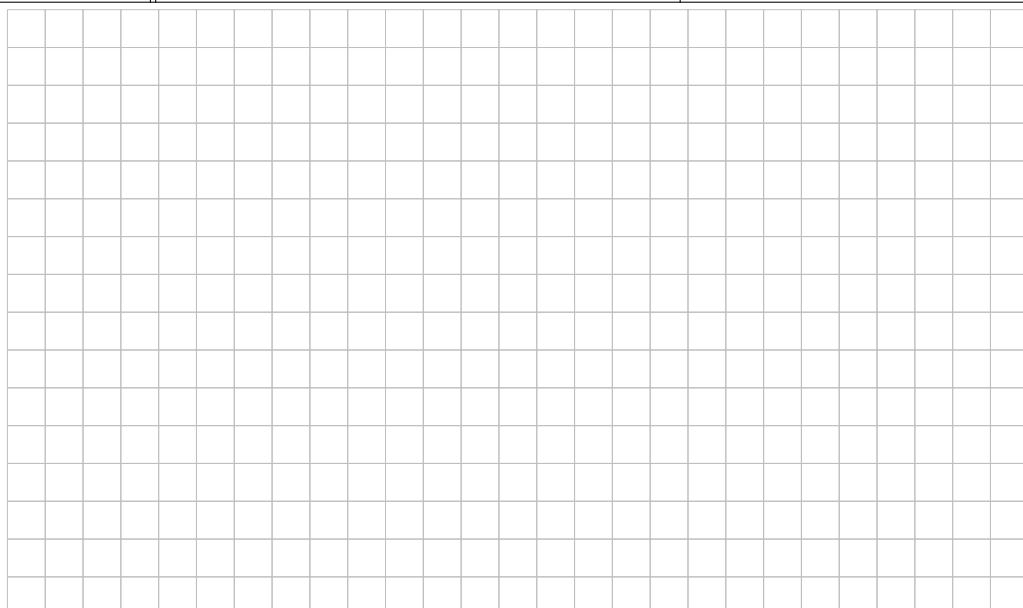
- (a) **6P** Das folgende Cmd-Programm implementiert eine binäre Suche, um die Position einer Zahl 'n' in einem sortierten Array 'a' der Länge 'm' zu finden. Bestimmen Sie die Mengen der Variablen, die von jeder Anweisung gelesen ('Reads') und geschrieben ('Writes') werden sowie die Menge der Anweisungen, die jeder Anweisung im Programm folgen ('Successors'). *Hinweis: Sie können annehmen, dass 'return'-Befehle keine Nachfolger haben, d.h. 'Successors' der leeren Menge entspricht.*

i	Befehl	Reads	Writes	Successors
0	lo := 0			
1	hi := m - 1			
2	t0 := hi < lo			
3	if t0 goto 19			
4	t1 := lo + hi			
5	m := t1 / 2			
6	t2 := a[m]			
7	t3 := n <= t2			
8	if t3 goto 11			
9	lo := m + 1			
10	goto 17			
11	t4 := a[m]			
12	t5 := t4 <= n			
13	if t5 goto 16			
14	hi := m - 1			
15	goto 17			
16	return m			
17	t6 := lo <= hi			
18	if t6 goto 4			
19	return -1			



- (b) 8P Bestimmen Sie für alle Programmzeilen des vorigen Programs die Gleichungen zur Bestimmung der May-Live-Variablen und der May-Used-Variablen.

i	Instruktion	Live(i)	Used(i)
0	lo := 0		
1	hi := m - 1		
2	t0 := hi < lo		
3	if t0 goto 19		
4	t1 := lo + hi		
5	m := t1 / 2		
6	t2 := a[m]		
7	t3 := n <= t2		
8	if t3 goto 11		
9	lo := m + 1		
10	goto 17		
11	t4 := a[m]		
12	t5 := t4 <= n		
13	if t5 goto 16		
14	hi := m - 1		
15	goto 17		
16	return m		
17	t6 := lo <= hi		
18	if t6 goto 4		
19	return -1		





