DEPARTMENT OF ELECTRONIC AND ELECTRICAL ENGINEERING

Autumn Semester 2010-2011 (2 hours)

Answers to Advanced Computer Architectures (EEE411, EEE6031) Q1..4

1. *A pipelined system is required to implement the recurrence relationship for calculating sin and cos values as shown below:*

$cos(\theta+\Delta) = cos\theta.cos\Delta - sin\theta.sin\Delta$

$sin(\theta+\Delta) = sin\theta.cos\Delta + cos\theta.sin\Delta$

where Δ is a small constant angular difference and $\theta=0, \Delta, 2\Delta, 3\Delta, \ldots$

The pipeline (without any hardware for initialisation) is organised as shown in Figure 1.

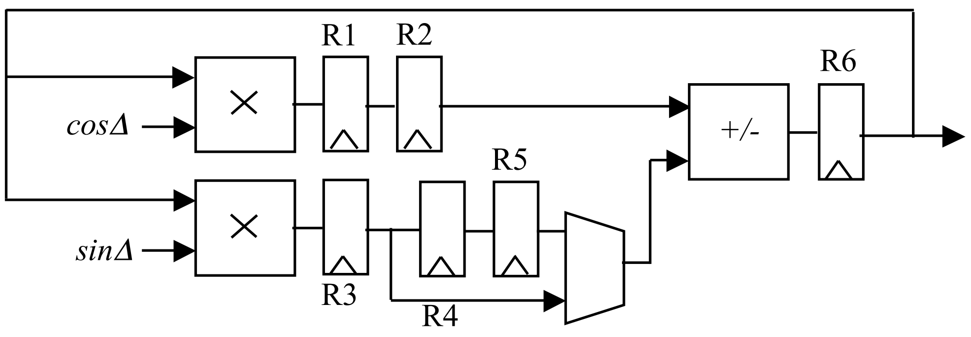


Figure 1: Pipelined System

a. *Draw the reservation table for the pipeline and hence ...*

Assume that the values fed back from R6 initially are $cos\theta$ and then $sin\theta$

T	R1	R2	R3	R4	R5	R6
1	$cos\theta cos\Delta$		$cos\theta sin\Delta$			
2	$sin\theta cos\Delta$	$cos\theta cos\Delta$	$sin\theta sin\Delta$	$cos\theta sin\Delta$		
3		$sin\theta cos\Delta$		$sin\theta sin\Delta$	$cos\theta sin\Delta$	$cos\theta cos\Delta$- $sin\theta sin\Delta$ $=cos(\theta+\Delta)$
4	$cos(\theta+\Delta)cos\Delta$		$cos(\theta+\Delta)sin\Delta$		$sin\theta sin\Delta$	$cos\theta sin\Delta$+ $sin\theta cos\Delta$ $=sin(\theta+\Delta)$
5	$sin(\theta+\Delta)cos\Delta$	$cos(\theta+\Delta)cos\Delta$	$sin(\theta+\Delta)sin\Delta$	$cos(\theta+\Delta)sin\Delta$		
6		$sin(\theta+\Delta)cos\Delta$		$sin(\theta+\Delta)sin\Delta$	$cos(\theta+\Delta)sin\Delta$	$cos(\theta+\Delta)cos\Delta$- $sin(\theta+\Delta)sin\Delta$ $=cos(\theta+2\Delta)$
7	$cos(\theta+2\Delta)cos\Delta$		$cos(\theta+2\Delta)sin\Delta$		$sin(\theta+\Delta)sin\Delta$	$cos(\theta+\Delta)sin\Delta$+ $sin(\theta+\Delta)cos\Delta$ $=sin(\theta+2\Delta)$
8	$sin(\theta+2\Delta)cos\Delta$	$cos(\theta+2\Delta)cos\Delta$	$sin(\theta+2\Delta)sin\Delta$	$cos(\theta+2\Delta)sin\Delta$		

(7)

b. *show how the successive values of cosθ and sinθ can be calculated using this hardware.*

The reservation table for the pipelined system shows that a sequential set of cos and sin values can be produced. However, to allow this to happen the multiplexer should be controlled to allow the lower input (from R3) to pass during timeslot 3, 6, 9, etc. and to let the upper input pass during timeslot 4, 7, 10, etc. Similarly, the adder/subtractor should produce lower-upper input during timeslot 3, 6, 9, etc. and lower+upper input during timeslot 4, 7, 10, etc. **(3)**

c. *How could the hardware be initialised to produce the sequence where $\theta = 0, \Delta, 2\Delta, 3\Delta, 4\Delta, 5\Delta, \ldots$*