- (c) 6P Berechnen Sie die Menge der May-Live-Variablen für jede Programmzeile. Tragen Sie die berechneten Ergebnisse in die untenstehende Tabelle ein, wobei jede Spalte  $\text{Live}_j$  die Ergebnisse der Iteration  $j$  der Berechnung bezeichnen soll. *Hinweis: Sie können einen gerichteten Pfeil ( $\downarrow \uparrow \rightarrow$ ) von der Zelle darüber, darunter oder links daneben als Kurzschreibweise benutzen, um anzugeben, dass Zellen den gleichen Inhalt haben.*

i	$\text{Live}_0$	$\text{Live}_1$	$\text{Live}_2$	$\text{Live}_3$
0	{}			
1	{}			
2	{}			
3	{}			
4	{}			
5	{}			
6	{}			
7	{}			
8	{}			
9	{}			
10	{}			
11	{}			
12	{}			
13	{}			
14	{}			
15	{}			
16	{}			
17	{}			
18	{}			
19	{}			





- (d) **6P** Berechnen Sie die Menge der May-Used-Variablen für jede Programmzeile. Tragen Sie die berechneten Ergebnisse in die untenstehende Tabelle ein, wobei jede Spalte  $\text{Used}_j$  die Ergebnisse der Iteration  $j$  der Berechnung bezeichnen soll. *Hinweis: Sie können die Kurzschreibweise  $t_i - t_j$  als Angabe für  $t_i$  bis  $t_j$  verwenden. Hinweis: Sie können einen gerichteten Pfeil ( $\downarrow \uparrow \rightarrow$ ) von der Zelle darüber, darunter oder links daneben als Kurzschreibweise benutzen, um anzugeben, dass Zellen den gleichen Inhalt haben.*

i	Used <sub>0</sub>	Used <sub>1</sub>	Used <sub>2</sub>	Used <sub>3</sub>
0	{}			
1	{}			
2	{}			
3	{}			
4	{}			
5	{}			
6	{}			
7	{}			
8	{}			
9	{}			
10	{}			
11	{}			
12	{}			
13	{}			
14	{}			
15	{}			
16	{}			
17	{}			
18	{}			
19	{}			







- (e) **5P** Ermitteln Sie die Lebensdauer jeder Variablen im Cmd-Programm. Tragen Sie in die untenstehende Tabelle die Programmzeilen ein, an denen die Lebensdauer der einzelnen Variablen beginnt und endet. Wie groß ist die Mindestanzahl an Registern, die bei der Zuweisung von Registern nach der Linear-Scan-Methode erforderlich ist? *Bemerkung: Nehmen Sie an, dass die Array-Variable 'a' auf den Hauptspeicher abgebildet wird und daher hier für die Registerzuweisung nicht berücksichtigt werden muss.*

Variable	Beginn	Ende
t0		
t1		
t2		
t3		
t4		
t5		
t6		
hi		
lo		
m		
n		



- (f) 5P Bestimmen Sie die Interferenzen zwischen den Variablen im vorigen Cmd-Programm. Geben Sie hierzu in der untenstehenden Tabelle für jede Variable an, mit welchen anderen Variablen diese Variable interferiert. Wie viele Register werden nach der Graphfärbungsmethode mindestens benötigt? *Be-merkung: Nehmen Sie an, dass die Array-Variable 'a' auf den Hauptspeicher abgebildet wird und daher hier für die Registerzuweisung nicht berücksichtigt werden muss.*