The system needs cos0 (=1) and sin0 (=0) to emerge sequentially from R6 to start the process. The easiest way to achieve this would be to set the contents of R6 to be 1 and to clear the remaining registers R1..5 (they all need to be initialised, in any event) **(2)**

d. *What is the throughput of this system in terms of sin/cos pairs produced per clock cycle?*

One pair of sin/cos is produced every 3 clock cycles. **(2)**

e. *Clearly, the time taken to produce a particular sinθ/cosθ pair (θ being well away from the origin) and the accuracy with which it could be produced depends on the value of Δ. How could this process be made more rapid without sacrificing accuracy?*

For good accuracy, Δ needs to be small but for good speed of reaching the result Δ needs to be big. A compromise might be to store a range of sin/cos values for various values of Δ. That is, assuming first quadrant operation, $\pi/4$, $\pi/8$, etc. using a small value of Δ to bridge the gap to the final value with the required accuracy. Rather than just increasing the value, if the adder/subtractor were controlled to add instead of subtract or vice versa then Δ could be subtracted from the angle rather than added. In this case, for example, cos($\pi/4-\Delta$) could be produced much more quickly. **(6)**

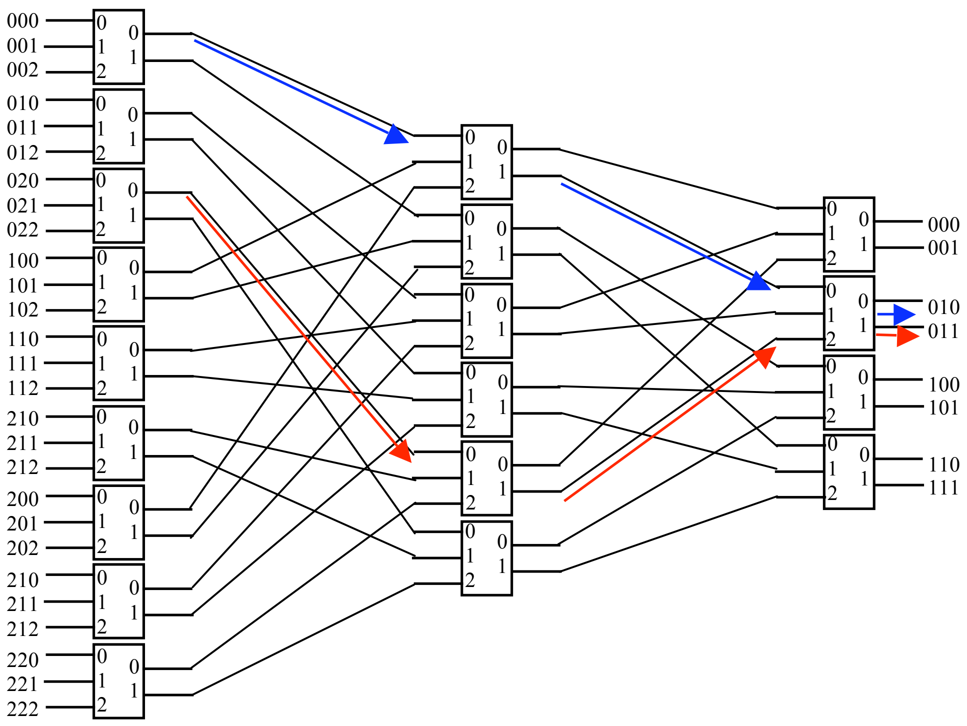
2. **a.** *What is the major difference between a blocking and non-blocking switch?*

In a non-blocking switch only an output being in use will prevent another route being established across the switch (directed to the same output). In a blocking switch, the existence of a particular routing may preclude another one being established – even if they do not share the same output. **(2)**

b. *What is the advantage of a blocking switch?*

Blocking switches tend to use fewer resources for the same size of switch – measured in terms of input and output ports. **(2)**

c. **i)** *Draw a schematic for a 3^3x2^3 Delta Network.*



(4)

ii) *Show how digit-selectable addressing works to route a message from a source on the left hand side of the network to a destination on the right hand side of the network.*