Variable	interferierende Variablen
t0	
t1	
t2	
t3	
t4	
t5	
t6	
hi	
lo	
m	
n	













## Abacus Language Reference Card

conventions and registers		scalar instructions	
<b>bv2nat</b> (b)	interpret bitvector b as radix-2 number	<b>add</b> rd, r1, rr	add signed $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) + \text{bv2int}(\text{Reg}[rr])$
<b>bv2int</b> (b)	interpret bitvector b as two's-complement number	<b>addu</b> rd, r1, rr	add unsigned $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) + \text{bv2nat}(\text{Reg}[rr])$
<b>Reg</b> [x]	scalar register x ( $x \in \{0, \dots, 7\}$ )	<b>addi</b> rd, r1, c	add signed immediate $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) + \text{bv2int}(c)$
<b>Vec</b> [x]	vector register x ( $x \in \{0, \dots, 7\}$ )	<b>addiu</b> rd, r1, c	add unsigned immediate $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) + \text{bv2nat}(c)$
<b>ovflw</b>	register for double precision arithmetic	<b>sub</b> rd, r1, rr	subtract signed $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) - \text{bv2int}(\text{Reg}[rr])$
<b>pc</b>	program counter	<b>subu</b> rd, r1, rr	subtract unsigned $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) - \text{bv2nat}(\text{Reg}[rr])$
<b>l1</b>	load-linked register	<b>subi</b> rd, r1, c	subtract signed immediate $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) - \text{bv2int}(c)$
<b>vlen</b>	vector length register	<b>subiu</b> rd, r1, c	subtract unsigned immediate $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) - \text{bv2nat}(c)$
<b>vmask</b>	vector mask register	<b>mul</b> rd, r1, rr	multiply signed $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) * \text{bv2int}(\text{Reg}[rr])$
load/store instructions		<b>mulu</b> rd, r1, rr	multiply unsigned $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) * \text{bv2nat}(\text{Reg}[rr])$
scalar load/store instructions		<b>muli</b> rd, r1, c	multiply signed immediate $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) * \text{bv2int}(c)$
vector load/store instructions		<b>maliu</b> rd, r1, c	multiply unsigned immediate $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) * \text{bv2nat}(c)$
<b>ld</b> rd, r1, rr	load word $\text{Reg}[rd] = \text{Mem}[\text{bv2nat}(\text{Reg}[r1]) + \text{bv2nat}(\text{Reg}[rr])] = \text{Reg}[rd]$	<b>div</b> rd, r1, rr	divide signed $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) / \text{bv2int}(\text{Reg}[rr])$
<b>st</b> rd, r1, rr	store word $\text{Mem}[\text{bv2nat}(\text{Reg}[r1]) + \text{bv2nat}(\text{Reg}[rr])] = \text{Reg}[rd]$	<b>divu</b> rd, r1, rr	divide unsigned $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) / \text{bv2nat}(\text{Reg}[rr])$
<b>ldi</b> rd, r1, c	load word immed. $\text{Reg}[rd] = \text{Mem}[\text{bv2nat}(\text{Reg}[r1]) + \text{bv2nat}(c)] = \text{Reg}[rd]$	<b>divi</b> rd, r1, c	divide signed immediate $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) / \text{bv2int}(c)$
<b>ldi</b> rd, r1, c	store word immed. $\text{Mem}[\text{bv2nat}(\text{Reg}[r1]) + \text{bv2nat}(c)] = \text{Reg}[rd]$	<b>diviu</b> rd, r1, c	divide unsigned immediate $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) / \text{bv2nat}(c)$
vector load/store instructions		scalar comparison operations	
<b>lv</b> vd, r1, rr	load vector $\text{for}(i=0..15)\text{Vec}[vd][i] = \text{Mem}[\text{Reg}[r1] + \text{bv2nat}(\text{Reg}[rr]) + i]$	<b>slt</b> rd, r1, rr	set less-than $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) < \text{bv2int}(\text{Reg}[rr])$
<b>sv</b> vd, r1, rr	store vector $\text{for}(i=0..15)\text{Mem}[\text{Reg}[r1] + \text{bv2nat}(\text{Reg}[rr]) + i] = \text{Vec}[vd][i]$	<b>sltu</b> rd, r1, rr	set less-than-unsigned $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) < \text{bv2nat}(\text{Reg}[rr])$
<b>lws</b> vd, r1, rr	load vec. w. stride $\text{for}(i=0..15)\text{Vec}[vd][i] = \text{Mem}[\text{Reg}[r1] + i * \text{bv2nat}(\text{Reg}[rr])] = \text{Vec}[vd][i]$	<b>sle</b> rd, r1, rr	set less-than-equal $\text{Reg}[rd] = \text{bv2int}(\text{Reg}[r1]) \leq \text{bv2int}(\text{Reg}[rr])$
<b>sws</b> vd, r1, rr	store vec. w. stride $\text{for}(i=0..15)\text{Mem}[\text{Reg}[r1] + i * \text{bv2nat}(\text{Reg}[rr])] = \text{Vec}[vd][i]$	<b>sleu</b> rd, r1, rr	set less-than-equal-unsigned $\text{Reg}[rd] = \text{bv2nat}(\text{Reg}[r1]) \leq \text{bv2nat}(\text{Reg}[rr])$
register move instructions		<b>seq</b> rd, r1, rr	set equal $\text{Reg}[rd] = \text{Reg}[r1] == \text{Reg}[rr]$
<b>mov</b> rd, c	move signed constant to register $\text{Reg}[rd] = \text{bv2int}(c)$	<b>sne</b> rd, r1, rr	set not equal $\text{Reg}[rd] = \text{Reg}[r1] != \text{Reg}[rr]$
<b>ovf</b> rd	move overflow to register $\text{Reg}[rd] = \text{ovflw}$	scalar bitwise logic operations	
<b>mvtm</b> rd	move register to mask register $\text{vmask} = \text{Reg}[rd]$	<b>and</b> rd, r1, rr	bitwise and $\text{Reg}[rd] = \text{Reg}[r1] \& \text{Reg}[rr]$
<b>mvtl</b> rd	move register to vector length register $\text{vlen} = \text{Reg}[rd]$	<b>or</b> rd, r1, rr	bitwise or $\text{Reg}[rd] = \text{Reg}[r1]   \text{Reg}[rr]$
thread synchronization		<b>xor</b> rd, r1, rr	bitwise exclusive or $\text{Reg}[rd] = \text{Reg}[r1] \text{ xor } \text{Reg}[rr]$
<b>l1</b> rd, r1, c	load linked $\text{Reg}[rd] = \text{Mem}[\text{bv2nat}(\text{Reg}[r1]) + \text{bv2nat}(c)]$	<b>neg</b> rd, r1	bitwise negation $\text{Reg}[rd] = !\text{Reg}[r1]$
<b>sc</b> rd, r1, c	store conditional $\text{Mem}[\text{bv2nat}(\text{Reg}[r1]) + \text{bv2nat}(c)] = \text{Reg}[rd]$	<b>sft</b> rd, r1, rr	left shift of $\text{Reg}[r1]$ by $\text{Reg}[rr]$ bits into $\text{Reg}[rd]$
<b>sync</b>	write back dirty cache blocks $\text{Reg}[vd] = 11$	vector instructions	
branch/jump instructions		vector arithmetic/logic instructions	
<b>be</b> rd, c	branch if zero $\text{if}(\text{Reg}[rd] == 0) \text{pc} = \text{pc} + \text{bv2int}(c)$	<b>vadd</b> vd, v1, vr	add signed $\text{Vec}[vd] = \text{bv2int}(\text{Vec}[v1]) + \text{bv2int}(\text{Vec}[vr])$
<b>bnz</b> rd, c	branch if not zero $\text{if}(\text{Reg}[rd] != 0) \text{pc} = \text{pc} + \text{bv2int}(c)$	<b>vaddu</b> vd, v1, vr	add unsigned $\text{Vec}[vd] = \text{bv2nat}(\text{Vec}[v1]) + \text{bv2nat}(\text{Vec}[vr])$
<b>jmp</b> rd	jump to register address $\text{pc} = \text{pc} + \text{Reg}[rd]$	<b>vsb</b> vd, v1, vr	subtract signed $\text{Vec}[vd] = \text{bv2int}(\text{Vec}[v1]) - \text{bv2int}(\text{Vec}[vr])$
<b>j</b> c	jump to immediate address $\text{pc} = \text{pc} + \text{bv2int}(c)$	<b>vsbu</b> vd, v1, vr	subtract unsigned $\text{Vec}[vd] = \text{bv2nat}(\text{Vec}[v1]) - \text{bv2nat}(\text{Vec}[vr])$