Representing the input address as a base-3 number and the output as a base-2 number, we can route from any input to a defined output by exiting via the port at each intermediate CPS defined by each digit of the output address. Consider output 011. Routing from input 022 (red arrows) or from input 001 (blue arrows), for example using the same routing information leads to the correct output. **(2)**

iii) *How many digits make up an address?*

3 digits (either input to output or vice versa). The number of digits is equal to the number of columns of CPS = *n* for an a^nxb^n Delta Network. **(1)**

iv) *What would the complexity (as measured by amount of logic) of this Delta Network be compared to CPS with the same number of input and output ports?*

Made up from *axb* CPS, an $a^n x b^n$ Delta Network has a^{n-1} CPS in the first column with $a^{n-1}b$ outputs. Hence the 2nd column has $a^{n-2}b$ CPS with $a^{n-2}b^2$ outputs and so on until the *n*-1th column has ab^{n-2} CPS with ab^{n-1} outputs and so the *n*th column has b^{n-1} CPS with b^n outputs (As required). Consequently, the total number of CPS is:

$$\sum_{i=0}^{n-1} a^{n-1-i} b^i$$

The complexity of an *axb* CPS is *kab* and so the complexity of the $a^n x b^n$ Delta Network is:

$$k\sum_{i=0}^{n-1} a^{n-i} b^{i+1}$$

whereas the complexity of an $a^n x b^n$ CPS is $a^n b^n$. In this case, the CPS will be of complexity 216*k*. Whilst the Delta Network will be of complexity 114*k*.

(3)

d. *A 4 input by 8 output CPS is used to transfer messages in a system. Each of the four sources is generating* $1x10^5$ *messages per second to route across the CPS, randomly distributed to the 8 outputs. It takes 5µs to transfer a message. If a message fails to be routed the source re-tries (note, this does not increase the rate at which the source transmits messages across the network) until the message is transferred.*

Estimate the average time taken to transfer each message.

The value of *R* the rate of message generation will be $2x10^5$ whilst the time for transmitting each message, T_{mess} will be $5x10^{-6}$, N=4, M=8. Consequently, p_A, the probability that a message will get through without contention is 0.83. However, if a message does not get through first time then it will need to be resent and so the average time taken to send a message will be:

$$p_A T_{mess} + 2(1-p_A)p_A T_{mess} + 3(1-p_A)^2 p_A T_{mess} + 4(1-p_A)^3 p_A T_{mess} + \ldots$$

$$\text{So, } T_{ave} = p_A T_{mess} \sum_{i=1}^{\infty} i(1-p_A)^{i-1}$$

Looking at the summation we can replace (1-p_A) by x, yielding

$$y = \sum_{i=1}^{\infty} i x^{i-1}$$

$$z = \int y.dx = \sum_{i=1}^{\infty} x^i + K = \frac{1}{1-x} - 1 + K$$

$$y = \frac{dz}{dx} = \frac{1}{(1-x)^2} = \frac{1}{p_A{}^2}$$

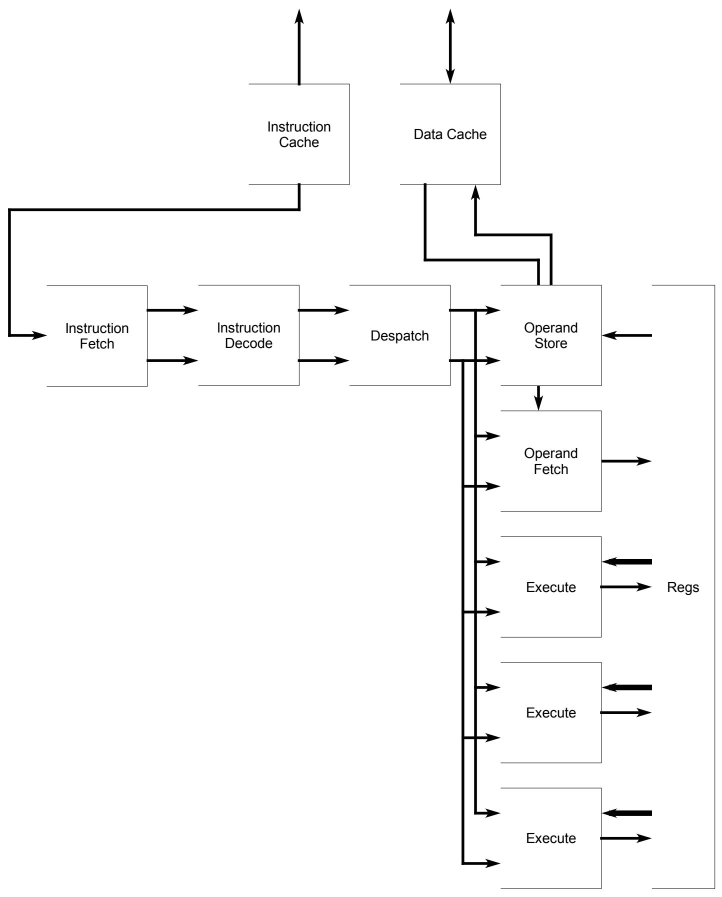
$$T_{ave} = p_A T_{mess} y = \frac{T_{mess}}{p_A} = 5x10^{-6}/0.83 = 6x10^{-6}\text{s}$$

(6)

You can use the following expression to help you:

$$p_A = \frac{M}{NRT_{mess}}\left(1-\left(1-\frac{RT_{mess}}{M}\right)^N\right)$$

3. a. *Describe, with the aid of a diagram, the organisation of a superscalar pipelined processor identifying the constraints that need to be imposed on the hardware and instruction architectures to make it work practically.*



Students should provide a diagram such as this. Issues that need to considered:

Control flow dependencies (e.g. BHC), data dependencies (e.g. register forwarding or reorder buffers), out of order execution, speculative execution (possibly), threading, grouping of instructions for execution. **(5)**

b. *What are the control dependency issues/problems that will affect the performance of such a processor?*

Describe, briefly, the possible solutions to the problems caused by control dependency.

Consider the following section of code:

```
ADD   R3,R1
CMP   R1,R2
JEQ   @target
```

We can see how these instructions progress down the pipeline.

clock cycle	Inst.Fetch	Inst.Decode	Opnd. Fetch	Execute	Operand Store
n	*ADD*				
n+1	*CMP*	*ADD*			
n+2	*JEQ*	*CMP*	*ADD*		
n+3	*?*	*JEQ*	*CMP*	*ADD*	
n+4	*?*	*?*	*JEQ*	*CMP*	*ADD*
n+5	*next instr.*	*?*	*?*	*JEQ*	*CMP*

At the point where a decision must be made about whether to continue fetching instructions contiguously or to start fetching from @target, neither the ADD nor the CMP instruction have been executed. Therefore, the state of the Z flag which controls whether the JEQ jumps is not yet valid. In fact the decision is not known until the CMP instruction clears the Execute stage and only at this point in time is the address of the next instruction known. In a superscalar processor the problem would be compounded because more instructions would be in-flight.

There are a number of ways of dealing with this problem:

Do nothing

The pipeline stalls - as shown above except that the ? would be No Operation instructions. This is very inefficient because, effectively in the case shown above, two non-useful instructions are executed every time a conditional jump is encountered.

Fixed Choice

Make a choice and accept the fact that if it is the wrong choice then the contents of the first part of the pipeline will need to be discarded. Of the two possible streams (A or B), the processor chooses stream A. In the first case, the choice is correct and the sequence of instructions carries on unbroken. In the second case, however, stream B should have been chosen and the contents of the Instruction Decode and Operand Fetch stages have to be invalidated and the correct stream must be fetched. This speculative execution is easier to implement with a reorder buffer where instructions are only retired once they have deemed to executed.

Branch History Cache

A Branch History Cache (BHC) may be employed. A BHC is a cache which records the choice made whenever a conditional jump is taken at a particular address (of the jump instruction in memory). Whenever a conditional jump is encountered, the instruction address is applied to the BHC and if there is a hit the choice made last time is used to direct which stream to fetch (A or B). When the true decision is known, the contents of the cache are updated and if the choice was wrong, the early part of the cache is flushed and the correct stream is fetched.

In the interests of simplicity, BHCs are usually simplified so that only a sub-set of the instruction address bits are used which makes the decision inexact. For example, if every odd bit was used (halving the width of the cache) then two conditional jumps at addresses which shared the same values of odd address bits would be confused. This only means that the probability of an incorrect choice is increased - no damage results from this.

Some BHCs employ a scheme whereby the wrong decision must be made twice in succession before the contents of the BHC are updated. This caters for the case of nested loops where the wrong decision will inevitably be made upon exit from an inner loop. If the BHC was updated at this point then it would also make the wrong decision when the inner loop was next encountered (due to an outer loop). If two consecutive errors were required to change the BHC then it would still make a mistake on exiting the inner loop but would retain the correct information for when it next encountered the loop. Hence the rate at which it would make this sort of mistake would be halved and performance would improve.