pseudo instructions		<b>vmul</b> vd, v1, vr	multiply signed $\text{Vec}[vd] = \text{bv2int}(\text{Vec}[v1]) * \text{bv2int}(\text{Vec}[vr])$
<b>nop</b>	do nothing $\rightsquigarrow j$ 1	<b>vmulu</b> vd, v1, vr	multiply unsigned $\text{Vec}[vd] = \text{bv2nat}(\text{Vec}[v1]) * \text{bv2nat}(\text{Vec}[vr])$
<b>movr</b> rd, r1	register transfer $\rightsquigarrow$ <b>addiu</b> rd, r1, 0	<b>vdv</b> vd, v1, vr	divide signed $\text{Vec}[vd] = \text{bv2int}(\text{Vec}[v1]) / \text{bv2int}(\text{Vec}[vr])$
<b>mod</b> rd, r1, rr	remainder (signed) $\rightsquigarrow$ <b>div</b> rd, r1, rr, <b>ovf</b> rd	<b>vdv</b> vd, v1, vr	divide unsigned $\text{Vec}[vd] = \text{bv2nat}(\text{Vec}[v1]) / \text{bv2nat}(\text{Vec}[vr])$
<b>modu</b> rd, r1, rr	remainder (unsigned) $\rightsquigarrow$ <b>divu</b> rd, r1, rr, <b>ovf</b> rd	<b>vslt</b> rd, v1, vr	set less-than $\text{Vec}[rd] = \text{Vec}[v1] < \text{Vec}[vr]$
<b>modi</b> rd, r1, c	remainder immediate (signed) $\rightsquigarrow$ <b>divi</b> rd, r1, rr, <b>ovf</b> rd	<b>vsle</b> rd, v1, vr	set less-than-equal $\text{Vec}[rd] = \text{Vec}[v1] \leq \text{Vec}[vr]$
<b>modiu</b> rd, r1, c	remainder immediate (unsigned) $\rightsquigarrow$ <b>diviu</b> rd, r1, rr, <b>ovf</b> rd	<b>vsgeu</b> rd, v1, vr	set greater-than-unsigned $\text{Vec}[rd] = \text{Vec}[v1] \geq \text{Vec}[vr]$
<b>sgt</b> rd, r1, rr	set greater than (signed) $\rightsquigarrow$ <b>slt</b> rd, rr, r1	<b>vsne</b> rd, v1, vr	set not equal $\text{Vec}[rd] = \text{Vec}[v1] != \text{Vec}[vr]$
<b>sgtu</b> rd, r1, rr	set greater than (unsigned) $\rightsquigarrow$ <b>sltu</b> rd, rr, r1	<b>vand</b> vd, v1, vr	bitwise and $\text{Vec}[vd] = \text{Vec}[v1] \& \text{Vec}[vr]$
<b>sge</b> rd, r1, rr	set gt. than/eq. to (signed) $\rightsquigarrow$ <b>sle</b> rd, rr, r1	<b>vor</b> vd, v1, vr	bitwise or $\text{Vec}[vd] = \text{Vec}[v1]   \text{Vec}[vr]$
<b>sgeu</b> rd, r1, rr	set gt. than/eq. to (unsigned) $\rightsquigarrow$ <b>sltu</b> rd, rr, r1	<b>vxor</b> vd, v1, vr	bitwise exclusive-or $\text{Vec}[vd] = \text{Vec}[v1] \text{ xor } \text{Vec}[vr]$
<b>slt</b> rd, r1, rr, c	branch on less-than (signed) $\rightsquigarrow$ <b>slt</b> rd, r1, rr; <b>bnz</b> rd, c	<b>vneg</b> vd, v1	bitwise negation $\text{Vec}[vd] = !\text{Vec}[v1]$
<b>sltu</b> rd, r1, rr, c	branch on less-than (unsigned) $\rightsquigarrow$ <b>sltu</b> rd, r1, rr; <b>bnz</b> rd, c	<b>vsft</b> rd, r1, rr	left shift of $\text{Vec}[r1]$ by $\text{Reg}[rr]$ bits into $\text{Vec}[rd]$
<b>sle</b> rd, r1, rr, c	branch on less-equal (signed) $\rightsquigarrow$ <b>sle</b> rd, r1, rr; <b>bnz</b> rd, c		
<b>sleu</b> rd, r1, rr, c	branch on less-equal (unsigned) $\rightsquigarrow$ <b>sleu</b> rd, r1, rr; <b>bnz</b> rd, c		
<b>bgd</b> rd, r1, rr, c	branch on greater-than (signed) $\rightsquigarrow$ <b>slt</b> rd, rr, r1; <b>bnz</b> rd, c		
<b>bgdu</b> rd, r1, rr, c	branch on greater-than (unsigned) $\rightsquigarrow$ <b>sltu</b> rd, rr, r1; <b>bnz</b> rd, c		
<b>bge</b> rd, r1, rr, c	branch on greater-equal (signed) $\rightsquigarrow$ <b>sle</b> rd, rr, r1; <b>bnz</b> rd, c		
<b>bgeu</b> rd, r1, rr, c	branch on greater-equal (unsigned) $\rightsquigarrow$ <b>sleu</b> rd, rr, r1; <b>bnz</b> rd, c		



## MiniC Language Reference Card

conventions used in reference card		expressions	
$\sigma, \sigma_1, \sigma_2$	boolean expressions	<b>type casts</b>	
$\tau, \pi$	general expressions	<b>(nat)</b> $\tau$	interprets $\tau$ as type <b>nat</b>
$n, m$	compile-time constant expressions	<b>(int)</b> $\tau$	interprets $\tau$ as type <b>int</b>
$\alpha_1, \alpha_2$	data types	<b>(bool)</b> $\tau$	interprets $\tau$ as type <b>bool</b>
module import and implementation		constructing and accessing compound types	
<b>package</b> pointedName	define root path for importing modules relative to current dir.	$\tau[\pi]$	array access
<b>include</b> pointedName	include textfile	$[(\tau_0, \dots, \tau_{n-1})]$	array of $n$ values
<i>// comment</i>	single line comment	$\tau.n$	tuple access
<i>/* comment */</i>	block comment (mult. lines)	$(\tau_0, \dots, \tau_{n-1})$	tuple of $n$ values
<b>function</b> $f(vdcl) : \alpha \{$ $stat$ $\}$	function $f$ with variable declarations $vdcl$ , body statement $stat$ and result type $\alpha$	$\tau_1 == \tau_2$	equality
		$\tau_1 != \tau_2$	inequality
variable declarations $vdcl ::=$		numeric relations (for both nat and int)	
general syntax is a comma-separated list of single declarations type $x_1, \dots, x_n$ , e.g. <b>nat</b> $x_1, x_2$ , <b>int</b> $z_1, z_2$		$\tau_1 < \tau_2$	less than
		$\tau_1 \leq \tau_2$	less than or equal to
		$\tau_1 > \tau_2$	greater than
		$\tau_1 \geq \tau_2$	greater than or equal to
data types $type ::=$		boolean operators	
<b>bool</b>	booleans	$! \sigma$	negation
<b>nat</b>	unsigned integers (machine dependent)	$\sigma_1 \& \sigma_2$	conjunction
<b>int</b>	signed integers (machine dependent)	$\sigma_1 \& \sigma_2$	disjunction
$[n] \alpha$	array having $n$ elements of type $\alpha$	$\sigma_1 \wedge \sigma_2$	exclusive or
$\alpha_1 * \dots * \alpha_n$	tuple type	$\sigma_1 \rightarrow \sigma_2$	implication
literals		$\sigma_1 \leftarrow \sigma_2$	equivalence
boolean constants are <b>false</b> and <b>true</b> ; examples for signed integers are 0, 1, 2, 3, ... while signed integers ..., -2, -1, -0, +0, +1, +2, +3, ...		arithmetic operators (for both nat and int)	
		$\tau + \pi$	addition
		$\tau - \pi$	subtraction
		$\tau * \pi$	multiplication
		$\tau / \pi$	division
		$\tau \% \pi$	modulo
		<b>abs</b> ( $\tau$ )	absolute value
		function call	
		$f(\tau_1, \dots, \tau_n);$	call function $f$ with parameter expressions $\tau_1, \dots, \tau_n$

statements $stat ::=$	
atomic statements	
$\lambda = \tau$	single word assignment
$\lambda_1, \lambda_2 = \tau$	double word assignment
$[name :] \textbf{assert}(\sigma);$	assertion
<b>sync</b>	thread synchronisation
composed statements	
<b>if</b> $(\sigma) S_1$ [ <b>else</b> $S_2$ ]	conditional statement
$S_1 S_2$	sequential execution
$\{ \alpha \ x; S \}$	declare variable $x$ of type $\alpha$ with scope $S$
<b>do</b> $S$ <b>while</b> $(\sigma)$	repeat $S$ while $\sigma$ holds
<b>while</b> $(\sigma) S$	while $\sigma$ holds, repeat $S$
<b>for</b> $(i=m .. n) S$	unconditional loop
<b>return</b> $\tau$	return value $\tau$
remarks on function calls	
the following restrictions apply	
<ul style="list-style-type: none"><li>• no recursive functions: a function is not allowed to call itself, not even via other function calls</li><li>• arguments of scalar types are provided via call-by-value, arrays and tuples via call-by-reference (hence the latter are potentially overwritten by the function)</li></ul>	





## 3-Address Code (Cmd) as Intermediate Language

- we will use 3-address like code (Cmd) as IL
- data types are only fixed sized bitvectors (of some unspecified bitwidth  $N$ )
- binary operators are

$$Op2 := \{+N, -N, *N, /N, \%N\} \cup \{<N, <=N\} \cup \\ \{+, -, *, /, \%\} \cup \{<, <=\} \cup \\ \{&, !\} \cup \{==, !=\}$$

- note that the arithmetic operators distinguish between unsigned (radix-2) and signed (2-complement) arithmetic
- equality (==) and inequality (!=) the same for all types

23 / 162

## 3-Address Code: Syntax

atomic statements and expressions are

- $y := x$  for variables and  $y := c$  for constants
- $y := x1 \odot x2$  for  $\odot \in Op2$
- $y1, y2 := x1 \odot x2$  for  $\odot \in Op2$
- $y := x1[x2]$
- $y[x1] := x2$
- goto  $L$ , where  $L$  is a line number
- if  $x$  goto  $L$ , where  $L$  is a line number
- return  $x$
- sync

a program consists of a sequence of the above statements, where each command is written to a single line with a unique number

24 / 162