Delayed Jumps

Delayed Jumps are an intuitive approach which depends upon the compiler. As the name suggests, a delayed jump is one whose effect is delayed relative to its position in the program. That is, it causes a jump some time after its position in the program - not at the point of its occurrence. Consider the initial example with some prior instructions shown (in this case the effect of the jump is delayed by 3 instructions):

```
inst1
inst2
inst3
ADD   R3,R1
CMP   R1,R2
JEQ   @target
```

To operate effectively, the compiler must be able to find a number of prior instructions which are definitely executed *but on which the result of the comparison does not depend*. The compiler re-orders the instruction stream and changes the JEQ to DJEQ (delayed jump equal) as follows:

```
ADD   R3,R1
CMP   R1,R2
DJEQ  @target
inst1
inst2
inst3
```

This re-ordering gives rise to the activity in the pipeline as shown.

clock cycle	**Inst.Fetch**	**Inst.Decode**	**Opnd. Fetch**	**Execute**	**Operand Store**
n	*ADD*				
n+1	*CMP*	*ADD*			
n+2	*DJEQ*	*CMP*	*ADD*		
n+3	*Inst1*	*DJEQ*	*CMP*	*ADD*	
n+4	*Inst2*	*Inst1*	*DJEQ*	*CMP*	*ADD*

2+3 = (5)

c. *A superscalar processor uses register forwarding to resolve data interlock problems. Show how this would operate with the following segment of a program:*

```
LOAD @addr,R1        ; R1 = (addr)
ADD  R1,R3,R2        ; R2 = R1 + R3
ADD  R2,R3,R2        ; R2 = R2 + R3
SUB  R2,R4,R1        ; R1 = R4 - R2
```

Assume R1 and R2 are not committed, that R3 points to DR1 (contains X) and is READY and R4 points to DR2 (contains Y) and is READY.

The LOAD instruction: R1 is set to point to a free data register, DR3 which is set to BUSY, and addr and DR3 are sent to the Reservation Station for the LOAD EU. The unit commences retrieval of the data from addr.

ADD R1,R3,R2: R2 is set to point to a free data register DR4, which is set to BUSY. The ADD instruction along with DR3 (proxy for R1) and the data from DR1, X, along with DR4 (for the result) are sent to the Reservation Station of an EU (1) capable of undertaking an ADD.

ADD R2, R3, R2: R2 points to DR4, which is BUSY, so R2 is set to a free data register DR5, which is set to BUSY. The ADD instruction along with DR4 (proxy for R2) and the data from DR1, X, along with DR5 (for the result) are sent to the Reservation Station of an EU (2) capable of undertaking an ADD – this operation may be grouped with the previous operation.

SUB R2, R4, R1: R1 points to DR3, which is BUSY, and so R1 is set to point to DR6, which is set to BUSY. The SUB instruction along with DR5 (proxy for R2) and the data from DR2, Y, along with DR6 (for the result) are sent to the Reservation Station of an EU (3) capable of undertaking a SUB.

The data from addr arrives and is forwarded to DR3 (which is set to READY) and to EU1. The first ADD operation takes place and the result is forwarded to DR4 (which is set to IDLE – R2 no longer points to it) and EU2. The second ADD takes place and result is forwarded to DR5 (which is set to READY) and to EU3. The SUB operation takes place and the result is forwarded to DR6 (which is set to READY). **(10)**

4. *A network of n processors is organised as shown in Figure 4. They communicate via a single time-shared bus.*

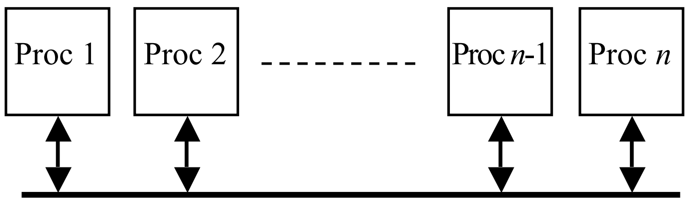


Figure 4: Network Organisation

A set of m tasks is to be partitioned across these n processors. Each task exhibits a similar set of characteristics:

- *each task's internal processing time is r seconds;*
- *each task communicates equally with every other task sending p messages to each other task;*
- *the time taken to transfer a message across the bus is t_b seconds and the time taken to transfer a message between two tasks on the same processor is t_i seconds where $t_b = kt_i$;*
- *internal processing within a task is not overlapped with its communication.*

a. *Show that the speed-up when the m tasks are mapped onto the n processors (over and above the speed achieved by mapping them all on to one of the processors) is:*

$$speed-up = \frac{n\left(\frac{r}{pt_i}+m-1\right)}{\frac{r}{pt_i}+\left(\frac{m}{n}-1\right)+km(n-1)}$$

You may assume that congestion on the bus is not a problem.

State all assumptions made.

n processors, m tasks. Therefore, m/n tasks per processor.

Each task communicates with m-1 other tasks and so the total number of uni-directional task-task links is $m(m-1)$. However, with m/n tasks per processor, the total number of internal task-task links on each processor is $(m/n)(m/n-1)$ and the total number of such internal links across the n processors is $m(m/n-1)$. Consequently, the total number of links across the external bus is $m(m-1)-m(m/n-1)=m^2(n-1)/n$.

The total time spent on external communication is, therefore, $pt_bm^2(n-1)/n$.

The total time spent on internal communication on each processor is $pt_im(m/n-1)/n$. This time is overlapped with the time spent similarly on other processors so it does not accumulate across all the processors.

The time spent internally processing is rm/n.

Consequently, the execution time is $rm/n+pt_im(m/n-1)/n+pt_bm^2(n-1)/n$. The computation is not overlapped with processing but computation and internal communication run in parallel across the processors. So:

$T_n = rm/n + pt_i m(m/n-1)/n + kpt_i m^2(n-1)/n$

To check:

$T_1 = rm + pt_i m(m-1)$

$$speedup = \frac{rm + pt_i m(m-1)}{r\frac{m}{n} + pt_i\frac{m}{n}\left(\frac{m}{n}-1\right) + kpt_i\left(\frac{n-1}{n}\right)}$$

$$= \frac{n\left(r + pt_i(m-1)\right)}{r + pt_i\left(\frac{m}{n}-1\right) + kpt_i m(n-1)}$$

$$= \frac{n\left(\frac{r}{pt_i} + m - 1\right)}{\frac{r}{pt_i} + \left(\frac{m}{n}-1\right) + km(n-1)}$$

QED **(8)**

b. **i)** *Show that if the m, number of tasks, dominates, the speed-up is approximately equal to 1/k.*

As m goes to infinity, the numerator goes to nm and the denominator goes to $m/n+kmn$. Assuming that n is not small, we can consider this to be kmn. Consequently, the ratio goes to approximately, $1/k$. **(2)**

ii) *What does this mean result imply?*

This result implies that as the number of tasks becomes very large relative to the number of processors the need to transmit a vast number of messages across the time-shared bus becomes the dominant factor in the performance. Indeed the performance does not go up, it goes down. This indicates that the application is a very poor candidate for parallelising – particularly in this kind of architecture where excessive external communication carries a significant penalty. **(4)**

c. *A particular application consists of 100 tasks where k=10. What ratio of r/pt_i is needed to make parallelisation just worthwhile.*

$$speed-up = \frac{n\left(\frac{r}{pt_i} + m - 1\right)}{\frac{r}{pt_i} + \left(\frac{m}{n}-1\right) + km(n-1)}$$

$$= \frac{n\left(\frac{r}{pt_i} + 99\right)}{\frac{r}{pt_i} + \left(\frac{100}{n}-1\right) + 1000(n-1)}$$

speed-up should be minimally 1+ with more than one processor for there to be any benefit. So with n=2 representing the best opportunity for this to happen: **(6)**

$$1 \le \frac{2\left(\frac{r}{pt_i} + 99\right)}{\frac{r}{pt_i} + 49 + 1000}$$

$$\frac{r}{pt_i} + 49 + 1000 \le 2\left(\frac{r}{pt_i} + 99\right)$$

$$1049 - 198 \le 2\frac{r}{pt_i}$$

$$\frac{r}{pt_i} \ge 425.5$$

$$\frac{r}{pt_i} = 426$$

To check, when *n*=3, the corresponding value is:

$$1 \le \frac{3\left(\frac{r}{pt_i} + 99\right)}{\frac{r}{pt_i} + 32.33 + 2000}$$

$$\frac{r}{pt_i} + 32.33 + 2000 \le 3\left(\frac{r}{pt_i} + 99\right)$$

$$2032.33 - 297 \le 3\frac{r}{pt_i}$$

$$\frac{r}{pt_i} \ge 578.4$$

$$\frac{r}{pt_i} = 579$$

So, the case where *n*=2 does represent a minimal condition.

NLS